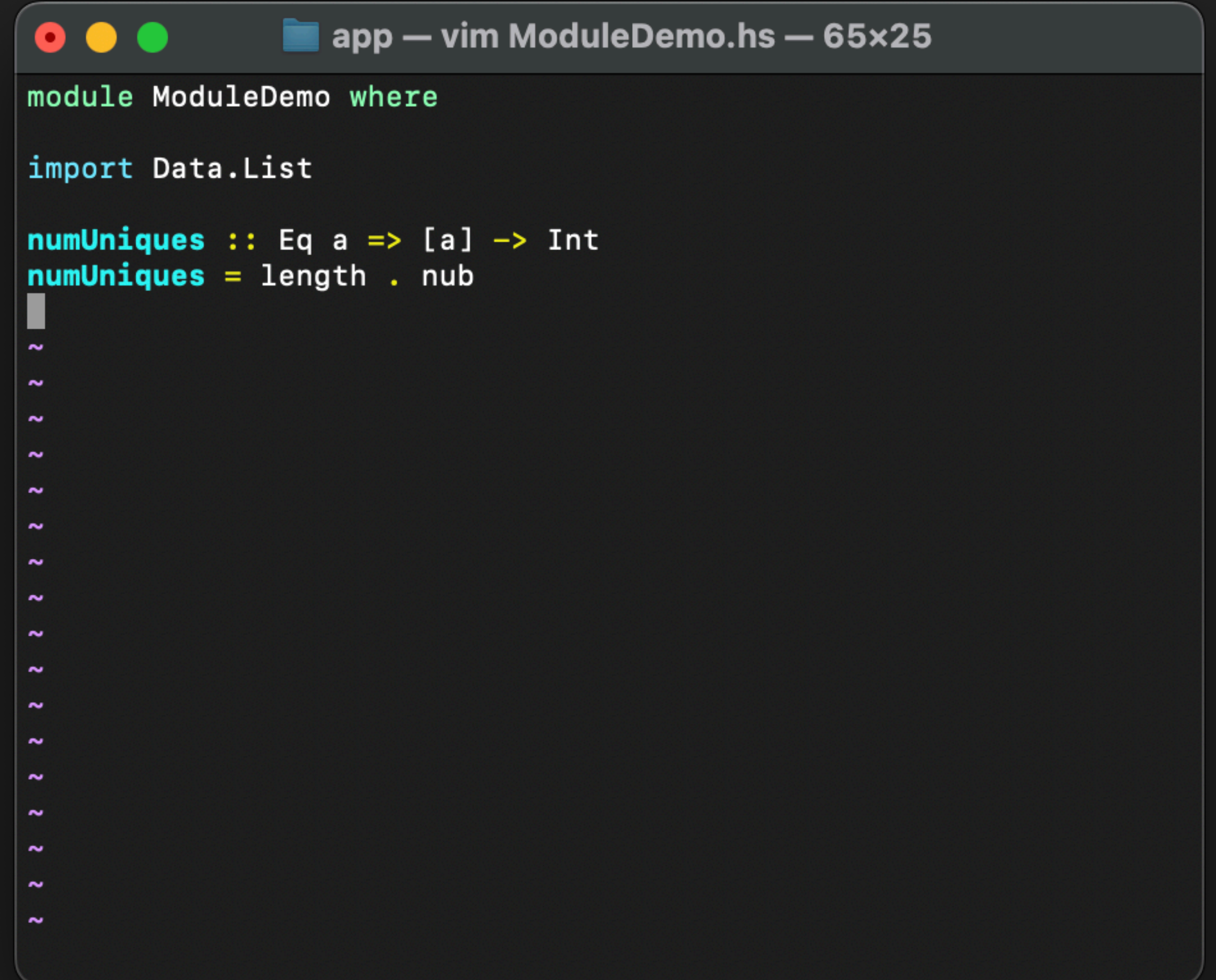


INTRODUCTION TO

FUNCTIONAL PROGRAMMING

MODULE SYSTEM: IMPORT

- ▶ Module is a collection of related functions, types, and typeclasses
- ▶ Use `import moduleName` to import everything
- ▶ Add `(f, g)` to import only `f` and `g`
- ▶ Hide some names with `hiding`
- ▶ Use fully qualified names with `qualified`



```
module ModuleDemo where

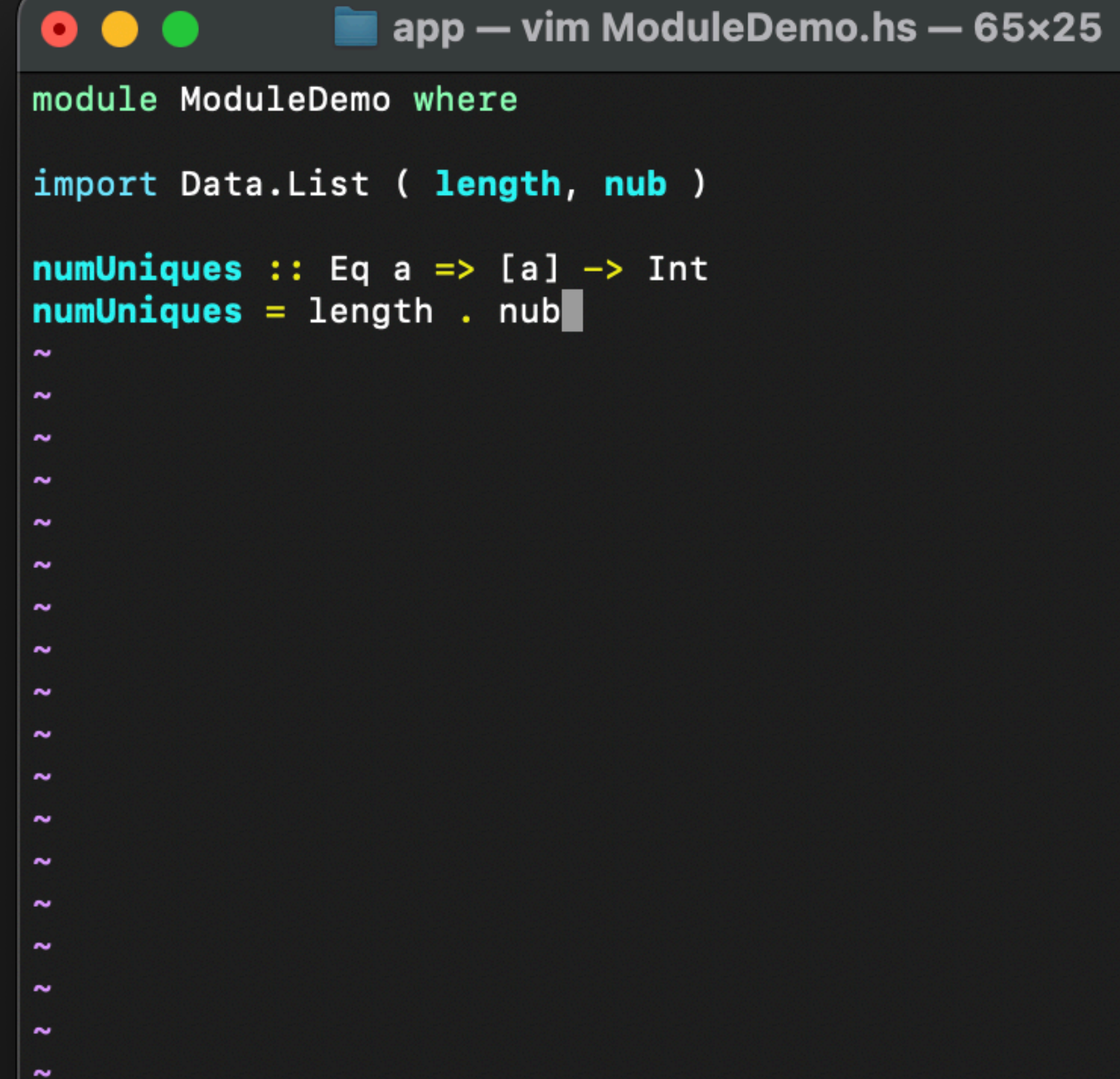
import Data.List

numUniques :: Eq a => [a] -> Int
numUniques = length . nub

~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
```

MODULE SYSTEM: IMPORT

- ▶ Module is a collection of related functions, types, and typeclasses
- ▶ Use `import moduleName` to import everything
- ▶ Add `(f, g)` to import only `f` and `g`
- ▶ Hide some names with `hiding`
- ▶ Use fully qualified names with `qualified`



```
module ModuleDemo where

import Data.List ( length, nub )

numUniques :: Eq a => [a] -> Int
numUniques = length . nub

~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
```

MODULE SYSTEM: IMPORT

- ▶ Module is a collection of related functions, types, and typeclasses
- ▶ Use `import moduleName` to import everything
- ▶ Add `(f, g)` to import only `f` and `g`
- ▶ Hide some names with `hiding`
- ▶ Use fully qualified names with `qualified`

```
app — vim ModuleDemo.hs — 65x25
module ModuleDemo where

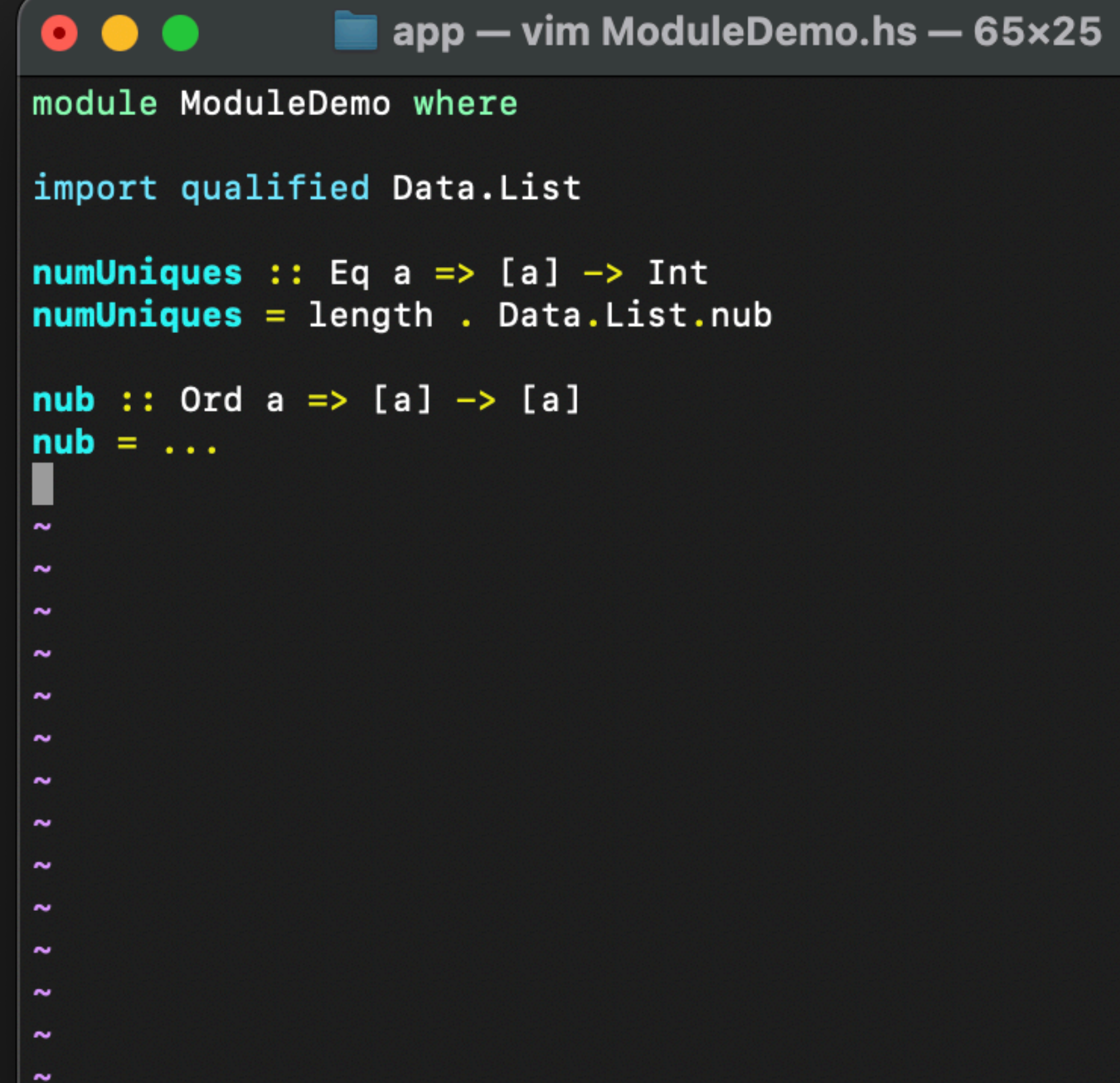
import Data.List hiding ( nub )

numUniques :: Eq a => [a] -> Int
numUniques = length . nub

nub :: Ord a => [a] -> [a]
nub = ...
~
~
~
~
~
~
~
~
~
~
~
~
~
```


MODULE SYSTEM: IMPORT

- ▶ Module is a collection of related functions, types, and typeclasses
- ▶ Use `import moduleName` to import everything
- ▶ Add `(f, g)` to import only `f` and `g`
- ▶ Hide some names with `hiding`
- ▶ Use fully qualified names with `qualified`



```
module ModuleDemo where

import qualified Data.List

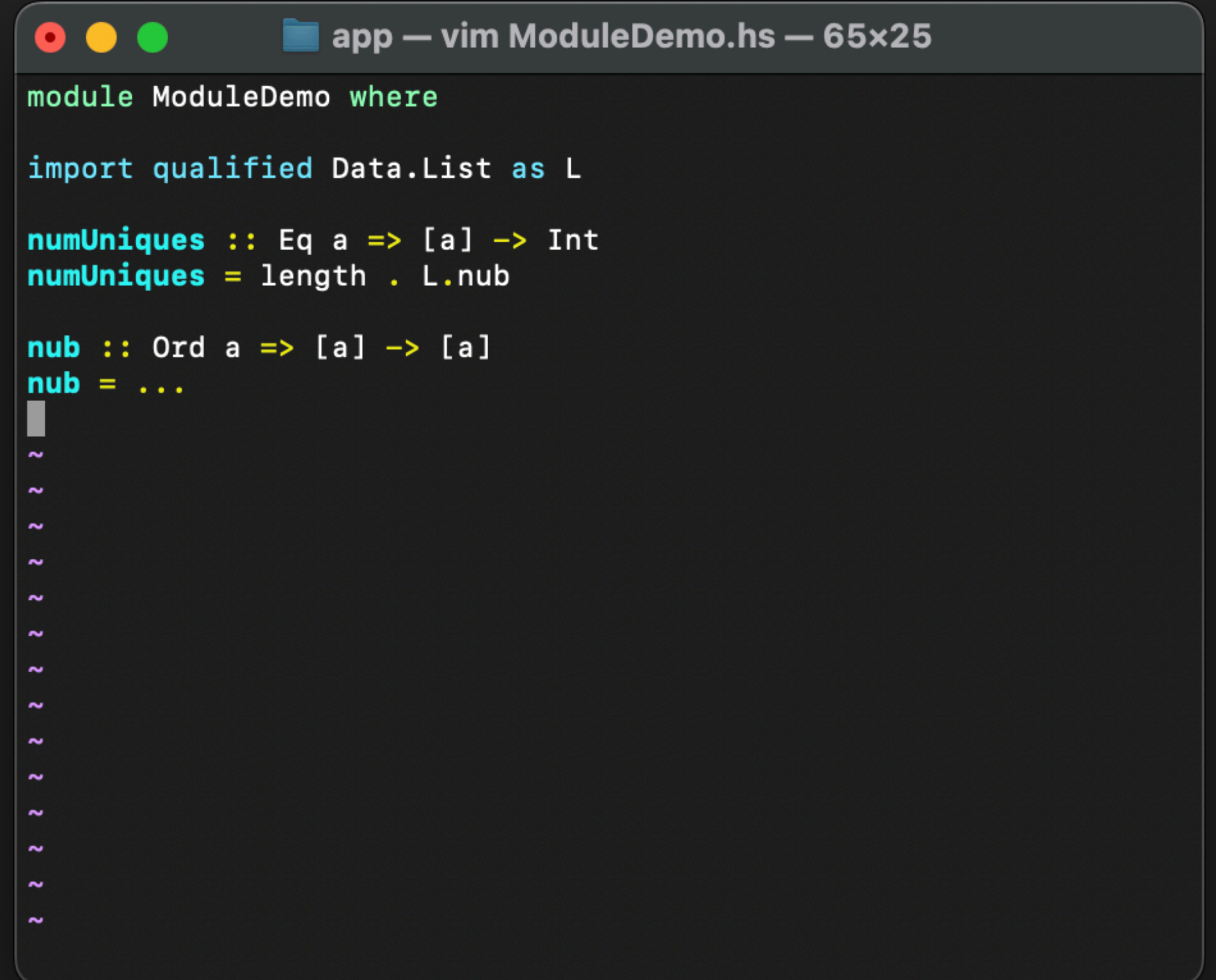
numUniques :: Eq a => [a] -> Int
numUniques = length . Data.List.nub

nub :: Ord a => [a] -> [a]
nub = ...

~
~
~
~
~
~
~
~
~
~
~
~
```

MODULE SYSTEM: IMPORT

- ▶ Module is a collection of related functions, types, and typeclasses
- ▶ Use `import moduleName` to import everything
- ▶ Add `(f, g)` to import only `f` and `g`
- ▶ Hide some names with `hiding`
- ▶ Use fully qualified names with `qualified`



```
module ModuleDemo where

import qualified Data.List as L

numUniques :: Eq a => [a] -> Int
numUniques = length . L.nub

nub :: Ord a => [a] -> [a]
nub = ...

~
~
~
~
~
~
~
~
~
~
~
~
```

MODULE SYSTEM: EXPORT

- ▶ Everything on the top level of the module is exported by default
- ▶ You can add names in parentheses to export only them
- ▶ You can re-export names from the imported modules
 - ▶ By default, only names defined in the module are exported

```
app — vim ModuleDemo.hs — 65x25
module ModuleDemo where

import qualified Data.List as L

numUniques :: Eq a => [a] -> Int
numUniques = length . L.nub

nub :: Ord a => [a] -> [a]
nub = undefined

~
~
~
~
~
~
~
~
~
~
~
```


MODULE SYSTEM: EXPORT

- ▶ Everything on the top level of the module is exported by default
- ▶ You can add names in parentheses to export only them
- ▶ You can re-export names from the imported modules
 - ▶ By default, only names defined in the module are exported

app — vim ModuleDemo.hs — 65x25

```
module ModuleDemo ( numUniques ) where
```

```
import qualified Data.List as L
```

```
numUniques :: Eq a => [a] -> Int
```

```
numUniques = length . L.nub
```

```
nub :: Ord a => [a] -> [a]
```

```
nub = undefined
```

```
~
```

```
~
```

```
~
```

```
~
```

```
~
```

```
~
```

```
~
```

```
~
```

```
~
```

```
~
```

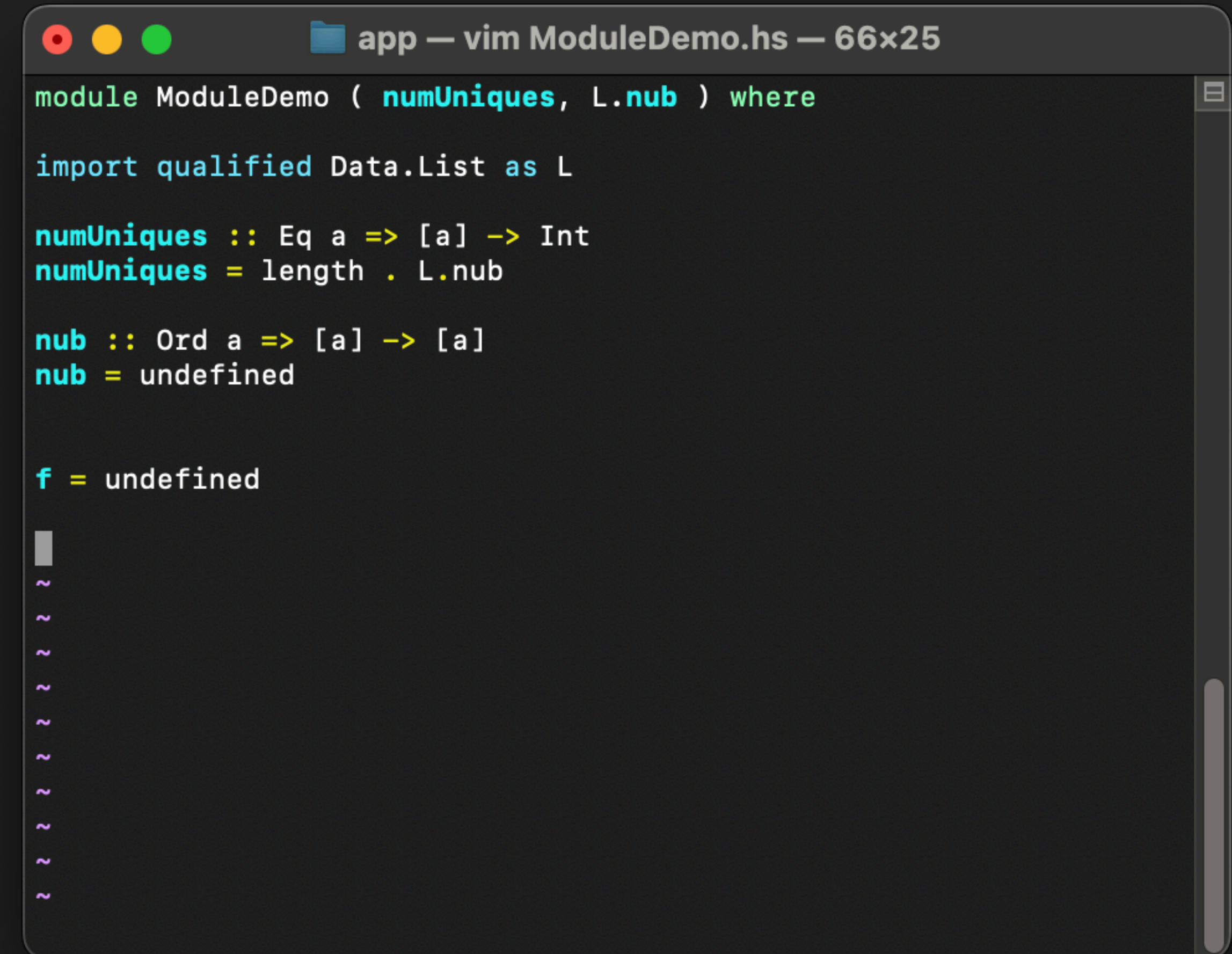
```
~
```

```
~
```

```
~
```

MODULE SYSTEM: EXPORT

- ▶ Everything on the top level of the module is exported by default
- ▶ You can add names in parentheses to export only them
- ▶ You can re-export names from the imported modules
 - ▶ By default, only names defined in the module are exported



```
app — vim ModuleDemo.hs — 66x25
module ModuleDemo ( numUniques, L.nub ) where

import qualified Data.List as L

numUniques :: Eq a => [a] -> Int
numUniques = length . L.nub

nub :: Ord a => [a] -> [a]
nub = undefined

f = undefined

~
~
~
~
~
~
~
~
~
~
```


MODULE SYSTEM: EXPORT

- ▶ Everything on the top level of the module is exported by default
- ▶ You can add names in parentheses to export only them
- ▶ You can re-export names from the imported modules
- ▶ By default, only names defined in the module are exported

The image shows three terminal windows illustrating Haskell module exports and a compilation error.

Top Left Window: app — vim ModuleDemo.hs — 66x25

```
module ModuleDemo ( numUniques, L.nub ) where

import qualified Data.List as L

numUniques :: Eq a => [a] -> Int
numUniques = length . L.nub

nub :: Ord a => [a] -> [a]
nub = undefined

f = undefined
```

Top Right Window: app — vim Main.hs

```
module Main (main) where

import ModuleDemo

main = do
  let xs = [1,1,1,2]
  print $ numUniques xs
  print $ nub xs
  print $ f xs
```

Bottom Window: app — -zsh — 66x25

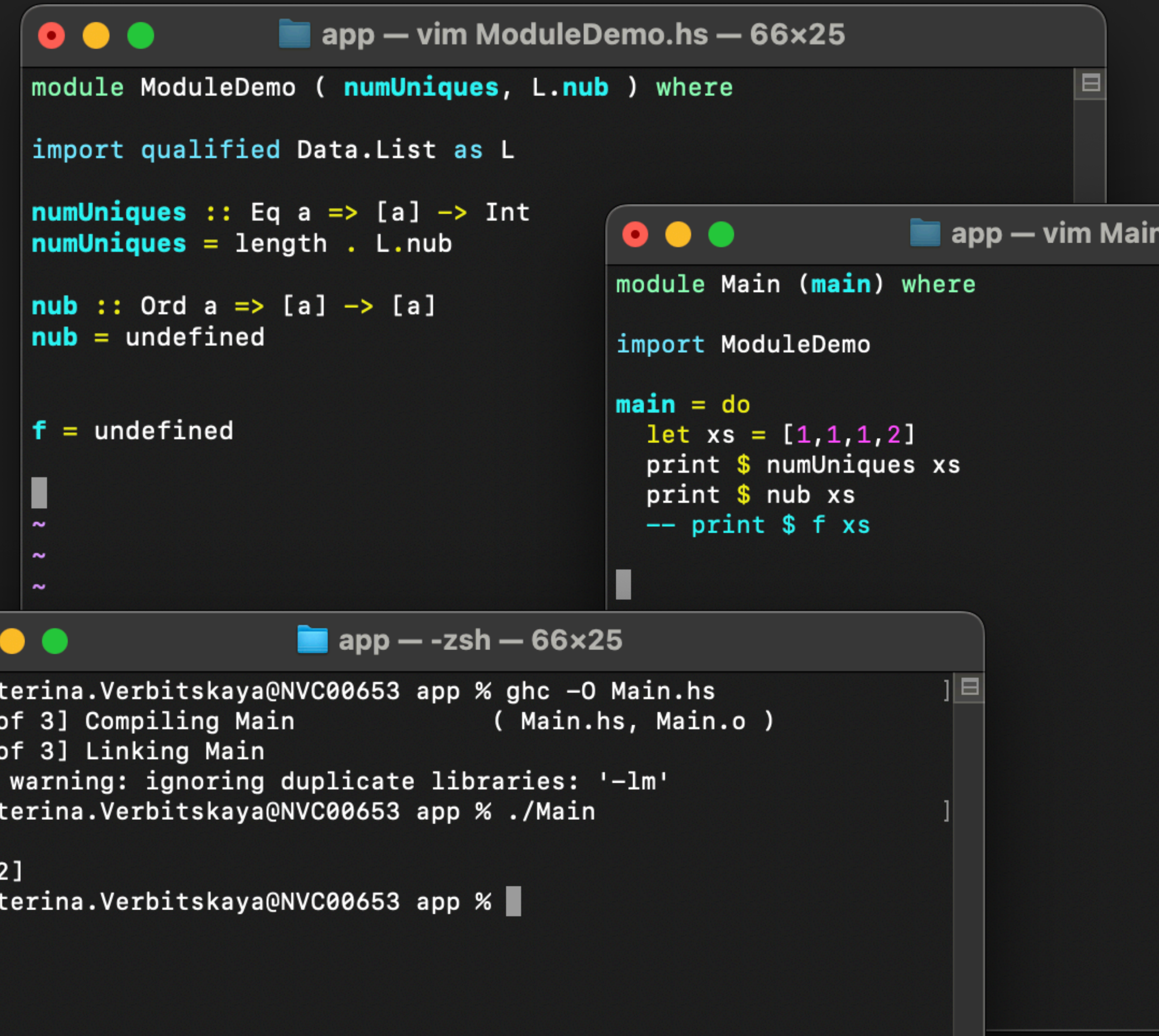
```
[Ekaterina.Verbitskaya@NVC00653 app % ghc -O Main.hs]
[1 of 3] Compiling ModuleDemo      ( ModuleDemo.hs, ModuleDemo.o )
[Source file changed]
[2 of 3] Compiling Main              ( Main.hs, Main.o )

Main.hs:9:11: error: Variable not in scope: f :: [a0] -> a1
9 |   print $ f xs
  |           ^
```

Ekaterina.Verbitskaya@NVC00653 app %

MODULE SYSTEM: EXPORT

- ▶ Everything on the top level of the module is exported by default
- ▶ You can add names in parentheses to export only them
- ▶ You can re-export names from the imported modules
- ▶ By default, only names defined in the module are exported

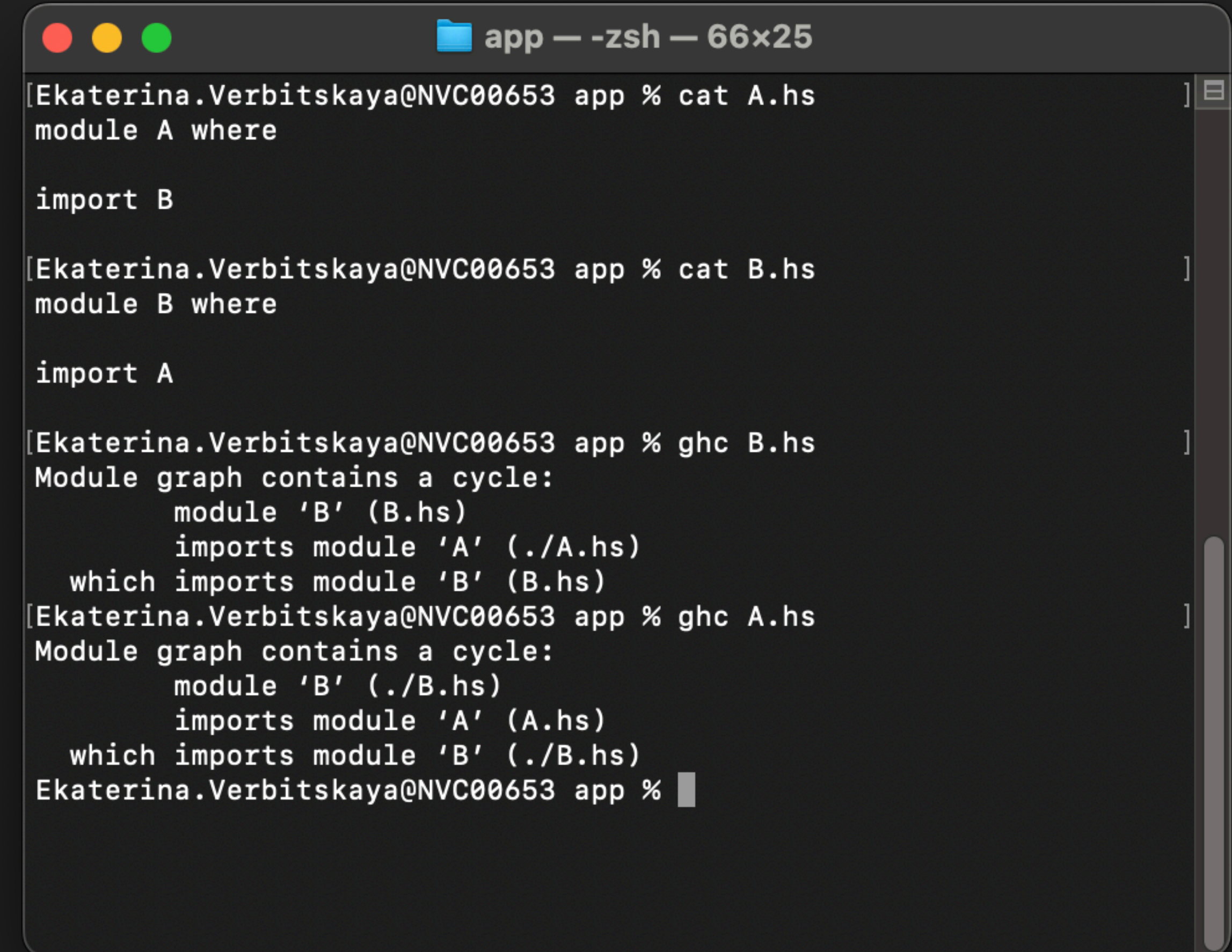


The image displays three terminal windows illustrating Haskell module usage:

- Top Left Window (vim ModuleDemo.hs):** Shows the definition of the `ModuleDemo` module. It imports `Data.List` as `L` and defines `numUniques` (a function returning the length of a list), `nub` (an undefined function), and `f` (an undefined function).
- Top Right Window (vim Main.hs):** Shows the definition of the `Main` module. It imports `ModuleDemo` and defines `main` to use `numUniques` and `nub` on the list `[1,1,1,2]`. It also includes a commented-out line for `f`.
- Bottom Window (-zsh):** Shows the compilation and execution of the program. It runs `ghc -O Main.hs`, which compiles and links the modules. The output shows a warning about duplicate libraries and the execution of `./Main`, which prints the results of the functions: `2` and `[1,2]`.

MODULE SYSTEM: CYCLIC DEPENDENCIES

- ▶ Cyclic dependencies are not allowed



```
app — -zsh — 66x25
[Ekaterina.Verbitskaya@NVC00653 app % cat A.hs
module A where

import B

[Ekaterina.Verbitskaya@NVC00653 app % cat B.hs
module B where

import A

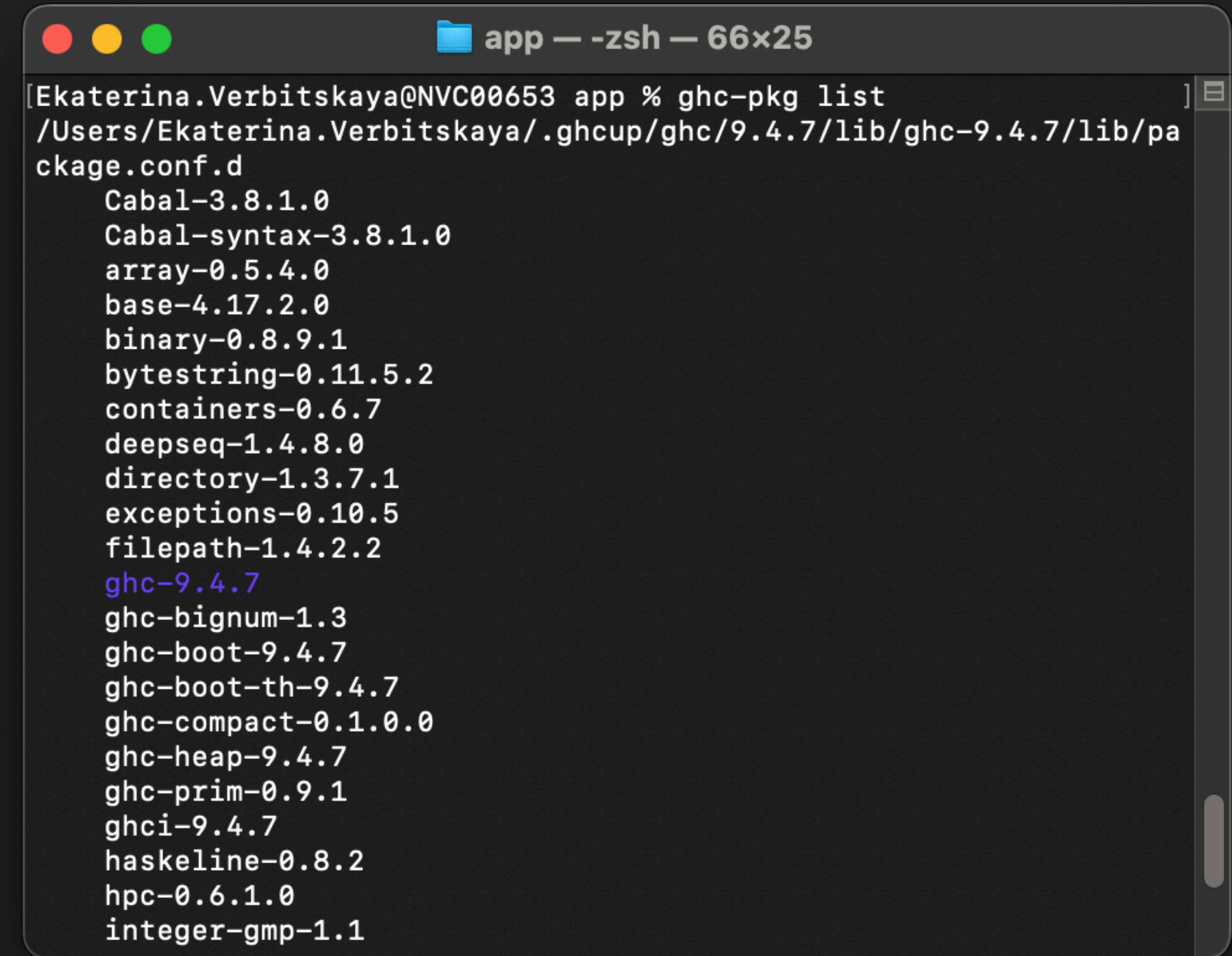
[Ekaterina.Verbitskaya@NVC00653 app % ghc B.hs
Module graph contains a cycle:
    module 'B' (B.hs)
      imports module 'A' (./A.hs)
    which imports module 'B' (B.hs)
[Ekaterina.Verbitskaya@NVC00653 app % ghc A.hs
Module graph contains a cycle:
    module 'B' (./B.hs)
      imports module 'A' (A.hs)
    which imports module 'B' (./B.hs)
Ekaterina.Verbitskaya@NVC00653 app %
```

EXERCISE

- ▶ Make three modules **A**, **B**, and **C**
- ▶ Add functions with names **f**, **g**, and **h** into each of the modules
- ▶ Make **C** depend on both **A** and **B**
- ▶ Make **B** depend on **A**
- ▶ Make **C** export only its function **h** and re-export the function **f** of module **A**

WHAT ARE THEY?

- ▶ Packages are collections of libraries
- ▶ Packages are units of distribution
- ▶ There might be some packages already installed on your system
- ▶ If the package you need is not installed, you can do it by **stack install** or **cabal install**



```
app — -zsh — 66x25
[Ekaterina.Verbitskaya@NVC00653 app % ghc-pkg list
/Users/Ekaterina.Verbitskaya/.ghcup/ghc/9.4.7/lib/ghc-9.4.7/lib/package.conf.d
  Cabal-3.8.1.0
  Cabal-syntax-3.8.1.0
  array-0.5.4.0
  base-4.17.2.0
  binary-0.8.9.1
  bytestring-0.11.5.2
  containers-0.6.7
  deepseq-1.4.8.0
  directory-1.3.7.1
  exceptions-0.10.5
  filepath-1.4.2.2
  ghc-9.4.7
  ghc-bignum-1.3
  ghc-boot-9.4.7
  ghc-boot-th-9.4.7
  ghc-compact-0.1.0.0
  ghc-heap-9.4.7
  ghc-prim-0.9.1
  ghci-9.4.7
  haskeline-0.8.2
  hpc-0.6.1.0
  integer-gmp-1.1
```


PACKAGE CONTAINERS

- ▶ A container is a data structure which holds some data, such as a dictionary or a tree
- ▶ [Data.Map](#), [Data.Set](#), [Data.Tree](#)...
- ▶ Many containers come in strict and lazy versions
 - ▶ Use the strict version if you need to access all of the values eventually
- ▶ [docs](#)

Modules

[\[Index\]](#) [\[Quick Jump\]](#)

Data

Containers

[Data.Containers.ListUtils](#)

[Data.Graph](#)

[Data.IntMap](#)

[Data.IntMap.Internal](#)

[Data.IntMap.Internal.Debug](#)

[Data.IntMap.Lazy](#)

Merge

[Data.IntMap.Merge.Lazy](#)

[Data.IntMap.Merge.Strict](#)

[Data.IntMap.Strict](#)

[Data.IntMap.Strict.Internal](#)

[Data.IntSet](#)

[Data.IntSet.Internal](#)

[Data.Map](#)

[Data.Map.Internal](#)

[Data.Map.Internal.Debug](#)

[Data.Map.Lazy](#)

Merge

[Data.Map.Merge.Lazy](#)

[Data.Map.Merge.Strict](#)

[Data.Map.Strict](#)

[Data.Map.Strict.Internal](#)

[Data.Sequence](#)

[Data.Sequence.Internal](#)

[Data.Sequence.Internal.Sorting](#)

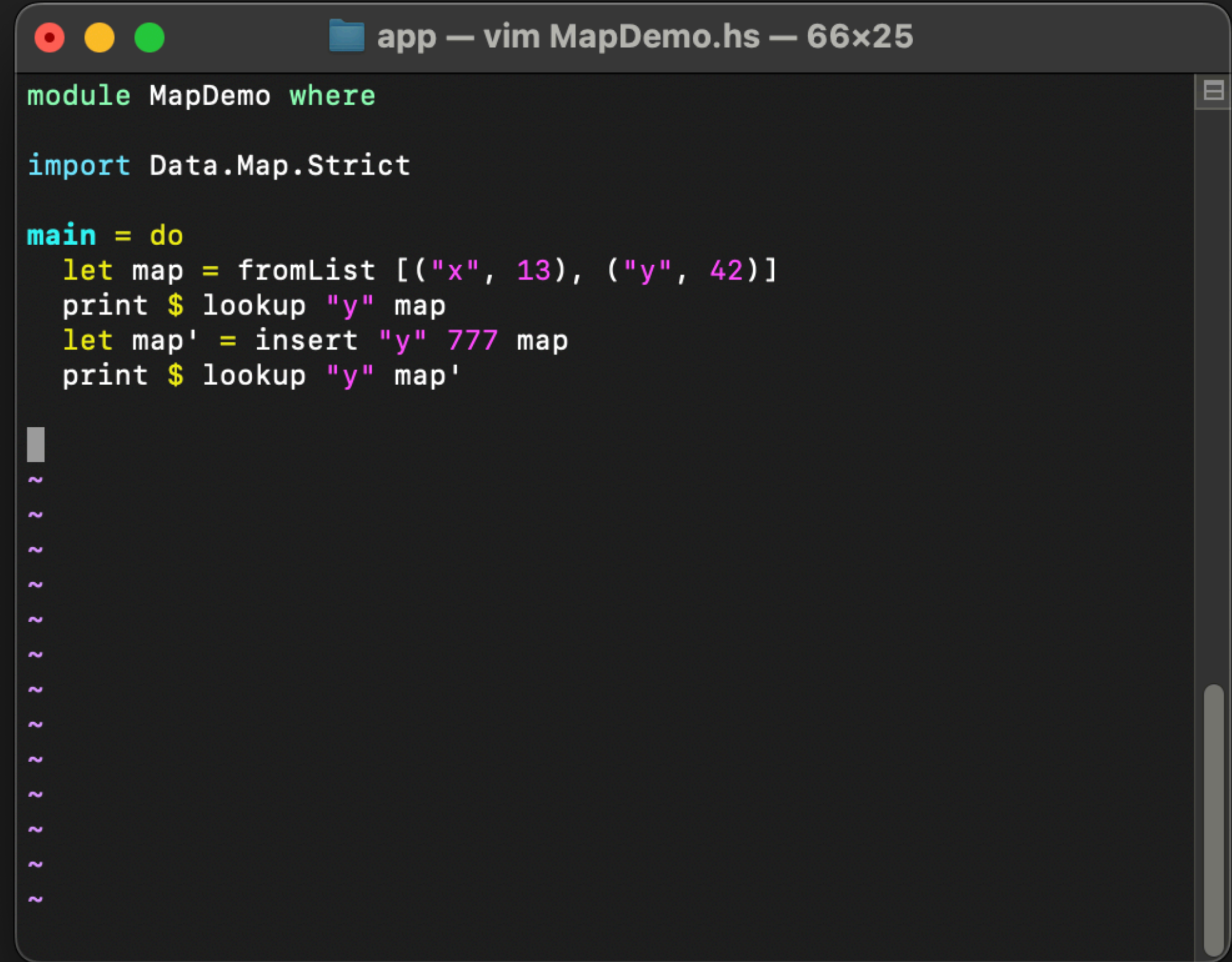
[Data.Set](#)

[Data.Set.Internal](#)

[Data.Tree](#)

CONTAINER MAP

- ▶ `Map k v` is a finite dictionary with keys of type `k` and values of type `v`



```
module MapDemo where

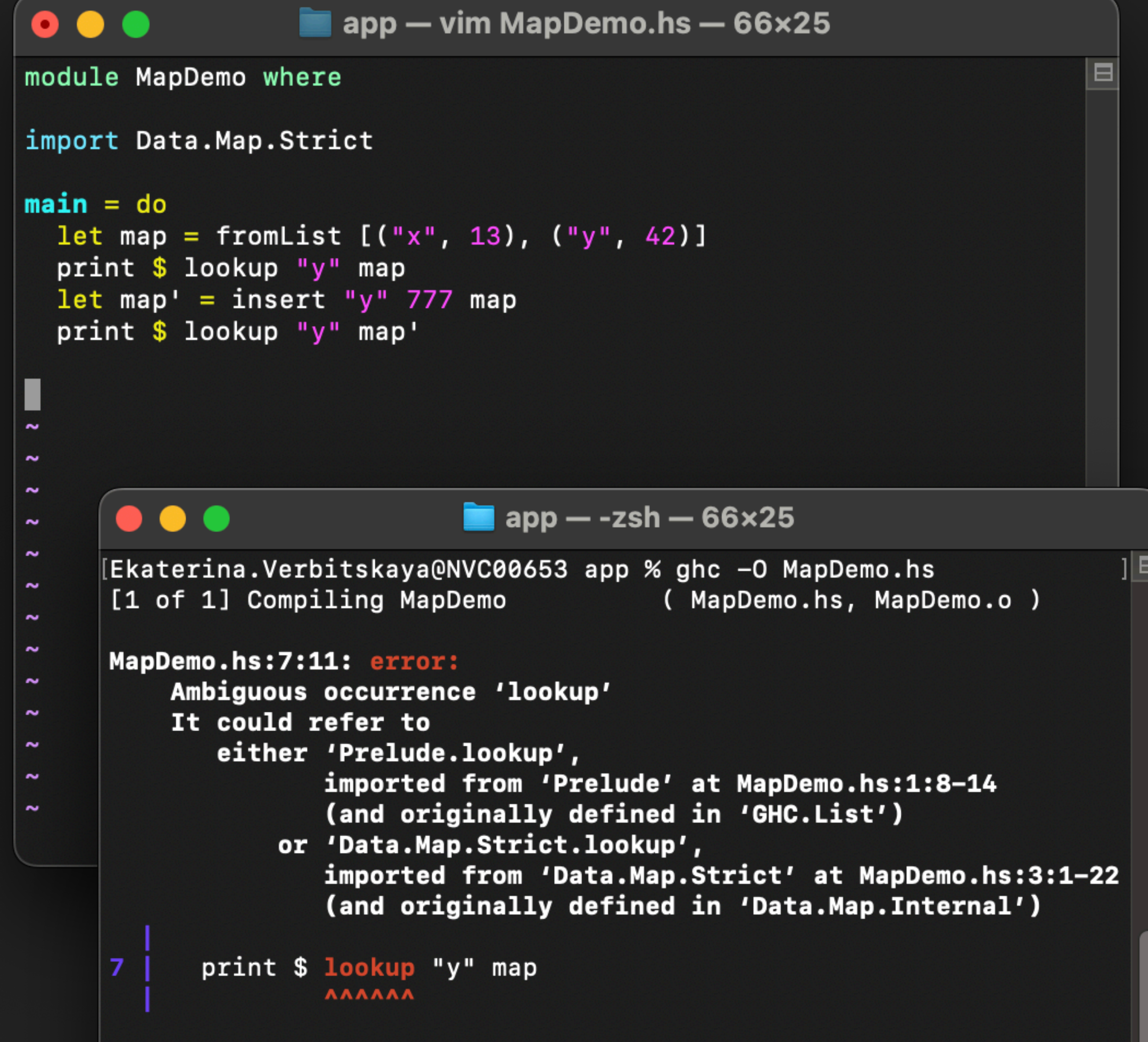
import Data.Map.Strict

main = do
  let map = fromList [("x", 13), ("y", 42)]
  print $ lookup "y" map
  let map' = insert "y" 777 map
  print $ lookup "y" map'
```

The screenshot shows a vim editor window titled "app — vim MapDemo.hs — 66x25". The code is written in Haskell and demonstrates the use of the `Map` container. It defines a module `MapDemo` and imports `Data.Map.Strict`. The `main` function is a `do` block that creates a map from a list of key-value pairs, looks up the value for key "y", inserts a new value for key "y", and looks up the value again. The code is color-coded: `module` is green, `where` is green, `import` is blue, `Data.Map.Strict` is blue, `main` is yellow, `= do` is yellow, `let` is yellow, `map` is yellow, `fromList` is yellow, `[("x", 13), ("y", 42)]` is pink, `print` is yellow, `$` is yellow, `lookup` is yellow, `"y"` is pink, `map` is yellow, `let map' = insert "y" 777 map` is yellow, `print` is yellow, `$` is yellow, `lookup` is yellow, `"y"` is pink, `map'` is yellow. There are also some purple squiggly lines at the bottom of the code block.

CONTAINER MAP

- ▶ `Map k v` is a finite dictionary with keys of type `k` and values of type `v`
- ▶ **Ambiguous occurrence** means that the same name comes from different modules



The screenshot shows a Haskell development environment. The top window is a code editor titled "app — vim MapDemo.hs — 66x25" containing the following code:

```
module MapDemo where

import Data.Map.Strict

main = do
  let map = fromList [("x", 13), ("y", 42)]
  print $ lookup "y" map
  let map' = insert "y" 777 map
  print $ lookup "y" map'
```

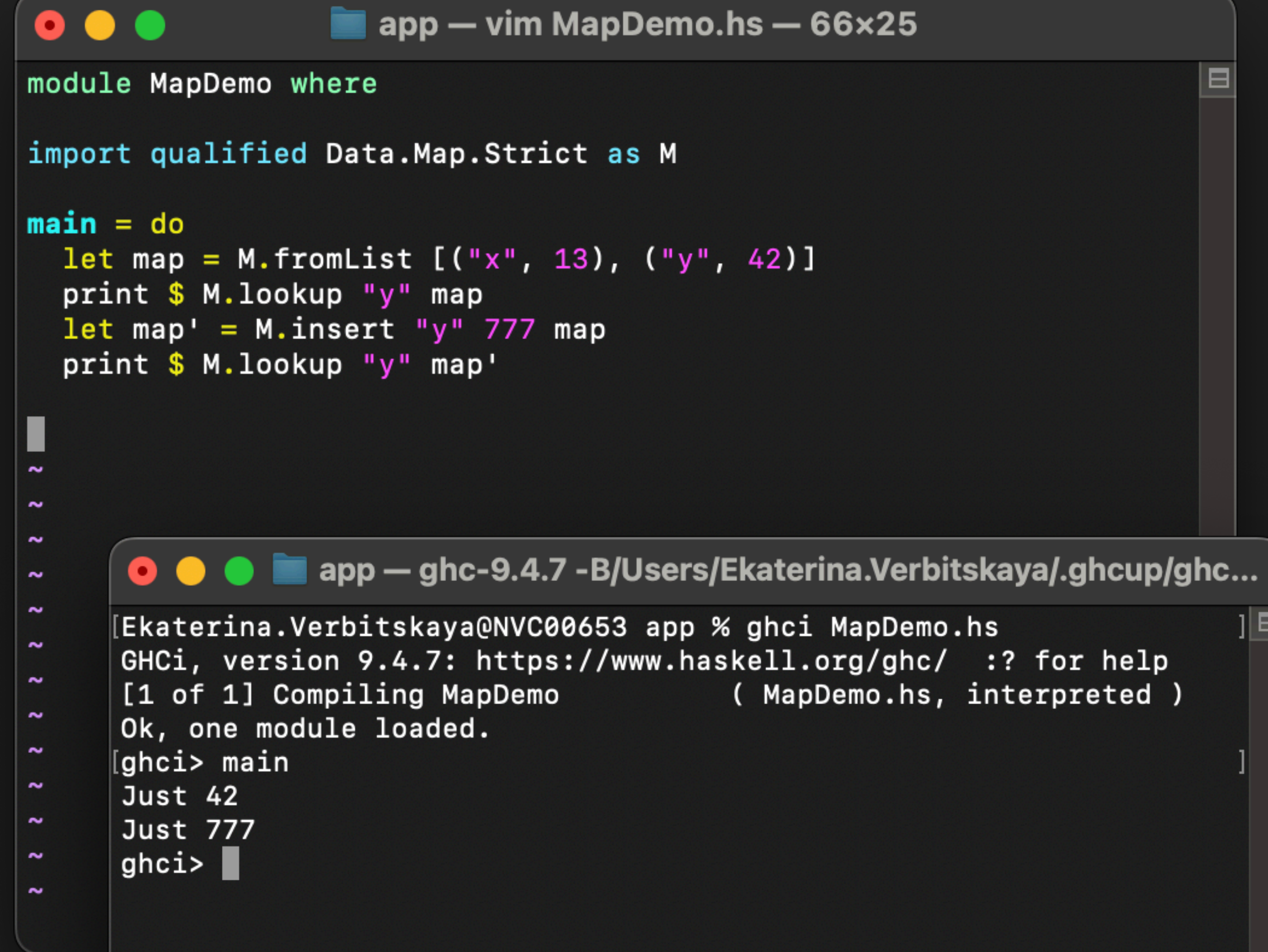
The bottom window is a terminal titled "app — -zsh — 66x25" showing the compilation of the code:

```
[Ekaterina.Verbitskaya@NVC00653 app % ghc -O MapDemo.hs
[1 of 1] Compiling MapDemo          ( MapDemo.hs, MapDemo.o )

MapDemo.hs:7:11: error:
  Ambiguous occurrence 'lookup'
  It could refer to
    either 'Prelude.lookup',
             imported from 'Prelude' at MapDemo.hs:1:8-14
             (and originally defined in 'GHC.List')
    or 'Data.Map.Strict.lookup',
         imported from 'Data.Map.Strict' at MapDemo.hs:3:1-22
         (and originally defined in 'Data.Map.Internal')
7 | print $ lookup "y" map
   ^^^^^^^
```


CONTAINER MAP

- ▶ `Map k v` is a finite dictionary with keys of type `k` and values of type `v`
- ▶ **Ambiguous occurrence** means that the same name comes from different modules
- ▶ Use qualified imports!



The screenshot shows a Haskell development environment. The top window is a code editor titled "app — vim MapDemo.hs — 66x25" containing the following code:

```
module MapDemo where

import qualified Data.Map.Strict as M

main = do
  let map = M.fromList [("x", 13), ("y", 42)]
  print $ M.lookup "y" map
  let map' = M.insert "y" 777 map
  print $ M.lookup "y" map'
```

The bottom window is a terminal titled "app — ghc-9.4.7 -B/Users/Ekaterina.Verbitskaya/.ghcup/ghc..." showing the execution of the code:

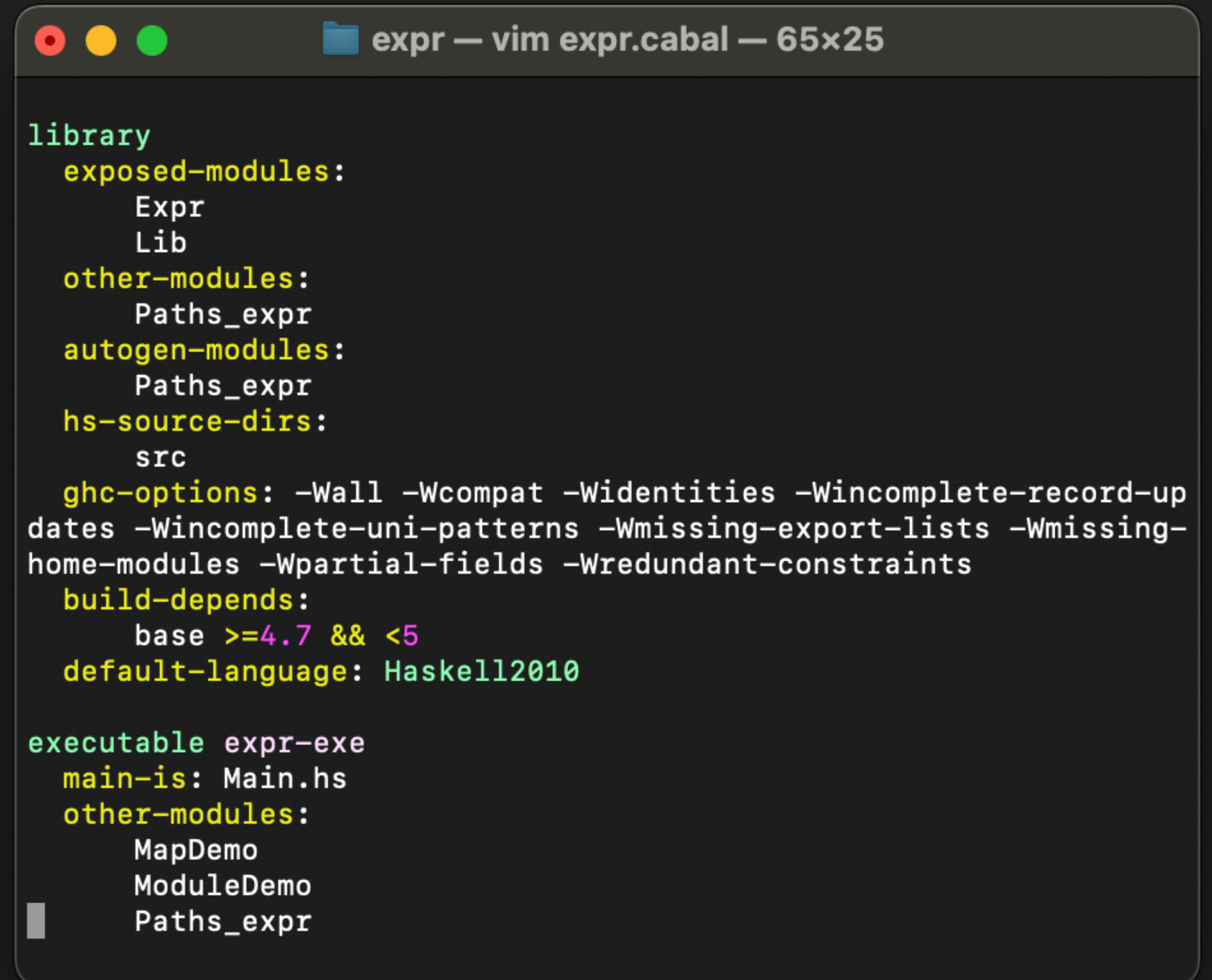
```
[Ekaterina.Verbitskaya@NVC00653 app % ghci MapDemo.hs
GHCi, version 9.4.7: https://www.haskell.org/ghc/  :? for help
[1 of 1] Compiling MapDemo          ( MapDemo.hs, interpreted )
Ok, one module loaded.
[ghci> main
Just 42
Just 777
ghci> ]
```


EXERCISE

- ▶ Replace associative list with a dictionary from `Data.Map` in `eval` in HW04

CABAL

- ▶ Build system for Haskell project
- ▶ Resolves dependencies specified in `*.cabal` file
- ▶ Builds libs, executables, and test suits
- ▶ May need some help with resolving dependencies



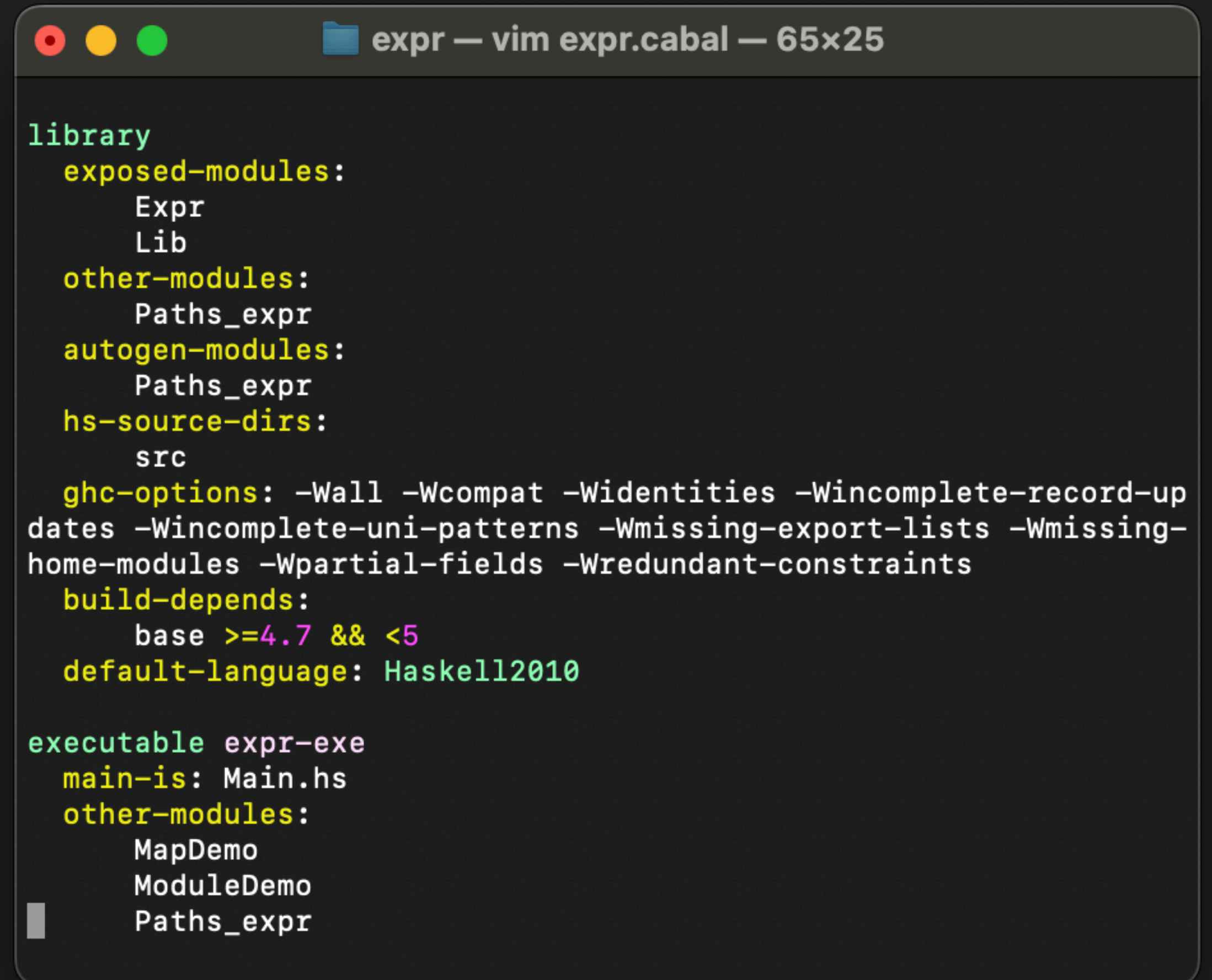
The screenshot shows a vim editor window titled "expr — vim expr.cabal — 65x25". The window displays the following Cabal configuration:

```
library
  exposed-modules:
    Expr
    Lib
  other-modules:
    Paths_expr
  autogen-modules:
    Paths_expr
  hs-source-dirs:
    src
  ghc-options: -Wall -Wcompat -Widentities -Wincomplete-record-updates
               -Wincomplete-uni-patterns -Wmissing-export-lists -Wmissing-home-modules
               -Wpartial-fields -Wredundant-constraints
  build-depends:
    base >=4.7 && <5
  default-language: Haskell2010

executable expr-exe
  main-is: Main.hs
  other-modules:
    MapDemo
    ModuleDemo
    Paths_expr
```

STACK

- ▶ Does what cabal does, but also:
 - ▶ Sandboxes everything, including ghc
 - ▶ Guarantees no conflict between dependencies when you use Stackage Its
- ▶ I recommend using stack
- ▶ Livecoding



```
expr — vim expr.cabal — 65x25

library
  exposed-modules:
    Expr
    Lib
  other-modules:
    Paths_expr
  autogen-modules:
    Paths_expr
  hs-source-dirs:
    src
  ghc-options: -Wall -Wcompat -Widentities -Wincomplete-record-up
dates -Wincomplete-uni-patterns -Wmissing-export-lists -Wmissing-
home-modules -Wpartial-fields -Wredundant-constraints
  build-depends:
    base >=4.7 && <5
  default-language: Haskell2010

executable expr-exe
  main-is: Main.hs
  other-modules:
    MapDemo
    ModuleDemo
    Paths_expr
```


EXERCISE

- ▶ Create a stack project
- ▶ Copy your code for Expressions there
- ▶ Make sure it builds and executes

WHAT IS A UNIT TEST

- ▶ A test which test 1 unit of functionality
- ▶ We usually test functions
 - ▶ Assert that an value computed by the function is equal to the expected
 - ▶ Assert that some predicate holds (e.g. `isJust`)
- ▶ Livecoding

```
app — vim TestDemo.hs — 65x25
module TestDemo where

import ModuleDemo ( numUniques )

test msg act exp =
  if act /= exp
  then do
    putStrLn "Error!"
    putStrLn msg
  else
    return ()

main = do
  test "numUniques [] == 0" (numUniques @Int []) 0
  test "numUniques [1,1,1] == 1" (numUniques [1,1,1]) 1
  test "numUniques [1,2,3] == 3" (numUniques [1,2,3]) 3
  test "numUniques [1,2,1] == 1" (numUniques [1,2,1]) 2

~
~
~
~
~
```

EXERCISE

- ▶ Move your tests for Expr into a test project
- ▶ Make sure they run