INTRODUCTION TO

FUNCTIONAL PROGRAMMING

OUTLINE

- Modules
- Packages
- Creating your own package
- Unit testing

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

```haskell
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

```haskell
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

```haskell
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

```haskell
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

```haskell
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

```haskell
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

```haskell
module ModuleDemo ( numUniques
                  , nub ) 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

```haskell
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 = ...
```

MODULE SYSTEM: EXPORT

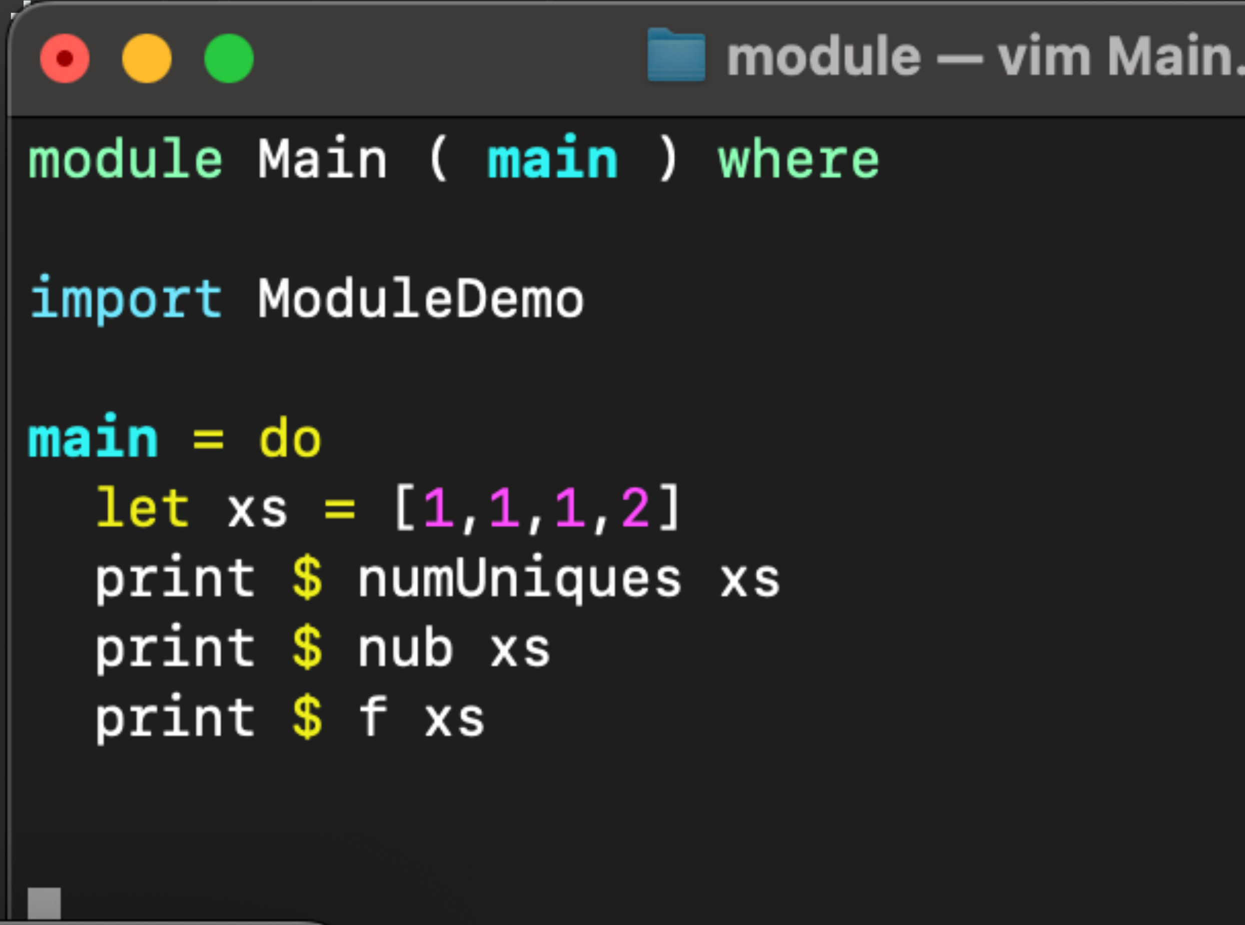
- Everything on the top level of the module is exported by default
- You can add names in parentheses to export only them
- You can re-export names from the imported modules
    - By default, only names defined in the module are exported

```haskell
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
```

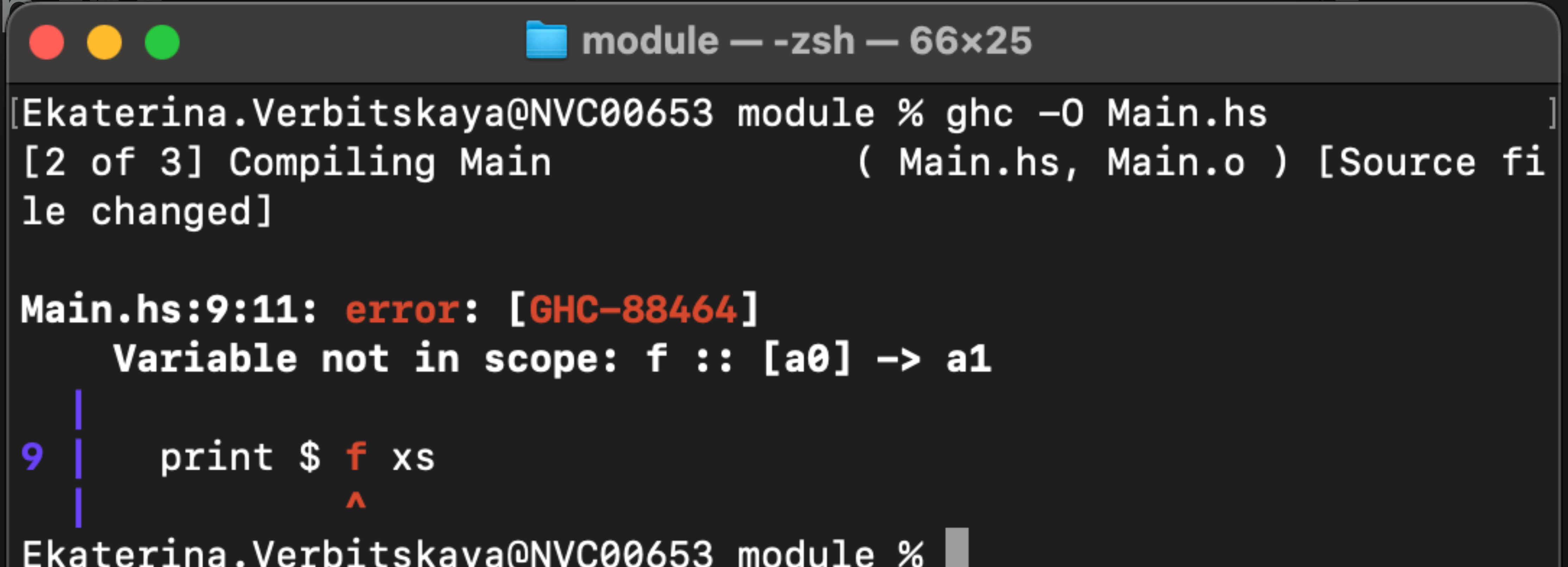
MODULE SYSTEM: EXPORT

- 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

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

import qualified Da

numUniques :: Eq a
numUniques = length

nub :: Ord a ⇒ [a]
nub = undefined

f = undefined
```



```haskell
module Main ( main ) where

import ModuleDemo

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

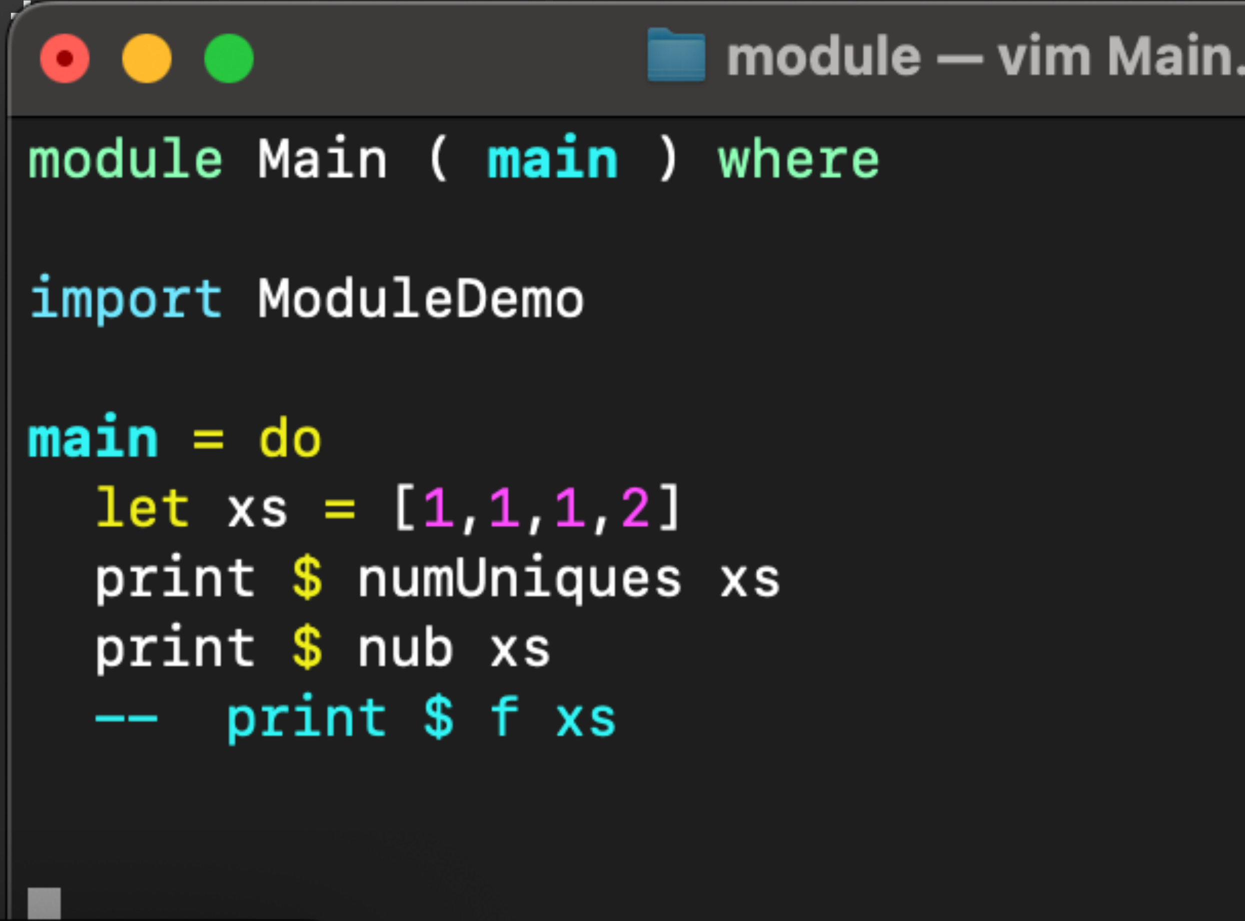


```
Ekaterina.Verbitskaya@NVC00653 module % ghc -O Main.hs
[2 of 3] Compiling Main             ( Main.hs, Main.o ) [Source fi
le changed]

Main.hs:9:11: error: [GHC-88464]
    Variable not in scope: f :: [a0] -> a1
  |
9 |   print $ f xs
  |           ^
Ekaterina.Verbitskaya@NVC00653 module %
```

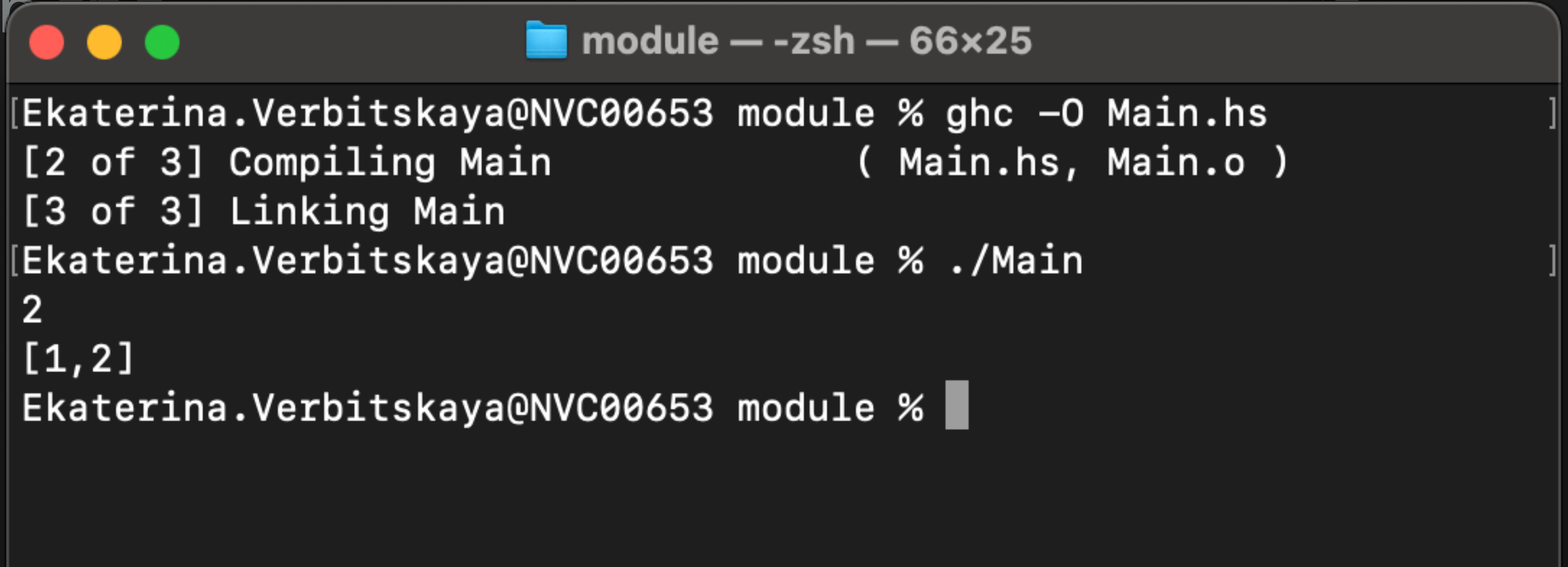
MODULE SYSTEM: EXPORT

- 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

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

import qualified Da

numUniques :: Eq a
numUniques = length

nub :: Ord a ⇒ [a]
nub = undefined

f = undefined
```



```
module Main ( main ) where

import ModuleDemo

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

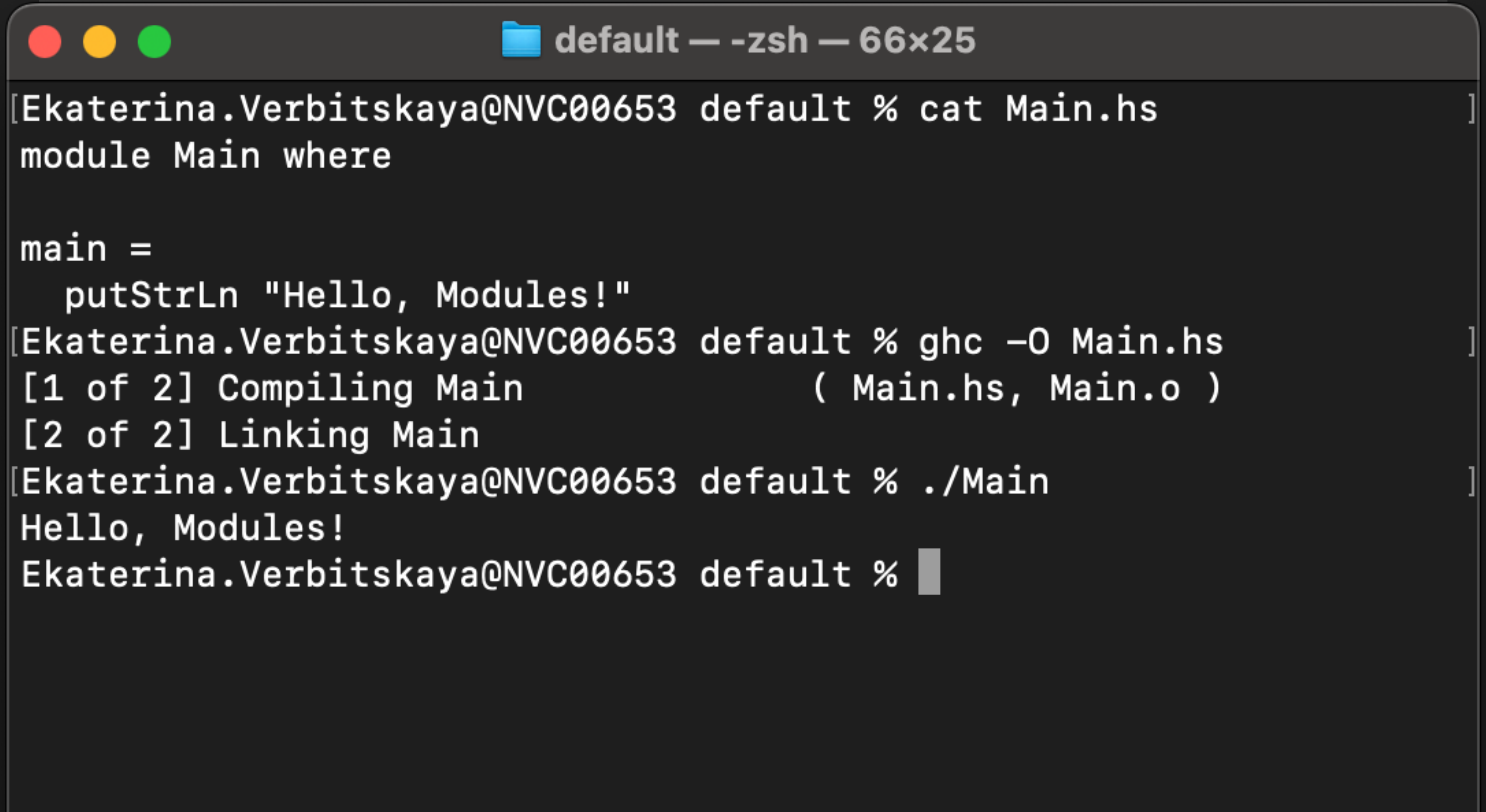


```
Ekaterina.Verbitskaya@NVC00653 module % ghc -O Main.hs
[2 of 3] Compiling Main             ( Main.hs, Main.o )
[3 of 3] Linking Main
Ekaterina.Verbitskaya@NVC00653 module % ./Main
2
[1,2]
Ekaterina.Verbitskaya@NVC00653 module %
```

DEFAULT MODULE NAME

- If you don’t specify the name of the module it is presumed to be `Main`
- By default, you can’t create an executable from the module with another name
  - use `-main-is` flag of `ghc` to circumvent this restriction

```
module Main where

main =
  putStrLn “Hello, Modules!"
```

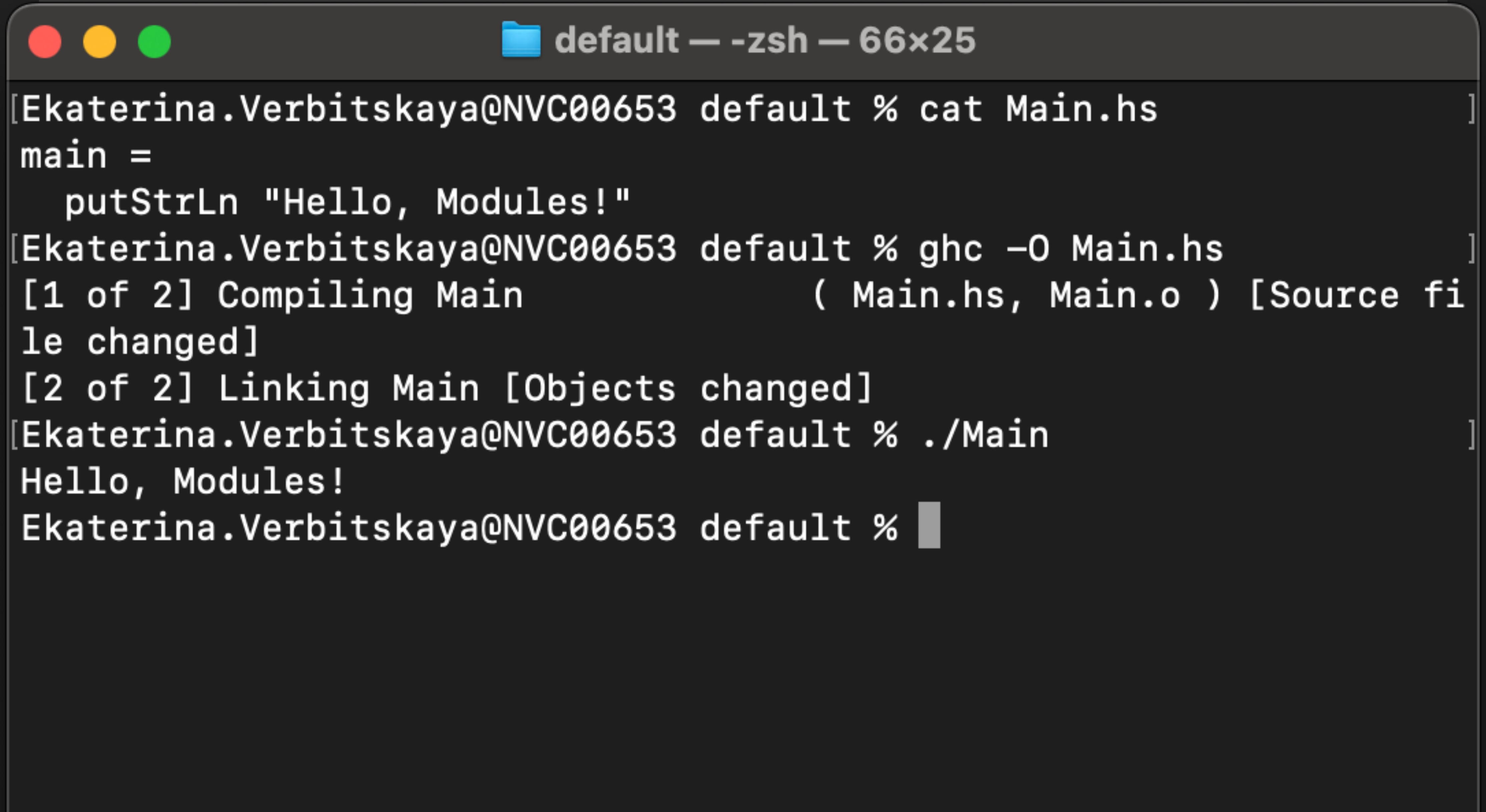


```
[Ekaterina.Verbitskaya@NVC00653 default % cat Main.hs              ]
module Main where

main =
  putStrLn "Hello, Modules!"
[Ekaterina.Verbitskaya@NVC00653 default % ghc -O Main.hs           ]
[1 of 2] Compiling Main             ( Main.hs, Main.o )
[2 of 2] Linking Main
[Ekaterina.Verbitskaya@NVC00653 default % ./Main                   ]
Hello, Modules!
Ekaterina.Verbitskaya@NVC00653 default % 
```

DEFAULT MODULE NAME

- If you don't specify the name of the module it is presumed to be `Main`
- By default, you can't create an executable from the module with another name
  - use `-main-is` flag of `ghc` to circumvent this restriction

```
main =
  putStrLn “Hello, Modules!"
```

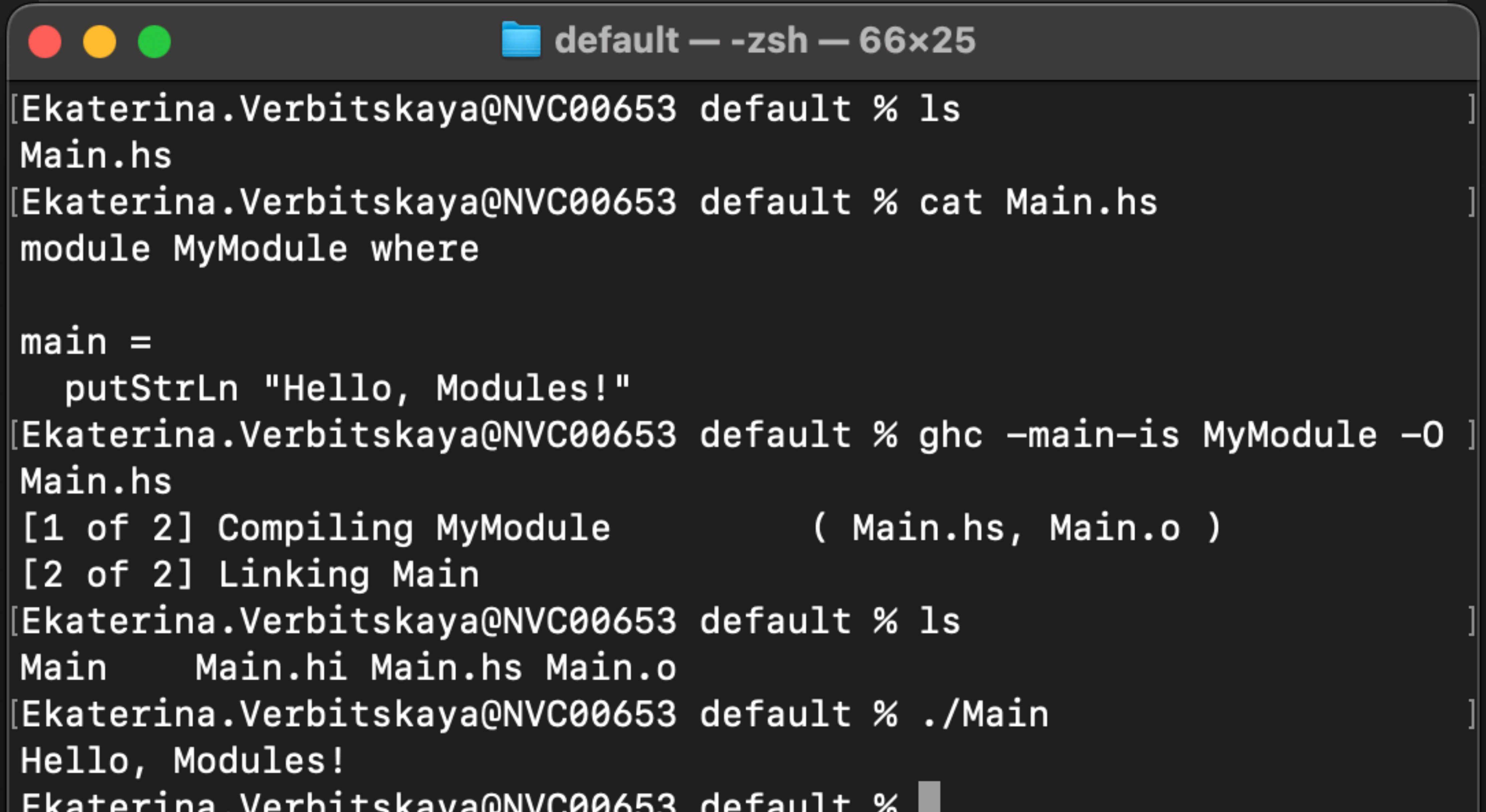


```
[Ekaterina.Verbitskaya@NVC00653 default % cat Main.hs                  ]
main =
  putStrLn "Hello, Modules!"
[Ekaterina.Verbitskaya@NVC00653 default % ghc -O Main.hs               ]
[1 of 2] Compiling Main             ( Main.hs, Main.o ) [Source fi
le changed]
[2 of 2] Linking Main [Objects changed]
[Ekaterina.Verbitskaya@NVC00653 default % ./Main                       ]
Hello, Modules!
Ekaterina.Verbitskaya@NVC00653 default % █
```

DEFAULT MODULE NAME

- If you don’t specify the name of the module it is presumed to be `Main`
- By default, you can’t create an executable from the module with another name
  - use `-main-is` flag of `ghc` to circumvent this restriction

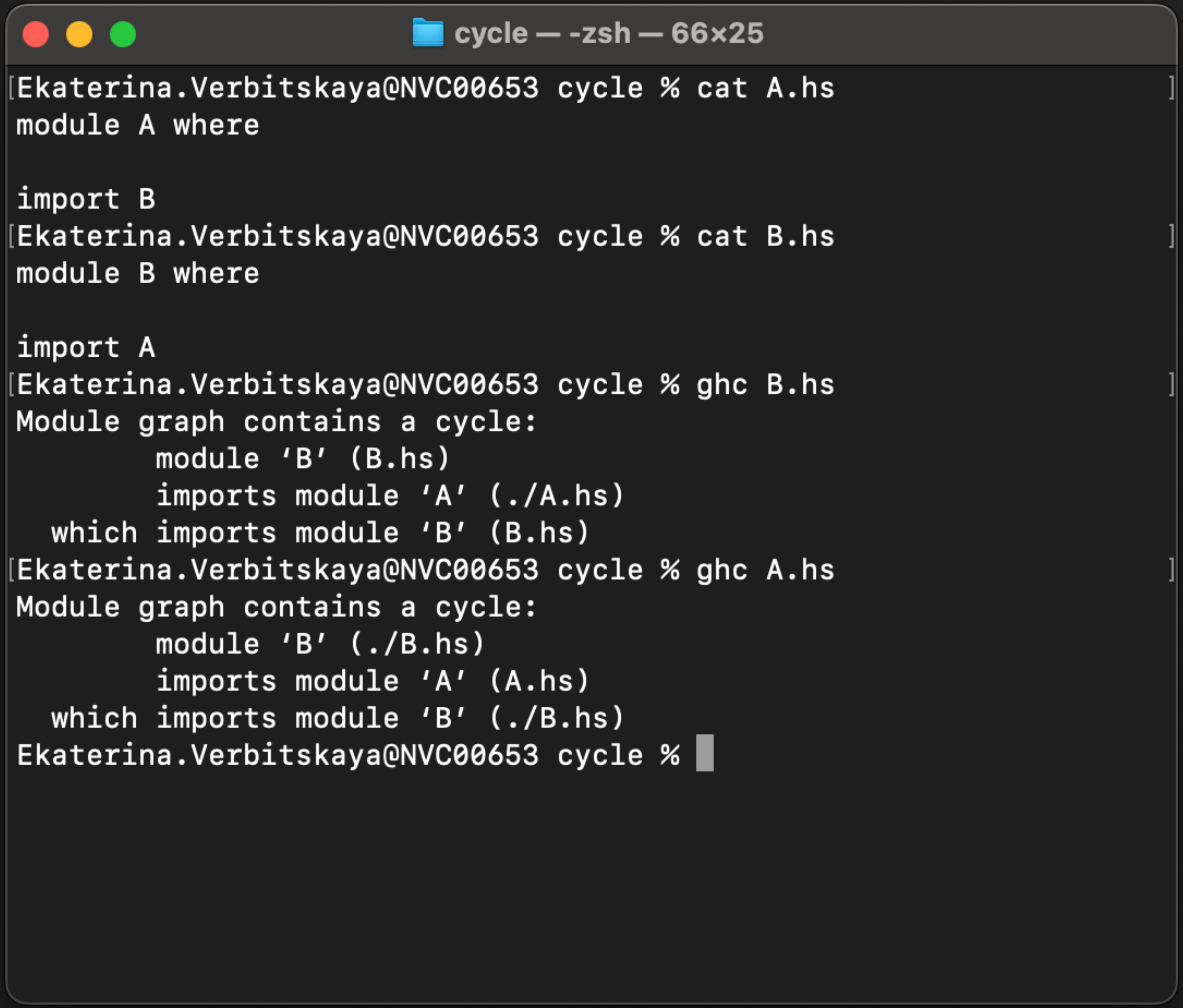
```haskell
module MyModule where

main =
  putStrLn “Hello, Modules!"
```



```
[Ekaterina.Verbitskaya@NVC00653 default % ls
Main.hs
[Ekaterina.Verbitskaya@NVC00653 default % cat Main.hs
module MyModule where

main =
  putStrLn "Hello, Modules!"
[Ekaterina.Verbitskaya@NVC00653 default % ghc -main-is MyModule -O
Main.hs
[1 of 2] Compiling MyModule         ( Main.hs, Main.o )
[2 of 2] Linking Main
[Ekaterina.Verbitskaya@NVC00653 default % ls
Main    Main.hi Main.hs Main.o
[Ekaterina.Verbitskaya@NVC00653 default % ./Main
Hello, Modules!
Ekaterina.Verbitskaya@NVC00653 default %
```

MODULE SYSTEM: CYCLIC DEPENDENCIES

- Cyclic dependencies are not allowed



```
Ekaterina.Verbitskaya@NVC00653 cycle % cat A.hs
module A where

import B
Ekaterina.Verbitskaya@NVC00653 cycle % cat B.hs
module B where

import A
Ekaterina.Verbitskaya@NVC00653 cycle % 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 cycle % 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 cycle %
```

EXERCISE

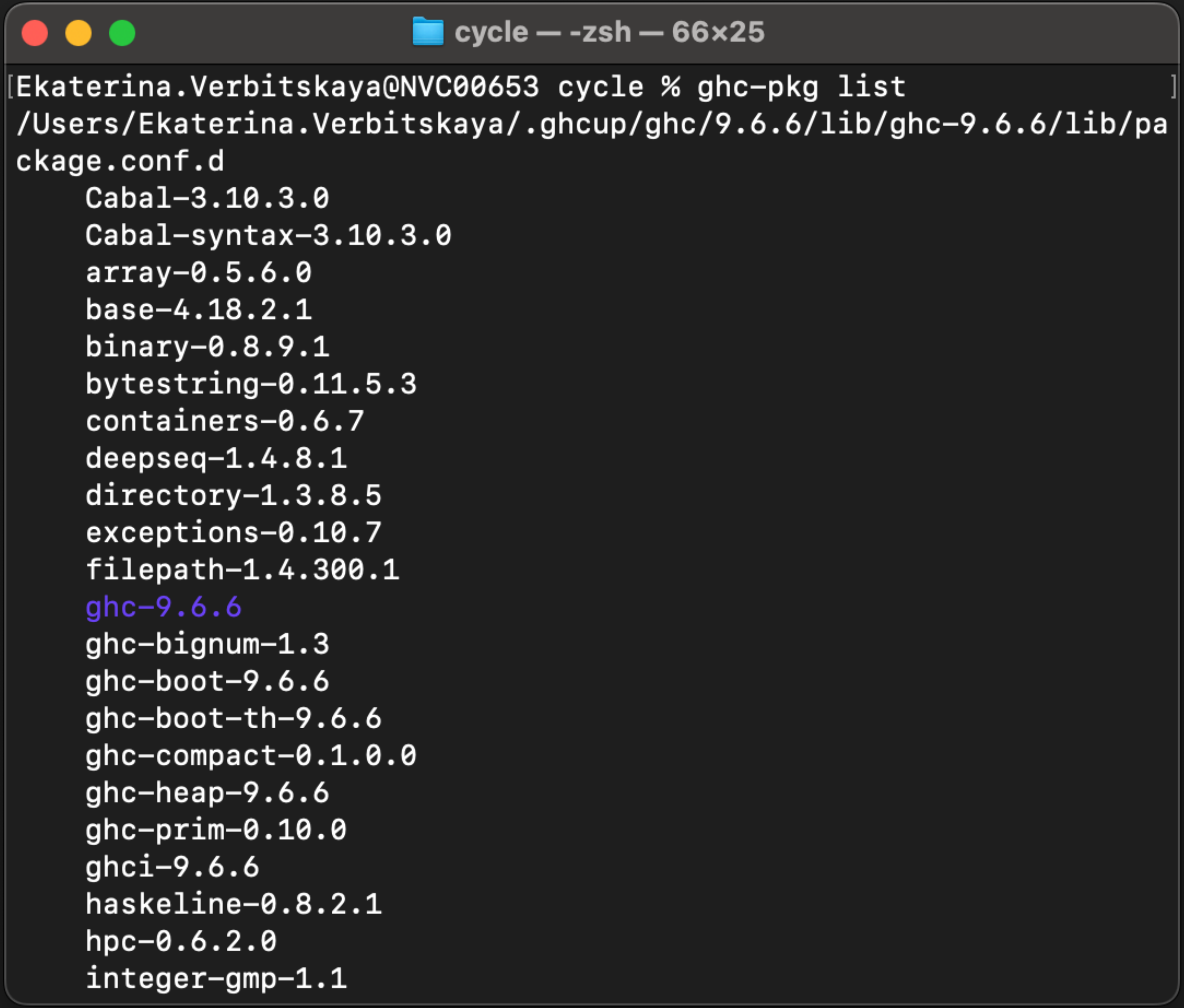
- Make three modules A, B, and C
- Add functions with names f, g, and h into each of the modules
- Make C depend on both A and B
- Make B depend on A
- Make C export only its function h and re-export the function f of module A

OUTLINE

- Modules
- Packages
- Creating your own package
- Unit testing

WHAT ARE THEY?

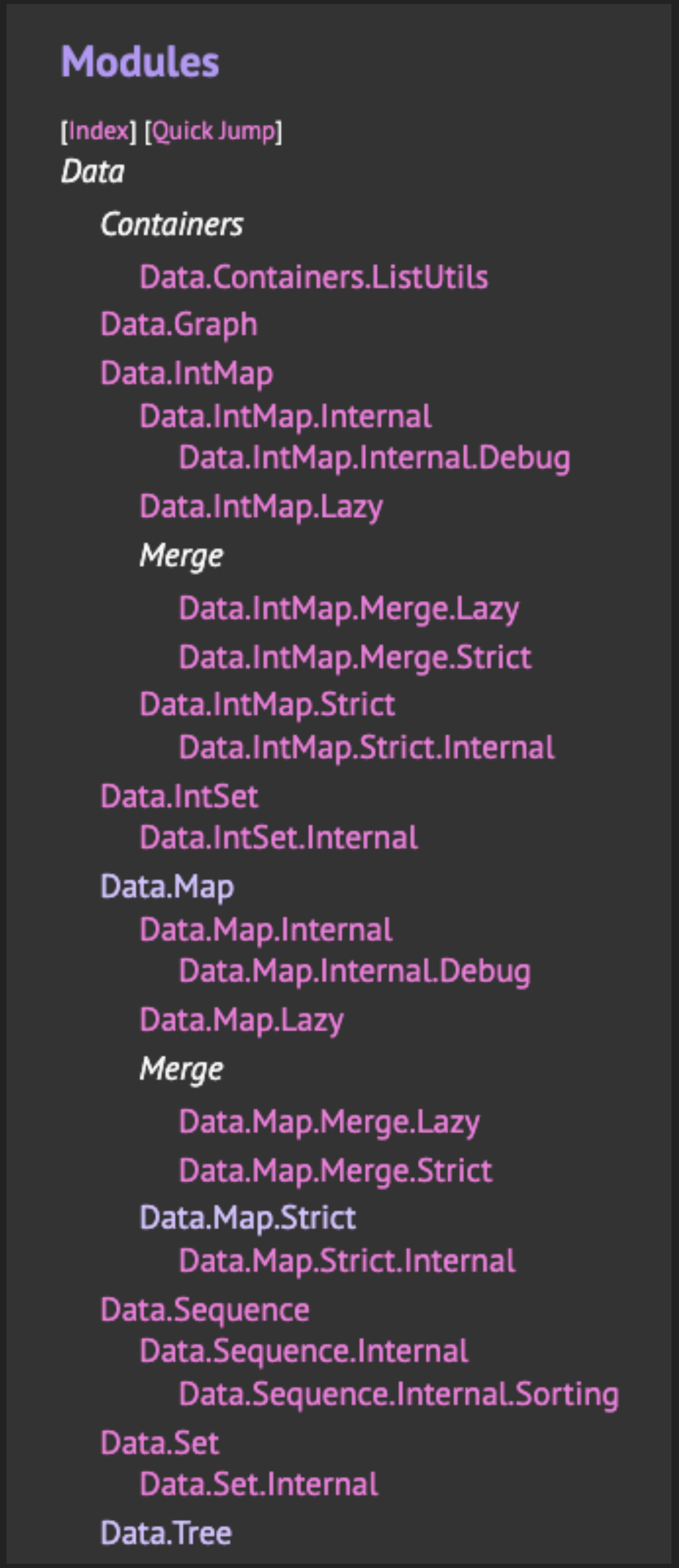
- Packages are collections of libraries
- Packages are units of distribution
- There might be some packages already installed on your system
- If the package you need is not installed, you can do it by stack install or cabal install



```
Ekaterina.Verbitskaya@NVC00653 cycle % ghc-pkg list
/Users/Ekaterina.Verbitskaya/.ghcup/ghc/9.6.6/lib/ghc-9.6.6/lib/pa
ckage.conf.d
    Cabal-3.10.3.0
    Cabal-syntax-3.10.3.0
    array-0.5.6.0
    base-4.18.2.1
    binary-0.8.9.1
    bytestring-0.11.5.3
    containers-0.6.7
    deepseq-1.4.8.1
    directory-1.3.8.5
    exceptions-0.10.7
    filepath-1.4.300.1
    ghc-9.6.6
    ghc-bignum-1.3
    ghc-boot-9.6.6
    ghc-boot-th-9.6.6
    ghc-compact-0.1.0.0
    ghc-heap-9.6.6
    ghc-prim-0.10.0
    ghci-9.6.6
    haskeline-0.8.2.1
    hpc-0.6.2.0
    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

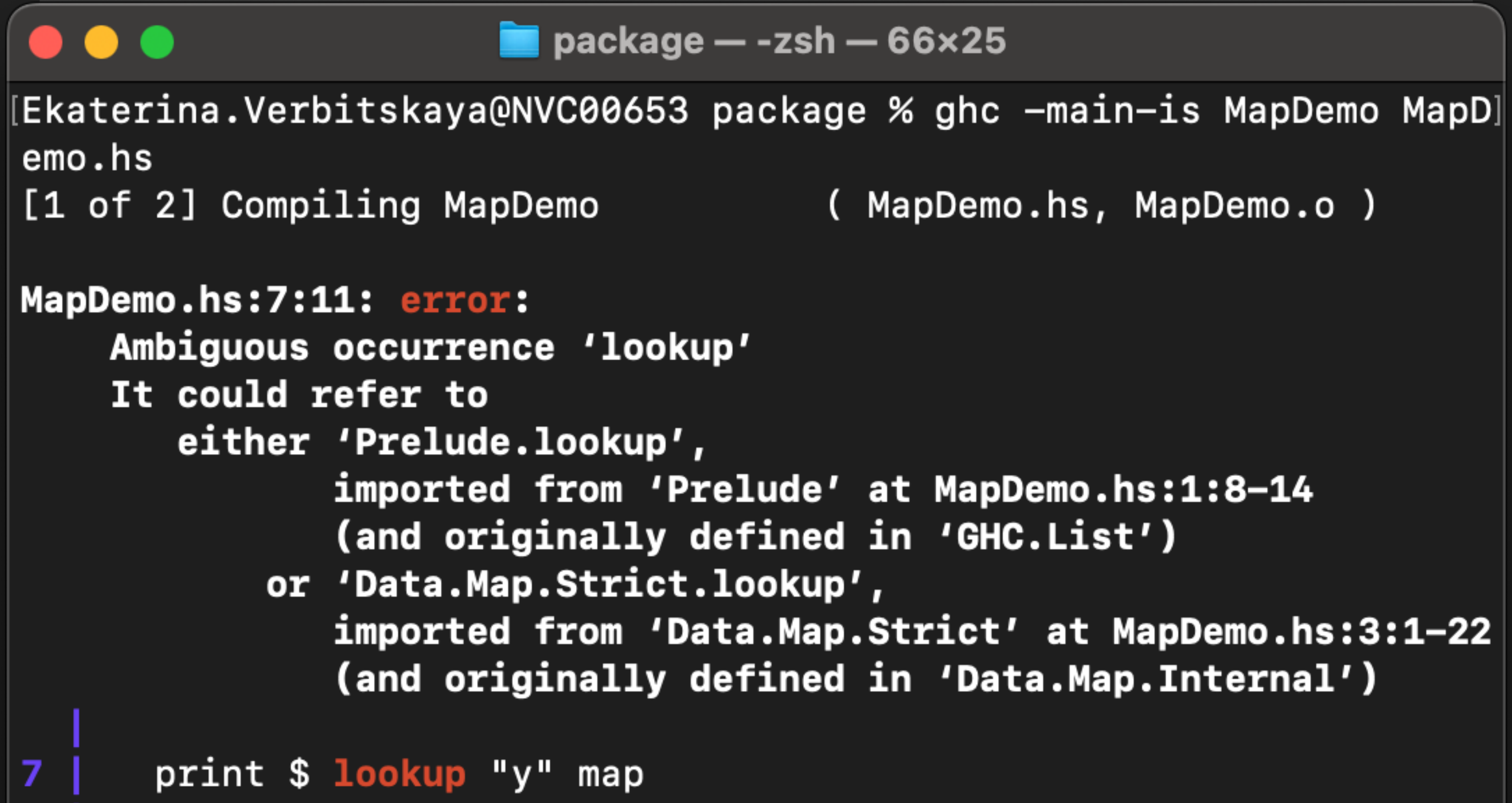
- Map k v is a finite dictionary with keys of type k and values of type v

```haskell
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'
```

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

```haskell
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'
```

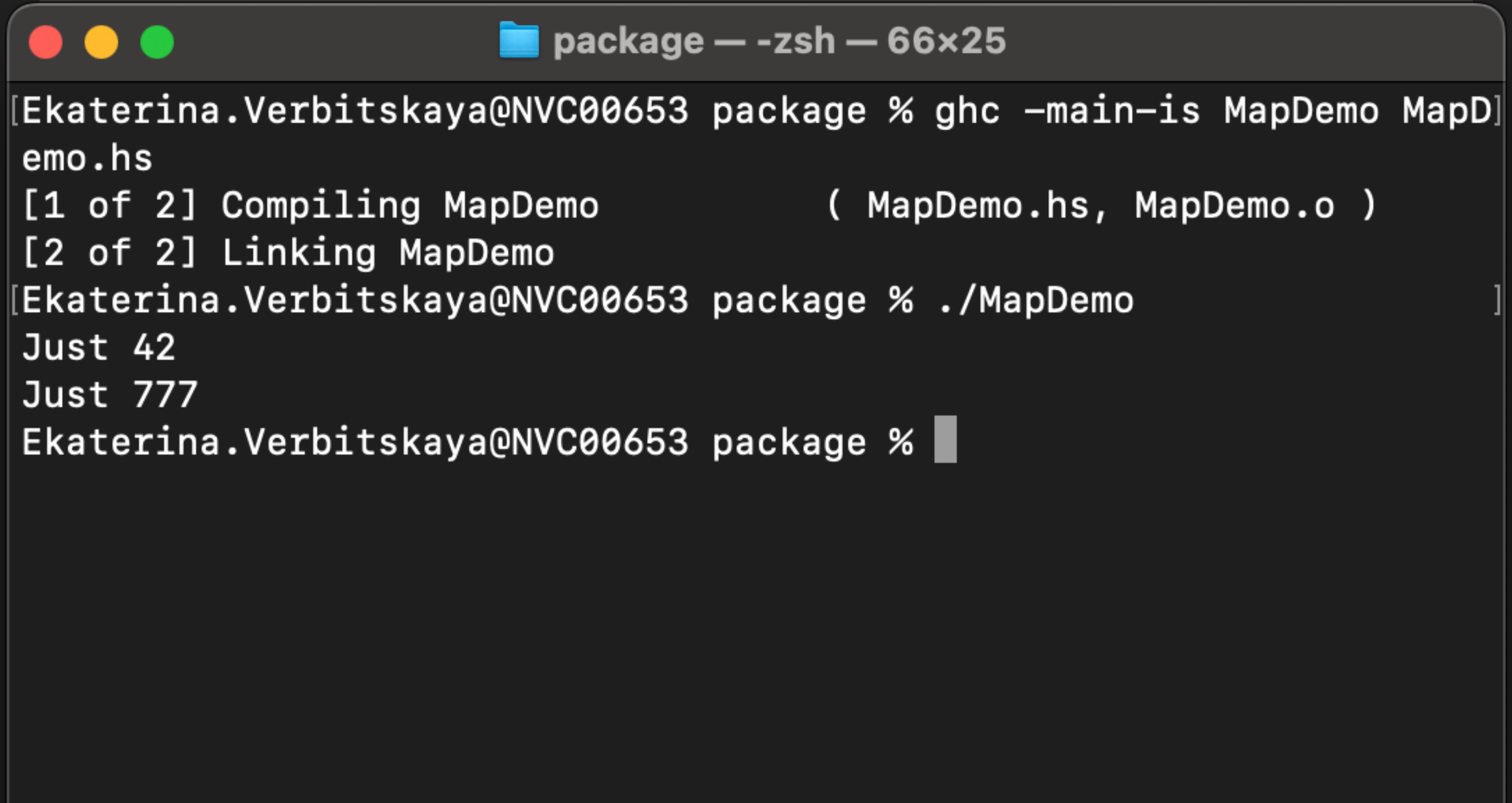


```
[Ekaterina.Verbitskaya@NVC00653 package % ghc -main-is MapDemo MapD]
emo.hs
[1 of 2] Compiling MapDemo          ( MapDemo.hs, MapDemo.o )

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!

```haskell
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'
```

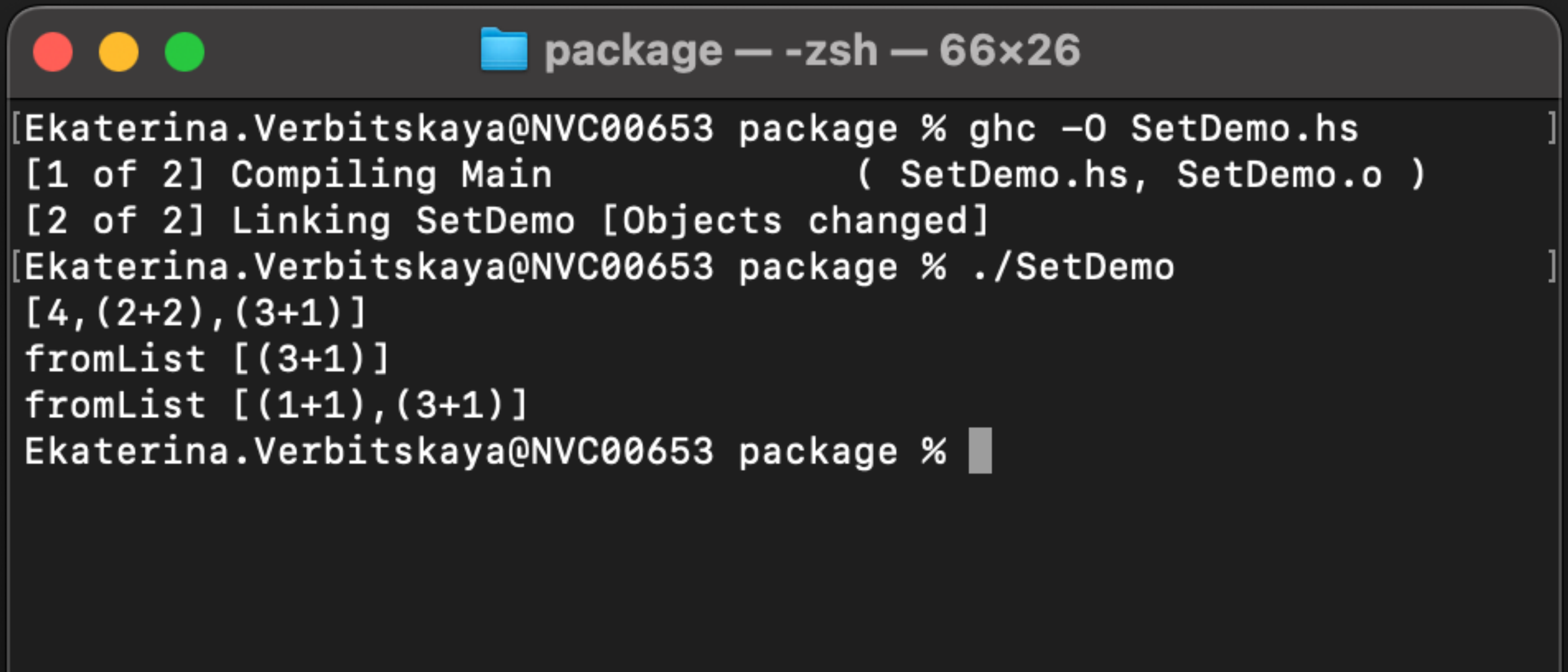


package — -zsh — 66×25

```
[Ekaterina.Verbitskaya@NVC00653 package % ghc -main-is MapDemo MapD]
emo.hs
[1 of 2] Compiling MapDemo          ( MapDemo.hs, MapDemo.o )
[2 of 2] Linking MapDemo
[Ekaterina.Verbitskaya@NVC00653 package % ./MapDemo                ]
Just 42
Just 777
Ekaterina.Verbitskaya@NVC00653 package % 
```

CONTAINER SET

- `Set e` represents a set of elements of type `e`
  - Based on size balanced trees
  - `union, difference, intersection:` $O(m * log(\frac{n}{m} + 1)), 0 < m \le n$ complexity



```
[Ekaterina.Verbitskaya@NVC00653 package % ghc -O SetDemo.hs          ]
[1 of 2] Compiling Main             ( SetDemo.hs, SetDemo.o )
[2 of 2] Linking SetDemo [Objects changed]
[Ekaterina.Verbitskaya@NVC00653 package % ./SetDemo                   ]
[4,(2+2),(3+1)]
fromList [(3+1)]
fromList [(1+1),(3+1)]
Ekaterina.Verbitskaya@NVC00653 package %
```

```haskell
import qualified Data.Set as S

data Expr = Lit Int | Plus Expr Expr

instance Show Expr where
  show (Lit n) = show n
  show (Plus x y) = '(' : show x ++ '+' : show y ++ ")"

eval (Lit n) = n
eval (Plus x y) = eval x + eval y

instance Ord Expr where
  x <= y = eval x <= eval y

instance Eq Expr where
  x == y = eval x == eval y

main = do
  let exprs = [Lit 4, Plus (Lit 2) (Lit 2), Plus (Lit 3) (Lit 1)]
  let set = S.fromList exprs
  let set' = S.insert (Plus (Lit 1) (Lit 1)) set
  print exprs
  print set
  print set'
```

EXERCISE

- Introduce variables into the expression data type, and rewrite eval to fetch the value of the var from a map from `Data.Map`
- What can go wrong here?

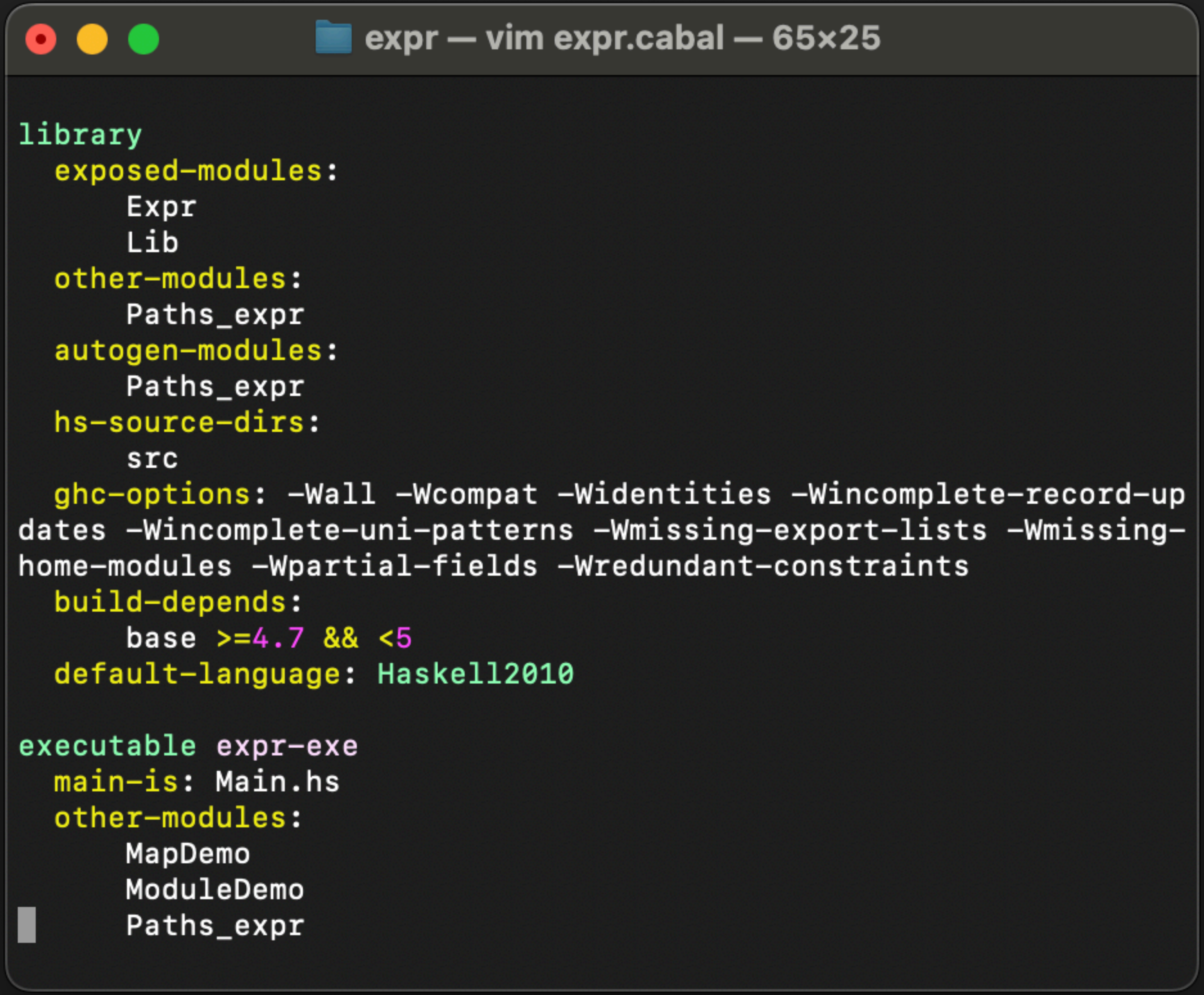
```haskell
import qualified Data.Map.Strict as M

data Expr = Lit Int | Plus Expr Expr | Var String

eval :: M.Map String Expr -> Expr -> Int
eval _ (Lit n) = n
eval _ (Plus x y) = eval x + eval y
eval state (Var v) = undefined
```

OUTLINE

- Modules
- Packages
- Creating your own package
- Unit testing

CABAL

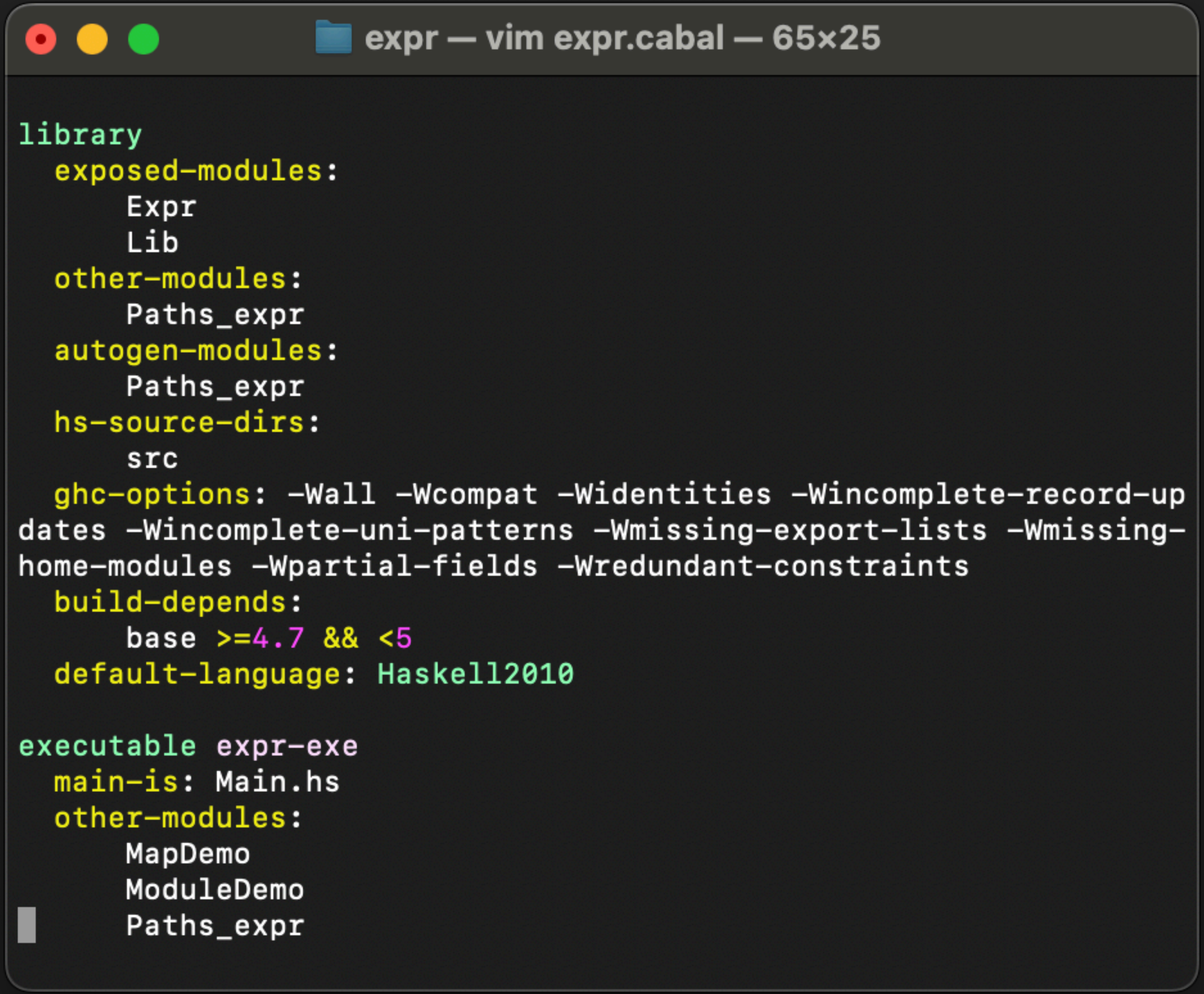
- Build system for Haskell project
- Resolves dependencies specified in a `*.cabal` file
- Builds libs, executables, and test suits
- May need some help with resolving dependencies



```
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
```

STACK

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



```
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

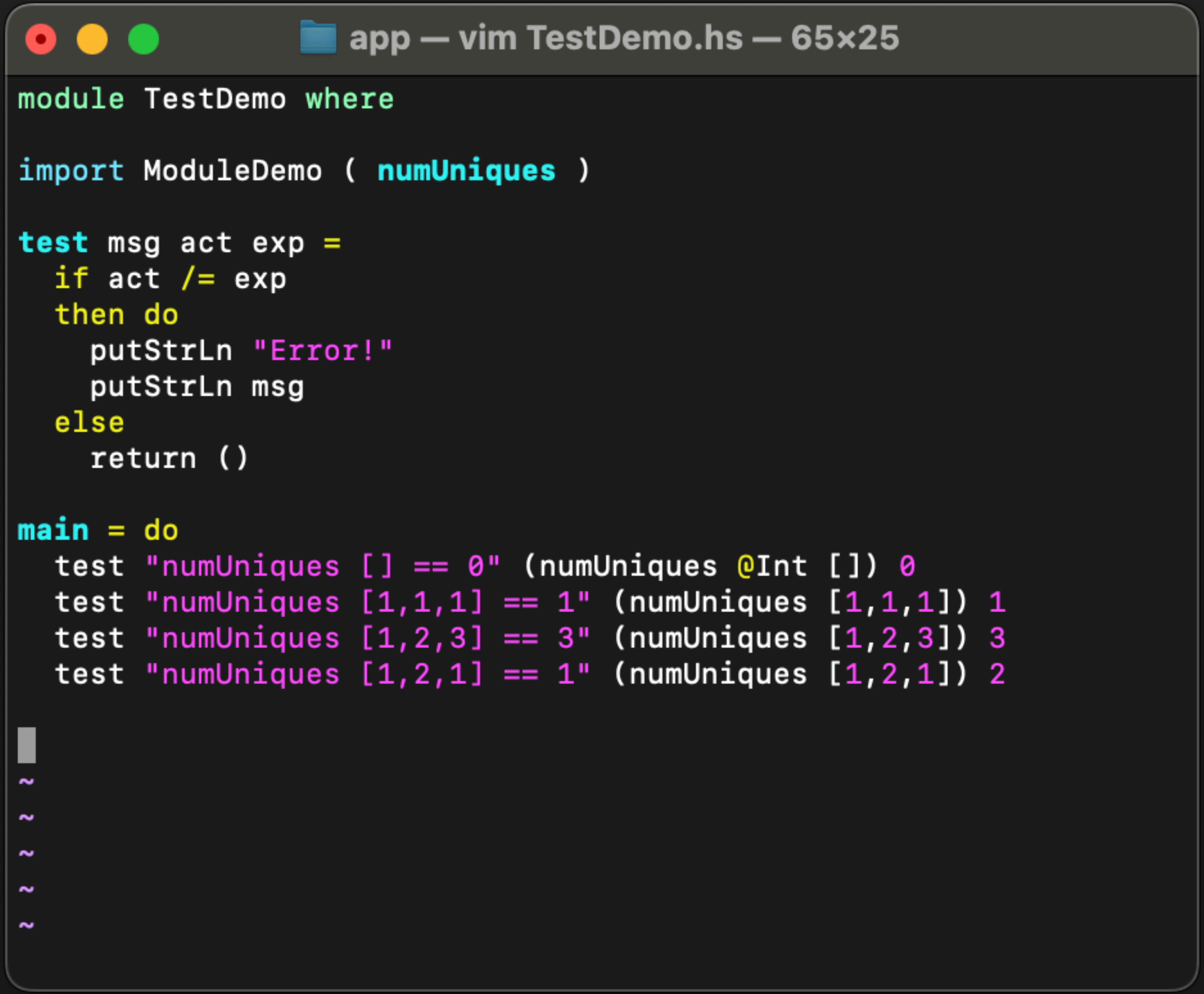
- Create a stack project
- Copy your code for Expressions there
- Make sure it builds and executes

OUTLINE

- Modules
- Packages
- Creating your own package
- Unit testing

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`)



```haskell
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