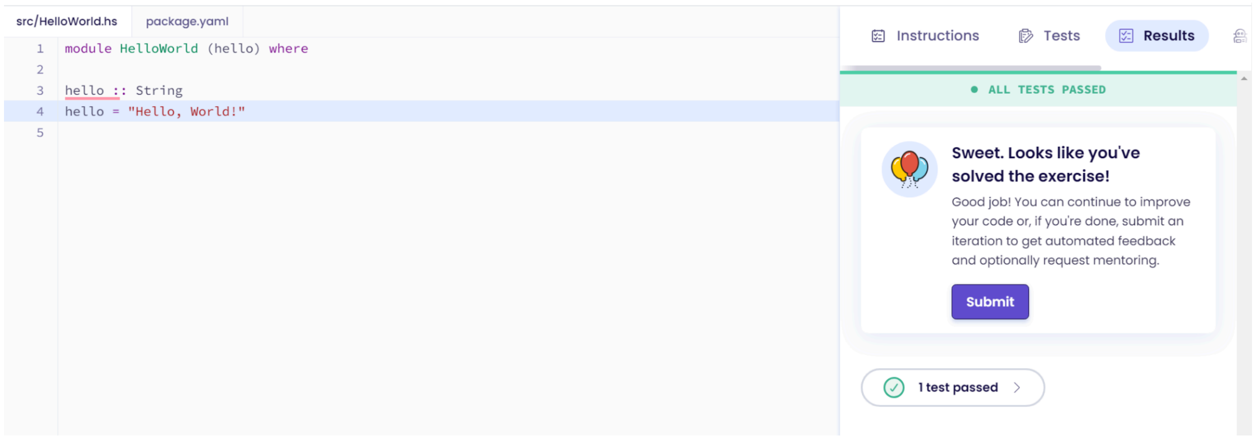
Hello World



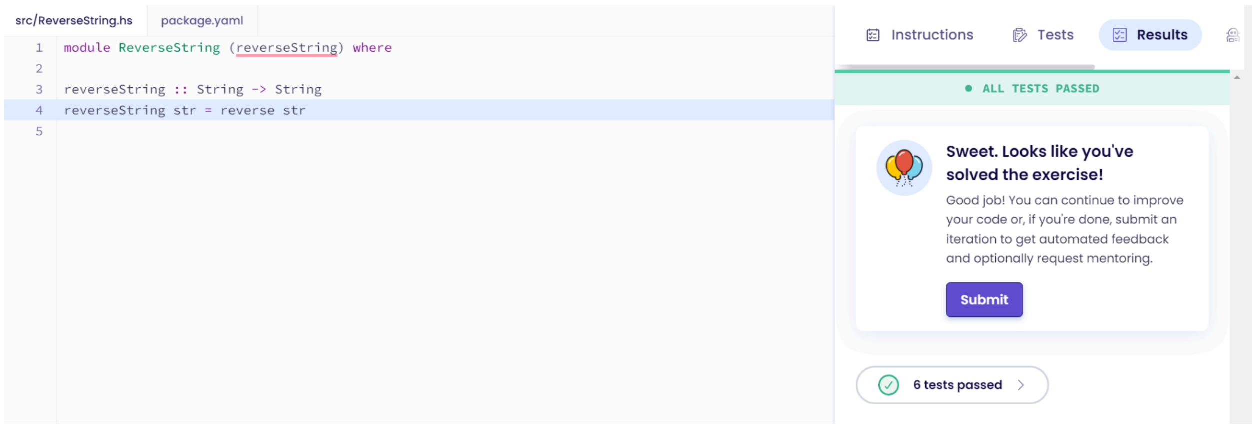
```haskell
module HelloWorld (hello) where

hello :: String
hello = "Hello, World!"

```

Sweet. Looks like you've solved the exercise!

Good job! You can continue to improve your code or, if you're done, submit an iteration to get automated feedback and optionally request mentoring.

1 test passed

Reverse String (easy)



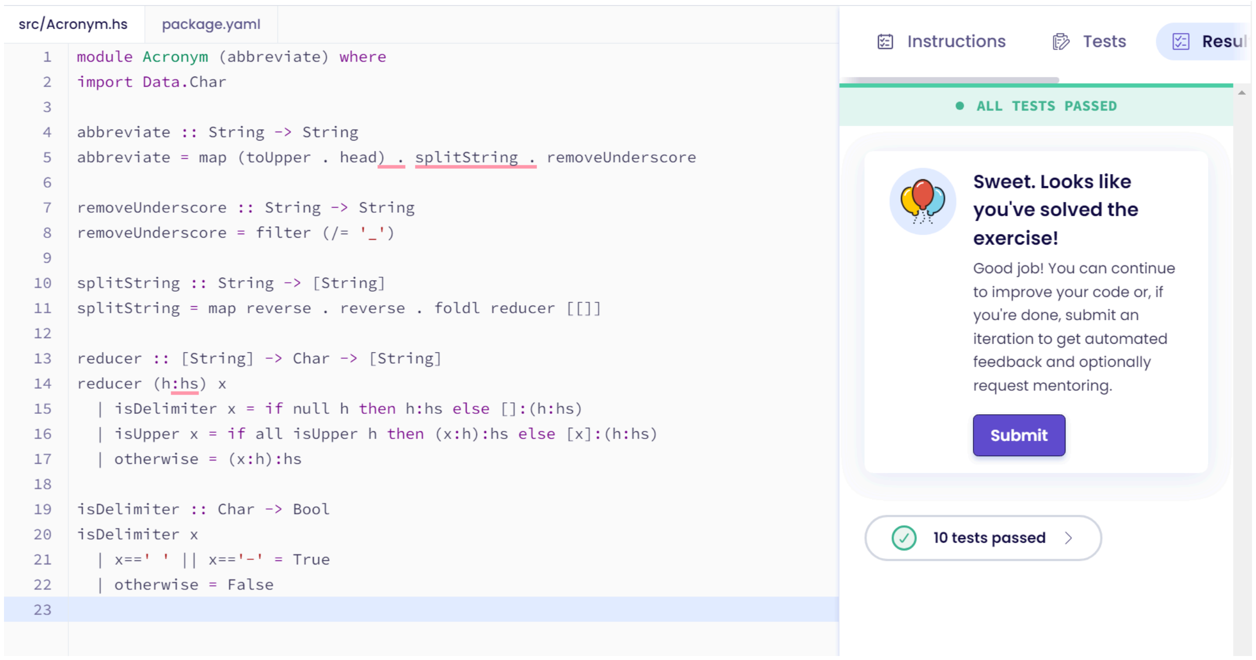
```haskell
module ReverseString (reverseString) where

reverseString :: String -> String
reverseString str = reverse str

```

Sweet. Looks like you've solved the exercise!

Good job! You can continue to improve your code or, if you're done, submit an iteration to get automated feedback and optionally request mentoring.

6 tests passed

Acronym (easy)



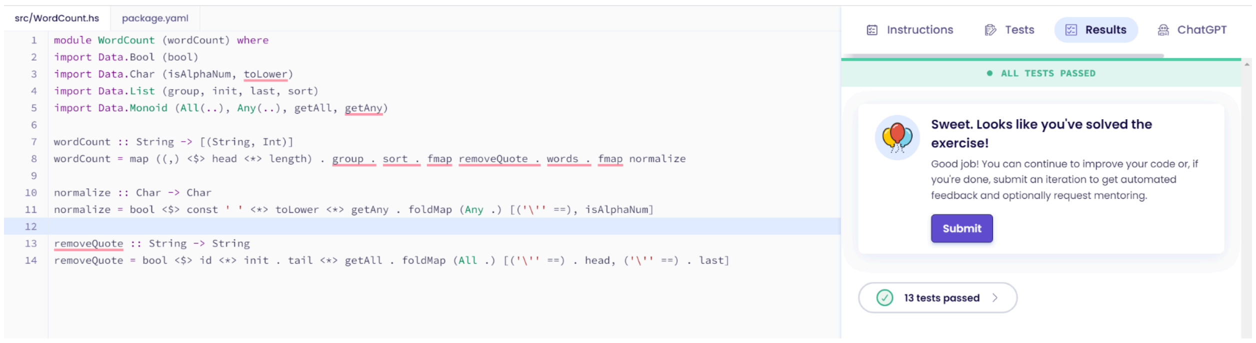
```haskell
module Acronym (abbreviate) where
import Data.Char

abbreviate :: String -> String
abbreviate = map (toUpper . head) . splitString . removeUnderscore

removeUnderscore :: String -> String
removeUnderscore = filter (/= '_')

splitString :: String -> [String]
splitString = map reverse . reverse . foldl reducer [[]]

reducer :: [String] -> Char -> [String]
reducer (h:hs) x
  | isDelimiter x = if null h then h:hs else []:(h:hs)
  | isUpper x = if all isUpper h then (x:h):hs else [x]:(h:hs)
  | otherwise = (x:h):hs

isDelimiter :: Char -> Bool
isDelimiter x
  | x==' ' || x=='-' = True
  | otherwise = False

```

Sweet. Looks like you've solved the exercise!

Good job! You can continue to improve your code or, if you're done, submit an iteration to get automated feedback and optionally request mentoring.

10 tests passed

Word Count (medium)



src/WordCount.hs package.yaml

```haskell
module WordCount (wordCount) where
import Data.Bool (bool)
import Data.Char (isAlphaNum, toLower)
import Data.List (group, init, last, sort)
import Data.Monoid (All(..), Any(..), getAll, getAny)

wordCount :: String -> [(String, Int)]
wordCount = map ((,) <$> head <*> length) . group . sort . fmap removeQuote . words . fmap normalize

normalize :: Char -> Char
normalize = bool <$> const ' ' <*> toLower <*> getAny . foldMap (Any .) [('\'' ==), isAlphaNum]

removeQuote :: String -> String
removeQuote = bool <$> id <*> init . tail <*> getAll . foldMap (All .) [('\'' ==) . head, ('\'' ==) . last]
```

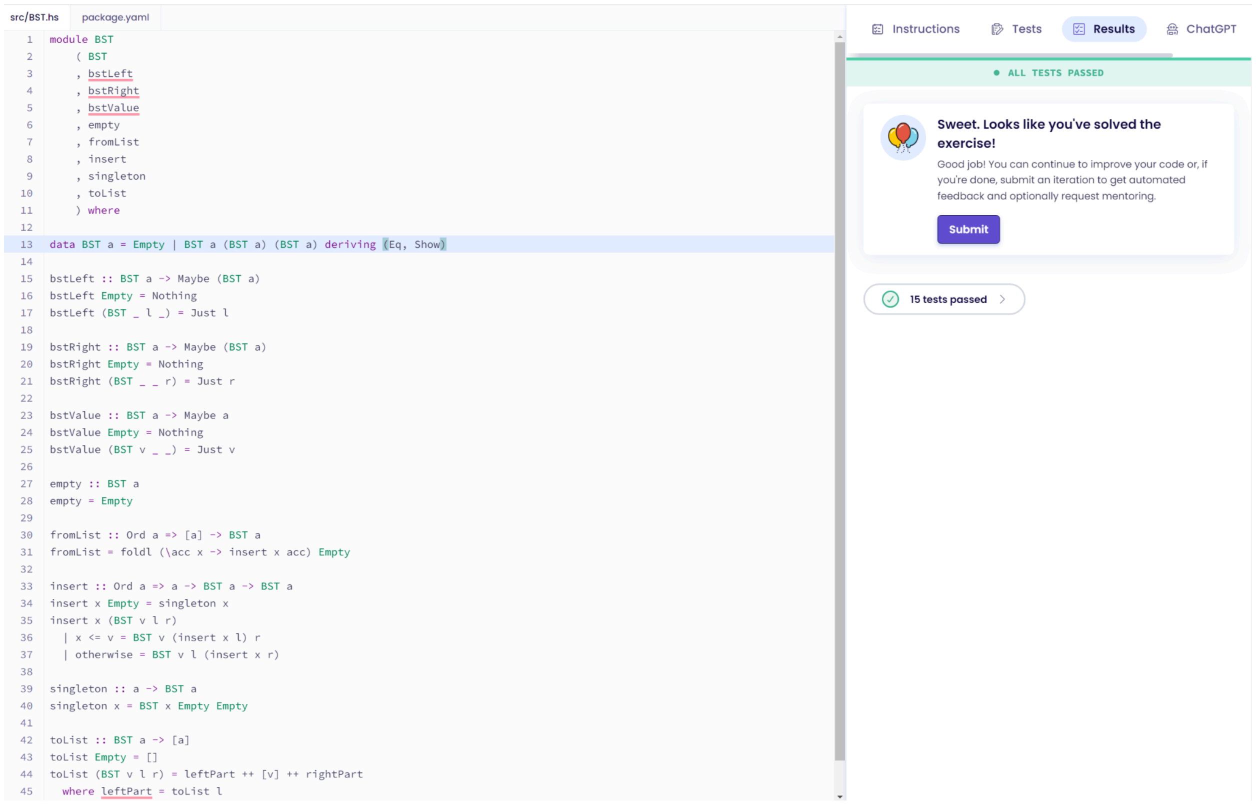
ALL TESTS PASSED

Sweet. Looks like you've solved the exercise!

Good job! You can continue to improve your code or, if you're done, submit an iteration to get automated feedback and optionally request mentoring.

13 tests passed

Binary Search Tree (medium)



src/BST.hs package.yaml

```haskell
module BST
    ( BST
    , bstLeft
    , bstRight
    , bstValue
    , empty
    , fromList
    , insert
    , singleton
    , toList
    ) where

data BST a = Empty | BST a (BST a) (BST a) deriving (Eq, Show)

bstLeft :: BST a -> Maybe (BST a)
bstLeft Empty = Nothing
bstLeft (BST _ l _) = Just l

bstRight :: BST a -> Maybe (BST a)
bstRight Empty = Nothing
bstRight (BST _ _ r) = Just r

bstValue :: BST a -> Maybe a
bstValue Empty = Nothing
bstValue (BST v _ _) = Just v

empty :: BST a
empty = Empty

fromList :: Ord a => [a] -> BST a
fromList = foldl (\acc x -> insert x acc) Empty

insert :: Ord a => a -> BST a -> BST a
insert x Empty = singleton x
insert x (BST v l r)
  | x <= v = BST v (insert x l) r
  | otherwise = BST v l (insert x r)

singleton :: a -> BST a
singleton x = BST x Empty Empty

toList :: BST a -> [a]
toList Empty = []
toList (BST v l r) = leftPart ++ [v] ++ rightPart
  where leftPart = toList l
```

ALL TESTS PASSED

Sweet. Looks like you've solved the exercise!

Good job! You can continue to improve your code or, if you're done, submit an iteration to get automated feedback and optionally request mentoring.

15 tests passed