Chapter - 14 → Testing

*) two broad categories of testing : a) unit testing
b) property testing

↓

Unit testing tests the smallest atomic units of software independent of one another. Unit testing allows the programmer to check that each fn. is performing the task it is meant to. Here, we assert that when the code runs with a specified i/p, the result is equal to the result we want.

↓

*) Spec testing is a somewhat newer version of unit testing. Written in terms of assertions that are in human-readable language.

↓

focus on specification testing : hspec library,
(hunit is also available)

↓

*) Property tests test the formal properties of programs w/o requiring formal proofs by allowing us to express a truth-valued, universally qualified fn. - usually equality - which will then be checked against randomly generated i/ps.

↓

The i/ps are generated randomly by the std. fones inside the QuickCheck library we use for property testing. This relies on the type system to know the kinds of data to generate. Default is for 100 pairs of i/ps & o/ps. If it fails any one of these, then we know our program doesn't have the specified

property. Property testing is fantastic for ensuring that we've met the minimum requirements to satisfy laws, such as the laws of monads or basic associativity.

→ Conventional testing : hspec library to demonstrate a test case

↓

test case for addition.

↓

Start a small project,

```
-- addition.cabal
name:                addition
version:             0.1.0.0
license-file:        LICENSE
author:              Chicken Little
maintainer:          sky@isfalling.org
category:            Text
build-type:          Simple
cabal-version:       >=1.10

library
  exposed-modules:     Addition
  ghc-options:         -Wall -fwarn-tabs
  build-depends:       base >=4.7 && <5
                     , hspec
  hs-source-dirs:      .
  default-language:    Haskell2010
```

LICENSE is req. for the build to not complain, even if empty.

hspec → declared dependency

. → the source module in the same module as the cabal file

↓

- Addition.hs
- addition.cabal
- LICENSE

↓

next steps include initializing a Stack file meant to describe what snapshot of Stackage we'll use :

↓

stack init

↓

Next we build the project that also installs the dependencies we require:

↓

stack build

↓

test bed for simple test case ready.

→ Truth according to Hspec:

```
import Test.Hspec
```

*) Our first hspec test

```
main :: IO()
main = hspec $ do
    describe "Addition" $ do
        it "1+1 is greater than 1" $ do
            (1+1) > 1 `shouldBe` True.
```

↓

asserted in both English & code that (1+1) should be greater than 1, and that is what hspec will test for us.

↓

Here we are nesting multiple do blocks. This syntax allows us to sequence monadic actions.

↓

hspec runs the code & checked that the arguments to shouldBe are equal.

↓

```
shouldBe :: (Eq a, Show a)
         => a -> a -> Expectation
```

↓ Contrait

```
(==) :: (Eq a) => a -> a -> Bool
```

↓

in a sense shouldBe is an augmented '==' embedded in hspec's model of the universe. Needs a show instance in order to render a value. meaning that show allows hspec to show us the result of tests, not just return a Bool.

```
main :: IO ()
main = hspec $ do
  describe "Addition" $ do
    it "1 + 1 is greater than 1" $ do
      (1 + 1) > 1 `shouldBe` True
    it "2 + 2 is equal to 4" $ do
      2 + 2 `shouldBe` 4
```

→ Addition
1+1 is greater than 1
2+2 is equal to 4

↓

```
dividedBy :: Integral a => a -> a -> (a, a)
dividedBy num denom = go num denom 0
  where go n   d count
        | n < d = (count, n)
        | otherwise =
            go (n - d) d (count + 1)
```

```
main :: IO ()
main = hspec $ do
  describe "Addition" $ do
    it "15 divided by 3 is 5" $ do
      dividedBy 15 3 `shouldBe` (5, 0)
    it "22 divided by 5 is\
       \ 4 remainder 2" $ do
      dividedBy 22 5 `shouldBe` (4, 2)
```

→ Short Exercise

```
recSum :: (Ord a, Num a) => a -> a -> a
recSum a b
  | a == 0 || b == 0 = 0
  | b == 1 = a
  | otherwise = a + recSum a $ b-1
```

3 * 2

```
main :: IO()
main = hspec $ do
  describe "recSum" $ do
    it "The product of 5 and 3 is 15" $ do
      recSum 5 3 `shouldBe` 15
    it "The product of 6 and 0 is 0" $ do
      recSum 6 0 `shouldBe` 0
```

→ Enter QuickCheck → first library to offer property testing

↓

done with assertion of laws or properties.

↙

*) add QuickCheck to build depends in the cabal file. Should already be installed since hspec has QuickCheck as a dependency.

*) import Test.QuickCheck

*) to the same describe block as the others

```
it "x+1 is always
    \ greater than x" $ do
    property $ \x -> x+1 > (x :: Int)
```

the compiler won't know what concrete type to align, and we would get ambiguity based errors.

↓

returns x+1 is always greater than a

↓

default no. of tests in Quickcheck = 100

*) Arbitrary instances : Quickcheck relies on a typeclass called Arbitrary and a newtype called Gen for generating its random data.

↓

arbitrary is a value of type Gen

↓

way of setting a default generator for a type.

↓

When we use the arbitrary, we have to specify the type to dispatch the right typeclass instance, as types & typeclass instances form unique pairings.

↓

Just a value, how to see a list of values of the correct type?

↓

using sample & sample' from Test.QuickCheck module in order to see some random data

↓

```
sample :: Show a => Gen a -> IO ()
sample' :: Gen a -> IO [a]
```

necessary since we are using a global resource of random values to generate the data.

↓

Here we're using IO so that our func. that generates our data can return a different result each time by pulling from a global resource of random values.

↓

Arbitrary typeclass is being used in order to provide a generator for sample. This isn't necessary if we have a Gen value ready to go already.

↓

Gen is a newtype with a single type argument; exists for wrapping up a func. to generate pseudo random values.

↓

The func. takes an argument that is usually provided by some kind of random value generator to give

as a pseudorandom value of the type, assuming it's a type that has an instance of the Arbitrary typeclass

↓

This is what we get, when we use the sample funcs. we use the arbitrary value but with a type, which gives us a list of random values.

↓

```
>> sample (arbitrary :: Gen Int)
>> sample (arbitrary :: Gen Double)
```

↓

Running sample arbitrary w/o defining the Gen type, in ghci, it defaults to () and returns a list of empty tuples. If mentioned in source file, we get ambiguity related errors.

↓

We can specify our own data for generating Gen values.

ex 1.
```
trivialInt :: Gen Int
trivialInt = return 1
```

Gen Int → returns a value inside a monad

↓

always returns 1

ex 2.
```
oneThroughThree :: Gen Int
oneThroughThree = elements [1, 2, 3]
```

↓

sample' oneThroughThree → returns a set of values sampled from [1..3]

with eq. prob. Prob. can be influenced by inc. the freq. of an element in the list (obvio!!)

*) choose and elements from QuickCheck library as generator of values.

```
choose :: System.Random.Random a
       => (a,a) -> Gen a

elements :: [a] -> Gen a
```

↓

```
genBool :: Gen Bool
genBool = choose (False, True)

genBool' :: Gen Bool
genBool' = elements [False, True]
```

↓

```
genOrdering :: Gen Ordering
genOrdering = elements [LT, EQ, GT]
```

*) complex examples:

```
genTuple :: (Arbitrary a, Arbitrary b)
         => Gen (a, b)
genTuple = do
    a <- arbitrary
    b <- arbitrary
```

return (a, b)

↓

>> sample genTuple

((),())
((),())
((),())

} defaulting a, b to ().

↳ initiates with 0 & 0.0

>> sample (genTuple :: Gen (Int, Float))

*) lists, Char, Maybe and Either among other instances of Arbitrary can be generated.

*) genEither :: (Arbitrary a, Arbitrary b)
⇒ Gen (Either a b)

```
genEither = do
    a ← arbitrary
    b ← arbitrary
    elements [Left a, Right b]
```

*) equal probability for Just & Nothing in Maybe

```
genMaybe :: Arbitrary a ⇒ Gen (Maybe a)
genMaybe = do
    a ← arbitrary
    elements [Nothing, Just a]
```

*) to influence probability of Just vs Nothing

```
genMaybe' :: Arbitrary a ⇒ Gen (Maybe a)
```

```
genMaybe' = do
    a ← arbitrary
    frequency [(1, return Nothing)
              ,(3, return (Just a))]
```

↓

inplace of elements which operates on equal probability, frequency uses weighted probability.

*) using Quickcheck w/o Hspec

↓

no longer required to specify → type (inferred from fn. defn) x in expression.

because the type of the func. provides that.

↓

```
prop_addnGreater :: Int → Bool
prop_addnGreater x = x+1 > x

runQc :: IO ()
runQc = quickCheck prop_addnGreater
```

↓

reports how many tests it ran, and if failed, then first test case to fail.

*) Quick check has some common error boundary, which will get tested. for example 0 happens to be a frequent point of failure, so Quickcheck tries to ensure that it is always tested.

→ Morse Code : a new project to play with testing

*) when we use `stack new project-name` to start a new project instead of `stack init` for an existing project, it automatically generates a file called setup.hs that looks like this,

```
import Distribution.Simple
main = defaultMain
```

} rarely need to modify it, good to recognize its there.

↓

configuring .cabal file properly. Stack new project-name already generates a few of these fields which needed to be added to.

```
name:                morse
version:             0.1.0.0
license-file:        LICENSE
author:              Chris Allen
maintainer:          cma@bitemyapp.com
category:            Text
build-type:          Simple
cabal-version:       >=1.10

library
  exposed-modules:     Morse
  ghc-options:         -Wall -fwarn-tabs
  build-depends:       base >=4.7 && <5
                     , containers
                     , QuickCheck
  hs-source-dirs:      src
  default-language:    Haskell2010
```

```
executable morse
  main-is:             Main.hs
  ghc-options:         -Wall -fwarn-tabs
  hs-source-dirs:      src
  build-depends:       base >=4.7 && <5
                     , containers
                     , morse
                     , QuickCheck
  default-language:    Haskell2010
```

```
test-suite tests
  ghc-options: -Wall -fno-warn-orphans
  type: exitcode-stdio-1.0
  main-is: tests.hs
  hs-source-dirs: tests
  build-depends:       base
                     , containers
                     , morse
                     , QuickCheck
  default-language:    Haskell2010
```

remember to

Capitalize

↓

Once this is setup, next we setup the src directory and Morse.src as our exposed modules.

↓

```
module Morse
    ( Morse
    , charToMorse
    , morseToChar
    , stringToMorse
    , letterToMorse
    ) where

import qualified Data.Map as M

type Morse = String

letterToMorse :: (M.Map Char Morse)
letterToMorse = M.fromList [key-value pairings]
```

to be understood as a balanced binary tree with each node being a (key value pair)

↓ to be orderable

↓ for efficient search

↓ list of tuples representing the mentioned key value pairs.

↓ converting char from one type to another

```
morseToLetter :: M.Map Morse Char
morseToLetter =
    M.foldWithKey (flip M.insert) M.empty letterToMorse
```

```
charToMorse :: Char → Maybe Morse
charToMorse c = M.lookup c letterToMorse

stringToMorse :: String → maybe [Morse]
stringToMorse s =
    sequence $ fmap charToMorse s

morseToChar :: Morse → maybe Char
morseToChar m = M.lookup m morseToLetter
```

lookup fn. for the map

key to lookup

map to look in

*) main event : setting up main module that will handle our Morse code conversions.

```
module Main where

import Control.Monad (forever, when)
import Data.List (intercalate)
import Data.Traversable (traverse)
import Morse (stringToMorse, morseToChar)
import System.Exit (exitFailure, exitSuccess)
import System.IO (hGetLine, hIsEof, stdin)
```

↓

```
convertToMorse :: IO ()
convertToMorse = forever $ do
    weAreDone ← hIsEOF stdin
    when weAreDone exitSuccess

    line ← hGetLine stdin
    convertLine line
    where
      convertLine line = do
        let morse = stringToMorse line
        case morse of
          Just x → putStrLn (intercalate " " x)
          Nothing → do
            putStrLn $ "Error: " ++ line
            exitFailure
```

⇓

```
convertFromMorse :: IO ()
convertFromMorse = forever $ do
    weAreDone ← hIsEOF stdin
    when weAreDone exitSuccess

    line ← hGetLine stdin
    convertLine line
    where convertLine line = do
            let decoded :: Maybe String
                decoded = traverse morseToChar
```

(words line)

```
case decoded of
    (Just x) → putStrLn x
    Nothing → do
        putStrLn "Error: " ++ line
        exitFailure
```

↓

```
main :: IO()
main = do
    mode ← getArgs
    case mode of
        [arg] →
            case arg of
                "from" → convertFromMorse
                "to" → convertToMorse
                _ → argError
        _ → argError

    where argError = do
            putStrLn "correct arg bro:"
            exitFailure.
```

*) to get the location of the exec using stack, use

stack exec which morse

*) Time to test : test suite in their own directory & file.

-- tests/tests.hs.

```
module Main where

import qualified Data.Map as M.
import Morse
import Test.QuickCheck
```

↓ setting up generators to ensure that sensible random values are used to test our prg

```
allowedChars :: [Char]
allowedChars = M.keys letterToMorse

allowedMorse :: [Morse]
allowedMorse = M.elems letterToMorse

charGen = gen Char
charGen = elements allowedChars

morseGen = gen Morse
morseGen = elements allowedMorse
```

↓ property we are looking to check.

String → morse → string should be same

```
prop_thereAndBackAgain :: Property
prop_thereAndBackAgain =
    forAll charGen
        (\c → ((charToMorse c) >>= morseToChar)
```

= = Just c)

↳ setting up main

```
main :: IO ()
main = quickCheck prop_thereAndBackAgain
```

↓

test using repl using the cmd :

stack ghci morse:tests → directory to load

↓

project

↓

builds & loads all requirements.

→ Arbitrary instances : an imp. part of QuickCheck is learning to write instances of the Arbitrary typeclass for our datatypes.

↓

necessary convenience for our code to integrate cleanly with QuickCheck code.

↓

i) Trivial datatype

```
data Trivial = Trivial
             deriving (Eq, Show)

trivialGen :: Gen Trivial
trivialGen = return Trivial
```

→ raises it to the required Gen structure.

```
instance Arbitrary Trivial where
  arbitrary = trivialGen
```

↓

```
main :: IO()
main = sample trivialGen
```

Gen values are generators of random values that QuickCheck uses to get test values from.

ii) Identity crisis

```
data Identity a =
  Identity a
  deriving()
```

```
identityGen :: Arbitrary a => Gen (Identity a)
identityGen = do
  a <- arbitrary
  return (Identity a)
```

(using Gen monad to pluck a single type value of type a out of the air)

↓

can be made the default generator for the Identity type by making it the arbitrary value in the Arbitrary instance.

↓

```
instance Arbitrary a => Arbitrary (Identity a) where
  arbitrary = identityGen
```

↓

```
identityGenInt :: Gen (Identity Int)
identityGenInt = identityGen.
```

→ making a gen. suitable for

↓

sample identityGenInt

sampling by making it unambiguous.

↓

changing the concrete type of Identity's type arg. allows us to generate values of different type.

iii) Arbitrary Products

```
data Pair a b =
    Pair a b
    deriving ()

pairGen :: (Arbitrary a, Arbitrary b)
        => Gen (Pair a b)
pairGen = do
    a <- arbitrary
    b <- arbitrary
    return (Pair a b)

instance (Arbitrary a, Arbitrary b) =>
         Arbitrary (Pair a b) where
    arbitrary = pairGen

pairGenIntString :: Gen (Pair Int String)
pairGenIntString = pairGen
```

iV) Arbitrary Sum types.

```
import Test.QuickCheck.Gen (oneof)
```

↓

We need to represent the exclusive properties of sum types in our gen. One way to do that is to pull out as many

arbitrary values as we require for the cases of our sum type. We have two data constructors in this sum type, so we'll want two arbitrary values. Then we'll repack them into Gen values, resulting in a value of type [Gen a] that can then be passed to oneof:

```
data Sum a b =
    First a
  | Second b
  deriving ()

sumGenEqual :: (Arbitrary a, Arbitrary b)
            => Gen (Sum a b)

sumGenEqual = do
  a <- arbitrary
  b <- arbitrary
  oneof [return $ First a,
         return $ Second b]
```

→ equal weights for sampling

```
sumGenCharInt : Gen (Sum Char Int)
sumGenCharInt = sumGenEqual
```

↓ different weighted

```
sumGenFirst :: (Arbitrary a, Arbitrary b)
            => Gen (Sum a b)

sumGenFirst = do
```

```
a ← arbitrary
b ← arbitrary
frequency [(10, return $ first a),
           (1, return $ second b)]
```

```
sumGenFirstCharInt :: Gen (Sum Char Int)
sumGenFirstCharInt = sumGenFirstPEs
```

↓

Key takeaway is that the Arbitrary instance for a datatype doesn't have to be the only way to generate or provide random values of our datatype for QuickCheck tests. We can develop other specific means as well.

*) CoArbitrary : counterpart to Arbitrary → enables generation of fns. fitting a particular type.

↓

Rather than talking about random values we can get via Gen, it lets us provide fns. with a value of type a as an argument in order to vary a Gen.

↓

```
arbitrary :: Arbitrary a => Gen a

coarbitrary :: CoArbitrary a
            => (a) -> Gen b -> Gen b
```

(a) → used to return a variant of

*) As long as our datatype has a Generic instance derived, we can get these instances for free.

```
{-# LANGUAGE DeriveGeneric #-}

module CoArbitrary where

import GHC.Generics
import Test.QuickCheck

data Bool' = True' | False'
             deriving ()

instance CoArbitrary Bool'

trueGen :: Gen Int
trueGen = coarbitrary True' arbitrary
```

↓

essentially this lets us randomly generate a fn. If we ever find ourself wanting to randomly generate anything with the (→) type inside it somewhere, it becomes salient in a hurry.