Laboratorium 7.

0) Przygotowanie środowiska

1. W konsoli/terminalu wpisujemy kolejno:

```
$ cd
$ mkdir haskell-lab7
$ cd haskell-lab7
```

1) Narzędzie `stack` : *pierwsze kroki* :)

1. Sprawdzamy, czy narzędzie `stack` jest zainstalowane:

```
$ which stack
```

2. Jeśli tak, to w konsoli/terminalu wpisujemy:

```
$ stack --version
$ stack --version | grep 'QuickCheck'
$ stack --version | grep 'HUnit'
$ stack --version | grep 'hspec'
```

3. Zapoznajemy się z zawartością strony; jeśli `stack` nie jest dostępny w systemie, to instalujemy go

4. W konsoli/terminalu wpisujemy kolejno:

```
$ stack new hello-stack
$ cd hello-stack
$ stack setup
$ tree .
```

5. Analizujemy strukturę (nowo utworzonego) projektu

6. W edytorze Visual Studio Code (lub Atom) otwieramy katalog `hello-stack`

7. Sprawdzamy zawartość plików: `app/Main.hs` , `src/Lib.hs` i `test/Spec.hs`

8. Analizujemy zawartość plików: `package.yaml` , `stack.yaml` i `hello-stack.cabal`

9. W konsoli/terminalu wpisujemy kolejno:

```
$ stack build
$ stack exec hello-stack-exe
$ stack test
```

10. Następnie wpisujemy:

```
$ ls -a
```

i analizujemy zawartość katalogu `.stack-work/dist/`

1. ***Zadania***:
    1. Znajdź lokalizację pliku `hello-stack-exe` i uruchom go
    2. Znajdź lokalizację pliku `hello-stack-test` i uruchom go
    3. Zapoznaj się z zawartością strony stackage.org
    4. (*opcjonalne*) Zapoznaj się z językiem YAML, np. tu
    5. Znajdź w pliku `stack.yaml` wartość klucza `resolver`
    6. (*opcjonalne*) Znajdź w plikach projektu fragmenty, które "wyglądają na miejsca" deklaracji *zależności* projektu (pakiety, biblioteki, frameworki,...)

2) Biblioteka `QuickCheck` : *elementarne testy*

1. Przechodzimy do katalogu `haskell-lab7`
2. Tworzymy nowy projekt:

```
$ stack new hello-quickcheck
$ cd hello-quickcheck
```

3. W pliku `package.yaml` zmieniamy fragment

```
dependencies:
 - base >= 4.7 && < 5
```

na

```
dependencies:
 - base >= 4.7 && < 5
 - QuickCheck
```

4. W konsoli/terminalu wpisujemy kolejno:

```
$ stack solver
$ stack test
```

5. Zmieniamy zawartość pliku `Spec.hs` na:

```haskell
import Test.QuickCheck

prop_plusAssociativeInt :: Int -> Int -> Int -> Bool
prop_plusAssociativeInt x y z = x + (y + z) == (x + y) + z

main :: IO ()
main = do
  putStrLn "\n*** Testing prop_plusAssociativeInt... ***"
  quickCheck prop_plusAssociativeInt
```

6. Uruchamiamy testy:

```
$ stack test
```

   i analizujemy wynik

7. [w pliku `Spec.hs`] zmieniamy wywołanie funkcji `quickCheck` na:

```haskell
quickCheck (withMaxSuccess 1000 prop_plusAssociativeInt)
```

8. Uruchamiamy testy:

```
$ stack test
```

   i analizujemy wynik

9. W pliku `Spec.hs` dodajemy (np. zaraz po `prop_plusAssociativeInt`)

```haskell
prop_reverse :: [Int] -> Bool
prop_reverse xs = reverse (reverse xs) == xs
```

   oraz modyfikujemy funkcję `main`

```haskell
main = do
  putStrLn "\n*** Testing prop_plusAssociativeInt... ***"
  quickCheck (withMaxSuccess 1000 prop_plusAssociativeInt)
  putStrLn "\n*** Testing prop_reverse... ***"
  quickCheck (withMaxSuccess 500 prop_reverse)
```

10. Uruchamiamy testy i analizujemy wynik

11. W pliku `Spec.hs` dodajemy

```haskell
prop_halfIdentity :: Double -> Bool
prop_halfIdentity x = ((* 2) . (/ 2) $ x) == x
```

12. W funkcji `main` dodajemy:

```
putStrLn "\n*** Testing prop_halfIdentity... ***"
quickCheck prop_halfIdentity
```

13. Ponownie uruchamiamy testy i analizujemy wynik

14. ***Zadania***:

    1. Napisz i przetestuj odpowiednik `prop_plusAssociativeInt` dla warości typu `Double`, czyli

       ```
       prop_plusAssociativeDouble :: Double -> Double -> Double -> Bool
       ```

       Sprawdź znaleziony błąd (wykonując obliczenie "ręcznie", np. w GHCi)

    2. Napisz i uruchom testy sprawdzające przemienność dodawania dla wartości typu `Int` i `Double`, czyli

       ```
       prop_plusCommutativeInt :: Int -> Int -> Bool
       prop_plusCommutativeDouble :: Double -> Double -> Bool
       ```

       Powtórz testy dla kilku różnych wartości `withMaxSuccess`

    3. Napisz i uruchom testy sprawdzające następujące warunki (dla listy):

       ```
       reverse [x] == [x]
       ```

       ```
       reverse (xs ++ ys) == reverse xs ++ reverse ys
       ```

    4. Napisz i uruchom testy sprawdzające następujący warunek:

       ```
       reverse (xs ++ ys) == reverse ys ++ reverse xs
       ```

       Powtórz testy dla kilku różnych wartości `withMaxSuccess`

    5. Zapoznaj się z opisem biblioteki QuickCheck

    6. Uruchom testy za pomocą funkcji `verboseCheck` (zamiast `quickCheck`)

    7. [*opcjonalne*] Sprawdź dostępność odpowiedników biblioteki `QuickCheck` dla znanych Ci języków

3) Biblioteka `QuickCheck`: *właściwości warunkowe, analiza danych testowych, nieskończone struktury danych*

1. Tworzymy nowy projekt:

```
$ stack new qch2
$ cd qch2
```

2. W pliku `package.yaml` zmieniamy fragment

```
dependencies:
 - base >= 4.7 && < 5
```

na

```
dependencies:
 - base >= 4.7 && < 5
 - QuickCheck
```

3. W konsoli/terminalu wpisujemy:

```
$ stack test
```

4. Zmieniamy zawartość pliku `Spec.hs` na:

```
import Test.QuickCheck

prop_MaxOfX_LtOrEq_YIsY :: Int -> Int -> Property
prop_MaxOfX_LtOrEq_YIsY x y = x <= y ==> max x y == y

main :: IO ()
main = do
  putStrLn "\n*** Testing prop_MaxOfX_LtOrEq_YIsY... ***"
  quickCheck prop_MaxOfX_LtOrEq_YIsY
```

5. Uruchamiamy testy:

```
$ stack test
```

i analizujemy wynik

6. W pliku `Spec.hs` dodajemy:

```
isOrdered xs = and $ zipWith (<=) xs (tail xs)
insertToOrdList e xs = takeWhile (< e) xs ++ [e] ++ dropWhile (< e) xs

prop_InsertPreservesOrder :: Int -> [Int] -> Property
prop_InsertPreservesOrder x xs = isOrdered xs ==> isOrdered (insertToOrdList x xs)
```

oraz modyfikujemy funkcję `main`, dodając:

```
putStrLn "\n*** Testing prop_InsertPreservesOrder... ***"
quickCheck prop_InsertPreservesOrder
```

7. Uruchamiamy testy i analizujemy wynik
8. W pliku `Spec.hs` dodajemy:

```
prop_InsertPreservesOrder2 :: Int -> [Int] -> Property
prop_InsertPreservesOrder2 x xs =
  isOrdered xs ==> classify (null xs) "empty list" $
    isOrdered (insertToOrdList x xs)
```

oraz modyfikujemy funkcję `main`, dodając:

```
putStrLn "\n*** Testing prop_InsertPreservesOrder2... ***"
quickCheck prop_InsertPreservesOrder2
```

9. Uruchamiamy testy i analizujemy wynik
10. W pliku `Spec.hs` dodajemy:

```
prop_InsertPreservesOrder3 :: Int -> [Int] -> Property
prop_InsertPreservesOrder3 x xs =
  isOrdered xs ==> collect (length xs) $
    isOrdered (insertToOrdList x xs)
```

oraz modyfikujemy funkcję `main`, dodając:

```
putStrLn "\n*** Testing prop_InsertPreservesOrder3... ***"
quickCheck prop_InsertPreservesOrder3
```

11. Uruchamiamy testy i analizujemy wynik
12. W pliku `Spec.hs` dodajemy:

```
prop_DoubleCycleEqOneCycle :: [Int] -> Property
prop_DoubleCycleEqOneCycle xs = not (null xs) ==> cycle xs == cycle (xs ++ xs)
```

oraz modyfikujemy funkcję `main`, dodając:

```
putStrLn "\n*** Testing prop_DoubleCycleEqOneCycle... ***"
quickCheck prop_DoubleCycleEqOneCycle
```

13. Uruchamiamy testy i analizujemy wynik
14. Przerywamy wykonanie programu `[Ctrl+C]`
15. Zmieniamy definicję funkcji `prop_DoubleCycleEqOneCycle` na:

```
prop_DoubleCycleEqOneCycle :: [Int] -> Int -> Property
prop_DoubleCycleEqOneCycle xs n =
  not (null xs) && n >= 0 ==>
    take n (cycle xs) == take n (cycle (xs ++ xs))
```

16. Uruchamiamy testy i analizujemy wynik
17. ***Zadania***:
    1. Napisz test sprawdzający warunek `if (x <= y) then min x y == x`
    2. Sprawdź działanie następującego wariantu `prop_InsertPreservesOrder` :

```
prop_InsertPreservesOrder4 :: Int -> [Int] -> Property
prop_InsertPreservesOrder4 x xs =
  forAll orderedList $ \xs -> isOrdered (insertToOrdList x xs)
```

    3. [*opcjonalne*] Napisz testy sprawdzające zachowanie warunku uporządkowania listy podczas wstawiania elementów dla przypadku listy uporządkowanej malejąco. Sprawdź dane testowe (`classify` , `collect`)
    4. [*opcjonalne*] Zapoznaj się z publikacją: K. Claessen, J. Hughes - QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs

4) Środowisko projektu w GHCi: *stack ghci*

1. Uruchamiamy konsolę GHCi:

```
$ ghci
```

2. W konsoli GHCi wpisujemy kolejno:

```
ghci> :t cycle
ghci> cycle [1,2,3] -- po chwili przerywamy wykonanie [Ctrl+C]
ghci> take 10 $ cycle [1,2,3]
ghci> take 10 $ cycle ([1,2,3] ++ [1,2,3])
ghci> :t quickCheck
ghci> :t classify
ghci> :t collect
ghci> import Test.QuickCheck
```

3. Zamykamy konsolę GHCi (`:q`)
4. Sprawdzamy, czy katalogiem bieżącym jest `qch2` (jeśli nie, to ustawiamy `qch2` jako katalog bieżący)
5. Uruchamiamy ponownie GHCi, ale tym razem w środowisku projektu:

```
$ stack ghci
```

6. W konsoli GHCi wpisujemy kolejno:

```
ghci> :t quickCheck
ghci> :t classify
ghci> :t collect
ghci> import Test.QuickCheck
ghci> :t quickCheck
ghci> :t classify
ghci> :t collect
ghci> :i Testable
ghci> :i Property
ghci> :i Gen
```

7. Następnie wpisujemy:

```
ghci> :l test/Spec.hs
ghci> quickCheck prop_MaxOfX_LtOrEq_YIsY
```

8. ***Zadania***:
    1. Zapoznaj się z opisem stack ghci
    2. Uruchom w GHCi pozostałe testy (tzn. `prop_InsertPreservesOrder` i kolejne)
    3. Uruchom w GHCi kolejno wszystkie testy z różnymi wartościami parametru `withMaxSuccess`
    4. Sprawdź w GHCi działanie następujących testów:

```
ghci> quickCheck $ forAll (choose (1,100)) (\n -> n > 0)
ghci> quickCheck $ forAll (arbitrary :: Gen Int) (\n -> n > 0)
```

    5. Uruchom w GHCi funkcję `main` z pliku `Spec.hs`
    6. Uruchom w GHCi funkcję `main` z pliku `app/Main.hs`

5) Generatory danych testowych 1: *typy wbudowane*

1. Tworzymy nowy projekt i przechodzimy do katalogu projektu:

```
$ stack new qch3
$ cd qch3
```

2. W sekcji `dependencies` pliku `package.yaml` dodajemy `QuickCheck`

3. Uruchamiamy `GHCi` w środowisku projektu

```
$ stack ghci
```

4. W konsoli GHCi wpisujemy kolejno:

```
ghci> :t quickCheck
ghci> import Test.QuickCheck
ghci> :t quickCheck
ghci> :t arbitrary
ghci> :i Arbitrary
ghci> :i Gen
ghci> :t generate
ghci> generate $ return 1
```

5. Następnie wpisujemy:

```
ghci> :t choose
ghci> generate $ choose (0, 1) -- powtarzamy kilka razy
ghci> generate $ choose (0.0, 1.0) -- jw.
ghci> generate $ choose ('a', 'z') -- jw.
ghci> generate $ choose ('A', 'Z') -- jw.
ghci> generate $ choose ('A', 'z') -- jw.
ghci> generate $ choose (False, True) -- jw.
```

6. Następnie wpisujemy:

```
ghci> :t elements
ghci> generate $ elements [1,10,100,1000] -- powtarzamy kilka razy
ghci> generate $ elements [1..10] -- jw.
ghci> generate $ elements ['a', 'e', 'i', 'o', 'u'] -- jw.
ghci> generate $ elements [False,True] -- jw.
ghci> generate $ elements [3.14, 2.71] -- jw.
```

7. Następnie wpisujemy:

```
ghci> generate (arbitrary :: Gen Bool) -- powtarzamy kilka razy
ghci> generate (arbitrary :: Gen (Bool, Bool)) -- jw.

ghci> generate (arbitrary :: Gen (Int, Bool, Double)) -- jw.
ghci> generate (arbitrary :: Gen [(Int,Bool,Double)]) -- jw.
```

```
ghci> listGen = generate $ frequency [(1, return [1]), (3, return [1,2])] -- jw.
ghci> listGen -- powtarzamy 10 razy
ghci> generate $ suchThat (arbitrary :: Gen Int) (\e -> 2 * e^2 - 3 * e > 0) -- powtarz
```

8. Następnie wpisujemy:

```
ghci> generate $ listOf (arbitrary :: Gen Int) -- powtarzamy kilka razy
ghci> generate $ listOf1 (arbitrary :: Gen Int) -- jw.
ghci> generate $ vectorOf 5 (arbitrary :: Gen Bool) -- jw.
ghci> generate $ vectorOf 5 (arbitrary :: Gen Int) -- jw.
ghci> generate $ vectorOf 2 (arbitrary :: Gen [Int]) -- jw.
ghci> generate $ shuffle [1..10] -- jw.
ghci> generate $ shuffle ['a'..'z'] -- jw.
ghci> generate $ sublistOf [1..10] -- jw.
ghci> generate $ (orderedList :: Gen [Int]) -- jw.
```

9. Następnie wpisujemy:

```
ghci> sample $ (arbitrary :: Gen [Int]) -- powtarzamy kilka razy
ghci> sample' $ (arbitrary :: Gen [(Int,Bool)]) -- jw.
```

10. ***Zadania***:

    1. Zapoznaj się z opisami wszystkich funkcji użytych w tym ćwiczeniu
    2. Sprawdź działanie następującego generatora:

    ```
    listGen :: Gen [Int]
    listGen = oneof [ return [], (:) <$> arbitrary <*> listGen ]
    ```

    3. Zmodyfikuj `listGen` tak, aby listy niepuste były tworzone 4 razy częściej niż puste (rozważ użycie `frequency`)
    4. [*opcjonalne*] Zapoznaj się z funkcjami `sized` i `resize`

6) Generatory danych testowych 2: *typy użytkownika*

1. Zmieniamy zawartość pliku `Spec.hs` projektu `qch3` na:

```
import Test.QuickCheck

data DaysOfWeek = Mon | Tue | Wed | Thu | Fri | Sat | Sun deriving (Eq, Show)

dayAfter Mon = Tue
dayAfter Tue = Wed
```

```
dayAfter Wed = Thu
dayAfter Thu = Fri
dayAfter Fri = Sat
dayAfter Sat = Sun
dayAfter Sun = Mon

prop_WeekCycle d = sevenTimesNext d == d
  where sevenTimesNext d' = iterate dayAfter d' !! 7

instance Arbitrary DaysOfWeek where
  arbitrary = elements [ Mon, Tue, Wed, Thu, Fri, Sat, Sun ]

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

2. W pierwszej linii pliku `Spec.hs` dodajemy:

```
import Control.Monad
```

3. W pliku `Spec.hs` (np. nad funkcją `main`) dodajemy:

```
data List a = Nil | Cons a (List a) deriving (Show, Eq)

instance Arbitrary a => Arbitrary (List a) where
  arbitrary = oneof [ return Nil, liftM2 Cons arbitrary arbitrary ]

reverse' :: List a -> List a
reverse' = go Nil
  where go acc Nil = acc
        go acc (Cons x xs) = go (Cons x acc) xs

prop_RevRevList_Eq_List :: List Int -> Bool
prop_RevRevList_Eq_List xs = reverse' (reverse' xs) == xs
```

4. Zmieniamy funkcję `main` na:

```
main :: IO ()
main = do
  putStrLn "\n*** Testing prop_WeekCycle... ***"
  quickCheck prop_WeekCycle
  putStrLn "\n*** Testing prop_RevRevList_Eq_List... ***"
  quickCheck prop_RevRevList_Eq_List
```

Zapisujemy zmiany

5. Uruchamiamy test (`stack test`) i analizujemy wynik
6. W pliku `Spec.hs` dodajemy:

```
length' :: List a -> Int
length' Nil = 0
length' (Cons _ xs) = 1 + length' xs

prop_RevRevList_Eq_List2 :: List Int -> Property
prop_RevRevList_Eq_List2 xs = collect (length' xs) $ reverse' (reverse' xs) == xs
```

7. W funkcji `main` dodajemy:

```
putStrLn "\n*** Testing prop_RevRevList_Eq_List2... ***"
quickCheck prop_RevRevList_Eq_List2
```

8. Uruchamiamy testy i analizujemy wynik
9. W pliku `Spec.hs` dodajemy:

```
data BinTree a = EmptyBT | NodeBT (BinTree a) a (BinTree a) deriving Show

insertIntoBinTree :: a -> BinTree a -> BinTree a
insertIntoBinTree x EmptyBT = NodeBT EmptyBT x EmptyBT
insertIntoBinTree x (NodeBT lt a rt) =
  if depthOfBinTree lt <= depthOfBinTree rt
     then NodeBT (insertIntoBinTree x lt) a rt
     else NodeBT lt a (insertIntoBinTree x rt)

depthOfBinTree :: BinTree a -> Int
depthOfBinTree EmptyBT = 0
depthOfBinTree (NodeBT lt _ rt) = 1 + max (depthOfBinTree lt) (depthOfBinTree rt)

prop_InsToBinTreeIncrItsDepthByAtMost1 :: a -> BinTree a -> Bool
prop_InsToBinTreeIncrItsDepthByAtMost1 x t0 =
  depthOfBinTree (insertIntoBinTree x t0) - depthOfBinTree t0 <= 1
```

10. W funkcji `main` dodajemy:

```
putStrLn "\n*** Testing prop_InsToBinTreeIncrItsDepthByAtMost1... ***"
quickCheck (prop_InsToBinTreeIncrItsDepthByAtMost1 :: Int -> BinTree Int -> Bool)
```

11. W konsoli GHCi (`stack ghci`) uruchamiamy testy kilkukrotnie (`ghci> main`) i analizujemy wynik. Dlaczego czasy wykonania testów nie zawsze są równe? (*uwaga*: pomocne może być użycie `verboseCheck` zamiast `quickCheck`)

12. Zmieniamy definicję instancji `Arbitrary (BinTree a)` na następującą:

```
instance Arbitrary a => Arbitrary (BinTree a) where
  arbitrary = frequency [ (1, return EmptyBT),
                          (3, liftM3 NodeBT arbitrary arbitrary arbitrary ) ]
```

    i powtarzamy testy. Co się zmieniło?

13. Ponownie zmieniamy definicję instancji `Arbitrary (BinTree a)` , tym razem na następującą:

```
instance Arbitrary a => Arbitrary (BinTree a) where
   arbitrary = sized arbitraryBinTree

arbitraryBinTree 0 = return EmptyBT
arbitraryBinTree n =
  frequency [ (1, return EmptyBT),
              (3, liftM3 NodeBT (arbitraryBinTree (n `div` 2))
                                arbitrary
                                (arbitraryBinTree (n `div` 2))) ]
```

    i powtarzamy testy. Co się zmieniło?

14. ***Zadania***:

    1. Uruchom test właściwości `prop_WeekCycle` używając `verboseCheck` zamiast `quickCheck`
    2. Zmodyfikuj definicję

       ```
       instance Arbitrary a => Arbitrary (List a) where
         arbitrary = oneof [ return Nil, liftM2 Cons arbitrary arbitrary ]
       ```

       tak, aby średnia długość generowanych list wynosiła 5 (rozważ wykorzystanie `frequency`)
    3. Dodaj test sprawdzający, czy dodanie elementu do drzewa `BinTree` zwiększa jego liczbę elementów o 1, tzn.

       ```
       elemCountOfBinTree (insertIntoBinTree e tr) - elemCountOfBinTree tr == 1
       ```

    4. Uporządkuj strukturę projektu rozdzielając (do różnych plików) kod testowany i testujący
    5. Zapoznaj się z opisami bibliotek `HUnit` i `Hspec` oraz porównaj ich możliwości z biblioteką `QuickCheck`
    6. Utwórz nowy projekt zawierający zalążek biblioteki kolekcji (na początek może to być tylko np. stos i kolejka) i napisz kilka testów dla każdej kolekcji (`QuickCheck` oraz `HUnit` lub `Hspec`)

7) QuickCheck i biblioteka `Checkers`

1. Tworzymy nowy projekt i przechodzimy do katalogu projektu:

```
$ stack new qch4
$ cd qch4
```

2. W pliku `package.yaml` zmieniamy fragment:

```
dependencies:
 - base >= 4.7 && < 5
```

na

```
dependencies:
 - base >= 4.7 && < 5
 - QuickCheck
 - checkers
```

3. Zmieniamy zawartość pliku `Spec.hs` na:

```
import Control.Monad
import Test.QuickCheck
import Test.QuickCheck.Checkers
import Test.QuickCheck.Classes

main :: IO ()
main = do
  putStrLn "\n*** Testing Maybe ... ***"
  let t1 = undefined :: Maybe (Int, String, Int)
  quickBatch $ functor t1
  quickBatch $ applicative t1
  quickBatch $ monad t1
```

4. Uruchamiamy testy

```
$ stack test
```

5. Zapoznajemy się z opisem biblioteki Checkers

6. W pliku `Spec.hs` dodajemy:

```
newtype Identity a = Identity a deriving (Eq, Ord, Show)

instance Functor Identity where
```

```
  fmap f (Identity x) = Identity $ f x

instance Applicative Identity where
  pure = Identity
  Identity f <*> Identity x = Identity $ f x

instance Monad Identity where
  return = pure
  Identity x >>= fm = fm x

instance Arbitrary a => Arbitrary (Identity a) where
  arbitrary = liftM Identity arbitrary

instance Eq a => EqProp (Identity a) where
  (=-=) = eq
```

7. W funkcji `main` dodajemy:

```
putStrLn "\n*** Testing Identity ... ***"
let t2 = undefined :: Identity (Int, String, Int)
quickBatch $ functor t2
quickBatch $ applicative t2
quickBatch $ monad t2
```

   i uruchamiamy testy

8. ***Zadania***:

   1. Sprawdź spełnienie odpowiednich 'praw' dla typu `Either String Int` (dla instancji klas typu `Functor`, `Applicative` i `Monad`)
   2. Sprawdź spełnienie odpowiednich 'praw' dla typu `[Int]`
   3. Dla typu danych (reprezentującego listę):

      ```
      data List a = Nil | Cons a (List a) deriving (Show, Eq)
      ```

      napisz instancje klas typu: `Functor`, `Applicative` i `Monad` oraz `Arbitrary` i `EqProp` a następnie sprawdź, czy utworzone instancje spełniają odpowiednie 'prawa' (odpowiadające klasom typu `Functor`, `Applicative` i `Monad`)