1 Tutorial 1: The simplest case.

This is the simplest example of using the GenCheck framework to test an implementation against a specification. This module contains a revList (reverse) function that reverses the order of the elements in a list, and two properties for this function: reversing a list twice returns the original list, and reversing a singleton list has no effect.

```
module TestReverseList where

import Test.GenCheck.System.SimpleCheck
```

The properties that the revList function must satisfy:

- propRevUnitEq - the reverse of a singleton list is the original
- propRevRevEq - reversing the reverse is the original list (idempotency)

Note that the first is a property of individual values and the second is a property of lists, so different test suites and test case generators are required.

```
propRevUnitEq :: (Eq a) ⇒ Property a
propRevUnitEq x = (revList [x]) == [x]

propRevRevEq :: (Eq a) ⇒ Property [a]
propRevRevEq xs = (revList.revList) xs == xs
```

The implementation to test.

```
revList :: [a] → [a]
revList xs  = rev xs []
  where rev [] xs' = xs'
        rev (y:ys) xs' = rev ys (y:xs')
```

To evaluate whether the properties are satisfied, we need to generate a collection of test cases, evaluate them and report the results of the test. To do this, we pick one of the test programs provided by GenCheck, and the best place to start is the friendly sounding module SimpleCheck. There are a number of test programs here that provide variations on how test cases are generated and results are reported. The name of the test program consists of three parts:

testsuite Describes the kind of test suite generated: std, deep, base

reporting Defines whether the first failure (Check), all failures (Test) or all results (Report) are printed to the screen

arguments functions ending in Args have additional parameters; those not ending in Args provide default values for these parameters and default test case generators (so must be instances of Testable, more on this class coming up soon).

The first property takes a value, puts it in a list, and tests that reversing the list doesn't change the value. This value could be of any Haskell type, but we have to pick one to actually run the tests, so we'll start with Char. Char is a "base" type (as opposed to a structure) so all of the elements of this type are the same size. SimpleCheck's baseTest, baseCheck, and baseReport are all appropriate as they don't distinguish test elements by size. We'll pick baseTest so that all of the test cases are evaluated, but only the failure cases are reported. Char is an instance of the Testable class, so we don't need to use the parameterized versions (baseTestArgs, etc.) and will settle for the default settings for now.

Note that the GenCheck test programs generally request the desired number of test cases as input, but then only uses that as a guide in developing the test suite. The actual number of test cases will vary according to the number of possible test cases in the test domain.

```
testUnitEq_1 = baseTest (propRevUnitEq :: Property Char) 100
```

The next property requires a list of values to reverse, so baseTest is no longer appropriate as it is only for scalar values. The test suite must have lists of a variety of different lengths; in GenCheck parlance the size of a structure is referred to as it's "rank". stdTest and deepTest both generate test suites for structures, with deepTest including much higher ranked test cases, but fewer at each rank. For lists, there is only one list structure for each rank, so we'll use deepTest (deepReport and deepCheck would produce the same test cases).

Setting the property to a list of Ints seems a good starting point, but in order to use the SimpleCheck test functions we need an instance of Testable. Is —[Int]— an instance of Testable or do we need to do something? Lists are instances of the Structure class, which provides the method to populate a structure with elements from another generator, and also an instance of Testable, so there are standard generators available. A default instance of Testable is provided for lists of any Testable type, and Ints are Testable, so we don't have to worry about how lists are generated - yet.

```
testRevIdem_1 = deepTest (propRevRevEq :: Property [Int]) 100
```

The tests are passing, but somewhat unsatisfying that we can't see the test cases. So, we switch to the baseReport / deepReport functions that report all of the results, flagging any failed test cases with "FAILED: ". These are the same cases that were used for baseTest / deepTest, so we already know they will be successful, but using the Report versions will allow an evaluation of the quality of the test suite. The report dumps a lot of data onto the screen, so it made sense to verify the properties worked first with the Test functions so we didn't miss any of the FAILED flags.

```
testUnitEq_2  = baseReport (propRevUnitEq :: Property Char) 100
testRevIdem_2 = deepReport (propRevRevEq :: Property [Int]) 100
```

What happens when one or more test cases fail? Let's create a failing property, such as comparing a reversed list with the original. The very first test case with two elements in the list fails.

```
propRevEqNot :: (Eq a) ⇒ Property [a]
propRevEqNot xs =  revList xs == xs

testRevEqNot_1 =  deepTest (propRevEqNot :: Property [Int]) 100 -- should fail
```

We also see that *every* other case failed too - maybe we shouldn't print out all the failures! This is where using deepCheck comes in handy - it stops after the first failure case.

```
-- isn't implemented yet, so this is redundant
testRevEqNot_2 =  deepCheck (propRevEqNot :: Property [Int]) 100
```

The SimpleCheck module provides a useful collection of simple test programs that start with default test parameters and test case generators, but allow the tester to make a few simple choices to customize the testing if desired. The simpleTest, simpleReport and simpleCheck functions are the glue to connect the reporting, test execution and test suite options for a test program.

The examples in this module have properties over standard Haskell types, where the default generators were already available. How are test cases generated when the module contains a new type? How are the test cases chosen for inclusion in the test suite, and how can that be controlled? Surely there is a better way to display the results? These questions and more are answered in the next part of the tutorial.

2 Tutorial 2: Testing with New Data Types

The first tutorial covered testing properties over standard Haskell types. Default test case generators were already supplied by GenCheck as instances of the Testable class, but what about new data types? In this tutorial, we explain how to use the type constructor definition and GenCheck enumeration combinators to construct the standard test generators based on the type constructor definitions.

There are three steps in testing a module against a specification using the GenCheck framework:

1. define and code the properties that make up the specification
2. select the test system that provides the most appropriate test scheduling and reporting schema for the current phase of the development
3. define the test suite for those properties, including the test case generators for each.

In the previous section the details of the test system and test suite generation were hidden by the SimpleCheck API. In this section, we look at each of these steps in detail and discuss how to generate new data types and customize test suites for a particular set of properties.

For this tutorial, we'll look at a list zipper implementation, taken from the ListZipper package (1.2.0.2). Only snippets of the code will be provided as the

package is rather large (see tutorial/list_zipper/ListZipper.hs for the module source). The Zipper data type is

```
data Zipper a = Zip ![a] ![a] deriving (Eq,Show)
```

where the "current" element or "cursor" is the head of the second list. The specification for ListZipper includes empty, cursor, start, end, left, right, foldrz, foldlz, and many more functions. The example below shows how to test (some) of the specification for this module, with a focus on the generators for the new type.

2.1 Specifications and Properties

A GenCheck specification is the collection of properties that any implementation of the specification must satisfy. Properties are univariate Boolean valued Haskell functions, where the input value may be of an arbitrarily complicated type i.e. an uncurried predicate. The GenCheck test programs accept a single property and test it over the defined suite of test cases. Only one property is tested per call because different properties may have different input types or require different test suites [1]. The specification module would normally be separate from the implementation, but may include functions that test the properties using GenCheck.

2.1.1 List Zipper Properties

A small set of properties from the ListZipper package specification.

```
module PropListZip where

import Test.GenCheck (Property)
import ListZipper as Zipper
```

A ziplist is empty iff the cursor is both at the beginning and the end

```
propEmptyP :: Property (Zipper a)
propEmptyP z = emptyp z == beginp z && endp z
```

Folding an operation over the zipper with the cursor at the start should be identical to folding it over the unzipped list. Note that the zipper fold folds the cursor through the zipper, but always passes the zipper to the function. (cursor safety is supposed to be guaranteed by the foldz' functions)

```
propFoldrz :: (Eq b) ⇒ (a → b → b) → b → Property (Zipper a)
propFoldrz  f y0 zxs =
   let xs = Zipper.toList zxs
```

[1] a slight modification of the test programs would allow multiple properties to be tested, part of the intent of the GenCheck framework'

```
   in (foldr f y0 xs == foldrz f' y0 (start zxs))
   where f' zs y = f (cursor zs) y

propFoldlz :: (Eq b) => (b -> a -> b) -> b -> Property [a]
propFoldlz  f y0 xs =
   let zxs = Zipper.fromList xs
       fval = foldl f y0 xs
       zxs' = start zxs
   in (fval == foldlz f' y0 zxs') && (fval == foldlz' f' y0 zxs')
   where f' y zs = f y (cursor zs)
```

Replace is equivalent to deleting the current element and inserting a new one.

```
propInsDelRepl :: (Eq a) => Property (a, Zipper a)
propInsDelRepl (x, xs) = Zipper.insert x (Zipper.delete xs) == Zipper.replace x xs
```

2.2 Test Suites and Standard Generators

The module System.TestSuite contains functions that assemble test suites from one or more test case generators, including three that should seem familiar: stdSuite, deepSuite and baseSuite. These suite builders take a collection of "standard" generators and a set of instructions in the form of a list of rank, count pairs. The allocation of test cases to ranks through these instructions is called a "test strategy" and can be used independently of the type of data being generated, assuming that the standard generators are available for that type. The System.SimpleCheck module test suites are test cases stored in a Haskell Map (as in Data.Map) indexed by rank, called a MapRankSuite.

A GenCheck generator is any function from a rank (positive integer) to a list of values. Generators may be finite or infinite, may include duplicates of the same value, and may be missing values from the total population of the type. Generators can be manually coded or built from a GenCheck enumeration of the type using a sampling strategy called an "enumerative strategy". These strategies can be applied to any data type that can be enumerated, i.e. ordered, and indexed by a finite rank.

The so called standard generators are just four different sampling strategies that can be applied to an enumeration of the generated type. These are:

exhaustive Produces all of the values of the specified rank in a sequence

extreme Starts with the boundary cases, then the middle, then progressively splits the intervals

uniform Select the specified number elements at uniform intervals through the enumeration

random Use the random generator to pick an infinite list of values

These are not the only valuable sampling strategies that could be used, but these were selected as the "standard" generator strategies to be used by the GenCheck API. The standard generator set is the basis for the test suites used by SimpleCheck test programs, the TestSuite module, and most of the higher level API functions as the default generators. A type is an instance of Testable if a set of standard generators is available for it; these generators can be constructed for any type that has an Enumeration associated with it.

2.2.1 Enumerating New Data Types

A GenCheck enumeration is a way of ordering the values of a type. For base types such as Int and Char, this is called a base enumeration (BaseEnum) and is effectively the same as the Haskell Enum class; the GenCheck class is called EnumGC to highlight that it is an extension of Enum. For structure types, the enumeration is first partitioned into finite subsets by the number of elements in the structure (called the rank), and then the subset is ordered and indexed. Structure types that are enumerated are instances of the Enumerated class.

Enumerations for standard Haskell data types such as list and tuples are provided. For new data types, enumerations can be easily made using the enumeration combinators —eConst, eNode, eSum, eProd, etc.— in the Generator.Enumeration module. These combinators simply mirror the structure of the type constructors, but produce the required enumerations.

2.2.2 List Zipper Enumeration and Generators

The GenCheck class Structure and Enumeration instances for Zipper. It is typical of what would be mechanically generated for the type, but has been converted to literate Haskell and documented. There is no special significance to the $_GC$ suffix in the name, that is just a convention so the GenCheck instances module is easily found.

```
{-# LANGUAGE FlexibleInstances,FlexibleContexts  #-}
module ListZipper_GC where

import Control.Monad (liftM2)

import Test.GenCheck
import Test.GenCheck.Generator.Enumeration

--import Test.GenCheck.Generator.Generator (Testable(..))
--import Test.GenCheck.Generator.StructureGens()
--import Test.GenCheck.Generator.Substitution (Structure(..))

import ListZipper
```

The Structure class has a single method called substitute. It replaces the elements in the structure with elements from a list and returns a newly populated instance of the structure and the remaining elements; if there are insufficient

elements in the list to populate the structure Nothing is returned along with the original list of elements.

A default substitution method can be mechanically derived for algebraic data types, by interpreting the type as a sum of products, and using substitute methods for any substructures. Alternate substitutions could be provided instead.

```
instance Structure Zipper where
  substitute = substZipper

substZipper :: Zipper a → [b] → (Maybe (Zipper b), [b])
substZipper (Zip x1 x2) ys =
  let (mx1, ys')  = substitute x1 ys
      (mx2, ys'') = substitute x2 ys'
      zip' = liftM2 Zip mx1 mx2
  in (zip', ys'')
```

The Zipper is just a product of two lists of like elements. The enumeration is built from the combinators eProd and eList (list gets a special combinator due to it's frequent appearance and special syntax).

The Enumeration consists of two functions that, given a rank, produce the number of structures and a selector function that generates a structure given an index. These functions are highly recursive, so are memoized using the Memoize Hackage (eMemoize just applies memoize to both the count and selector functions in the Enumeration records).

The selector function from the Enumeration creates instances of the structure. The Label type is used as a way to instantiate those structures and provide a label for the parametric type variables in the structure as a convenience.

```
instance Testable (Zipper Label) where
  stdTestGens = stdEnumGens

instance Enumerated Zipper where
  enumeration = eZipper

eZipper :: Enumeration Zipper Label
eZipper = eMemoize e
  where
    e = eProd Zip (eList A) (eList A)
```

Gordon says: This should be in Test.GenCheck.Generator.Enumeration ??? *(* ???)

```
eList :: a → Enumeration [] a
eList x = mkEnum c s
  where c _ = 1  -- there is only one list of any length
        s r k | k == 1 = take r $ repeat x
        s _ _ | otherwise = error "Selector error"
```

Generators are created from enumerations and enumerative strategies using Generator.enumGenerator for structure types and BaseGen.baseEnumGen for

base types. The standard generator set, and therefore the Testable instance, is built using stdEnumGens.

2.3 The Testing Process

Typically a GenCheck user would start with the default test suites and only reporting failures during the development cycle. Nearing completion, s/he would switch to the test programs that provide more control over the test suite generation, and report all of the test cases in the results to ensure good coverage.

The objective of GenCheck is to provide a testing framework that scales in scope from a very simple QuickCheck like interface, through progressively more thorough test suites and reporting, during the course of the development towards production. The highly modular GenCheck API allows an arbitrarily deep level of control, with a commensurate level of complexity, supporting customization and new development as required.

2.3.1 Testing the List Zipper Module

```
module Test_ListZipper where

import System.Random(mkStdGen)

import Test.GenCheck.System.SimpleCheck
           (stdTest, stdTestArgs, stdReport, stdReportArgs)

-- because we're doing more than just the usual stuff
import Test.GenCheck
import Test.GenCheck.Generator.BaseGens (genIntAll, genLowCharAll, genUpperCharRnd)

-- the specification, module, and GenCheck instances for the Zipper data type
import PropListZip
import ListZipper as Zip
import ListZipper_GC()

genZip_Char :: StandardGens (Zipper Char)
genZip_Char = substStdGenAll (stdTestGens :: StandardGens (Zipper Label))
                             genLowCharAll

genValZip_Char :: Rank → [(Char, Zipper Char)]
genValZip_Char r =
  let gc = (genUpperCharRnd (mkStdGen 657985498)) -- random value generator
      gz = genAll genZip_Char -- exhaustive zipper generator
  in zip (gc 1) (gz r) -- (Char, Zipper Char) generator with random Char, exhaustive zippers

testFoldrz_1, testFoldrz_2 :: Rank → Count → IO ()
testFoldrz_1 r n =
  let propFoldrz_str = propFoldrz (:) "" :: Property (Zipper Char)
```

```
      lbl = "Compare concatenating strings with right fold, up to rank " ++
(show r)
  in  stdTestArgs genZip_Char lbl r propFoldrz_str n
testFoldrz_2 r n =
  let propFoldrz_str = propFoldrz (:) "" :: Property (Zipper Char)
      lbl = "Compare concatenating strings with right fold, up to rank " ++
(show r)
  in  stdReportArgs genZip_Char lbl r propFoldrz_str n

testFoldlz_1, testFoldlz_2 :: Rank → Count → IO ()
testFoldlz_1 r n =
  let propFoldlz_sum = propFoldlz (+) 0 :: Property [Int]
      lbl = "Compare sum over integer list to left fold, up to rank " ++ (show r)
  in  stdTest propFoldlz_sum n
testFoldlz_2 r n =
  let propFoldlz_sum = propFoldlz (+) 0 :: Property [Int]
      lbl = "Compare sum over integer list to left fold, up to rank " ++ (show r)
  in  stdReport propFoldlz_sum n

testInsDelRepl_1 :: Rank → IO ()
testInsDelRepl_1 r =
  let lbl = "Compare insert/delete to replace, up to rank " ++ (show r)
      -- do up to maxr tests for zippers up to rank maxr
      instruct maxr = [(rk, toInteger maxr) | rk ← [1..maxr]]
      suite = genSuite genValZip_Char (instruct r)
  in  simpleReport lbl propInsDelRepl suite
```