Notes from 12/7

Administrative

I juggled some deadlines, and eliminated Assignment 12. See the updated schedule.

Module header

You'll probably need some of these libraries:

```
stack install aeson conduit conduit-extra
stack install resourcet transformers-base
stack install QuickCheck quickcheck-special
stack install tasty tasty-hunit tasty-quickcheck
```

And we'll also use overloaded strings in what follows.

```
{-# LANGUAGE OverloadedStrings #-}
module N20171207 where
```

These imports pertain to the rest of the file.

```
import Control.Exception (bracket)
import Control.Monad.Base (liftBase)
import Control.Monad.Trans.Resource (runResourceT)
import Data.Aeson ((.=), (.:))
import qualified Data.Aeson as Js
import qualified Data.ByteString.Lazy.Char8 as LBC
import Data.Conduit
import qualified Data.Conduit.Binary as CB
import qualified Data.Conduit.List as CL
import qualified Data.Conduit.Text as CT
import Data.Int (Int8)
import qualified Data.Text as T
import System.IO
import Test.QuickCheck (quickCheck)
import Test.QuickCheck.Special (Special(..))
import Test.Tasty
import Test.Tasty.HUnit
import Test.Tasty.QuickCheck
```

It's pretty commonplace to have two import statements from the same module, as we did with `Data.Aeson` above. One of them is qualified, so we refer to definitions with a prefix, like `Js.encode` and don't add too much to the current namespace. The other import is selective, it only imports a few specific defintitions to use unqualified. We're doing that for the Aeson operators `(.=)` and `(.:)`.

Test frameworks

Testing is an important concern for software projects. Most languages have a library to help with *unit* testing, which means testing a specific function by providing certain inputs and verifying its outputs or effects. But we'll start with another form of testing, which is native to Haskell and functional programming.

QuickCheck

QuickCheck is a library for property-based testing, an idea that was pioneered and matured between two functional languages, Erlang and Haskell. It has since been ported to other languages, including Python.

In property-based testing, rather than writing specific test cases, you try to encode general *properties* of your functions. Then, the testing tool generates bunches of specific cases on its own, and tests whether your property holds for each of them. If it finds a case where your property fails, it prints the details as a counterexample.

Any function that produces a boolean can be a property. We are expecting the boolean to be true, so if you want a property that's always false, just use `not`.

Arithmetic tests

Here's a simple example of a property: addition should be commutative. Write that as a function of any two integers, which produces a `Bool`:

```
prop_commutative_add :: Int -> Int -> Bool
prop_commutative_add x y =
  x+y == y+x
```

I could test that on particular integers by invoking it directly from GHCi:

```
ghci> prop_commutative_add 3 4
True
```

But the main purpose is that quickCheck will generate the lots of arguments for me. Here's how we do that:

```
ghci> quickCheck prop_commutative_add
+++ OK, passed 100 tests.
```

It generated a hundred random pairs of `Int` values, passed them to the function, and verified that it always returned `True`!

But what happens if a property fails on some input? Here's an example using rational numbers. We'd like it to be true that $\frac{x}{y} \cdot y = x$. Seems sensible, right?

```
prop_rational_div_mul1 :: Rational -> Rational -> Bool
prop_rational_div_mul1 x y =
  (x/y) * y == x
```

Here's a few manual test cases, and then we'll ask `quickCheck` to evaluate it:

```
ghci> prop_rational_div_mul1 3 4
True
ghci> prop_rational_div_mul1 18 94
True
ghci> prop_rational_div_mul1 10231 2984
True
ghci> quickCheck prop_rational_div_mul1
*** Failed! Exception: 'Ratio has zero denominator' (after 1 test):
0 % 1
0 % 1
```

QuickCheck found a problem right away! We might not have thought to test it, but our property actually *isn't* valid when the demoninator `y` is zero!

We should restate the property, ensuring as a *precondition* that $y \neq 0$. For that, we use the `(==>)` operator. The boolean expression on the left is the precondition. QuickCheck will generate a hundred cases that ensure the precondition is met, and then for each of them also test the expression on the right.

```
prop_rational_div_mul2 :: Rational -> Rational -> Property
prop_rational_div_mul2 x y =
  y /= 0  ==>  (x/y)*y == x
```

The other change that was needed is that due to the `(==>)` operator, this function produces a `Property` rather than a `Bool`.

```
ghci> quickCheck prop_rational_div_mul2
+++ OK, passed 100 tests.
```

Great, so the property holds for `Rational` values. What about `Double`? Those are floating-point approximations, so there may be some error that creeps in. The only change I need is in the type signature:

```
prop_rational_div_mul3 :: Double -> Double -> Property
prop_rational_div_mul3 x y =
  y /= 0  ==>  (x/y)*y == x
```

```
ghci> quickCheck prop_rational_div_mul3
*** Failed! Falsifiable (after 1 test and 2 shrinks):
0.9749639888248032
0.8276847531797582
```

QuickCheck is showing us the `x` and `y` it found for which the property fails. It would be more helpful if it also shows us the calculated value on the left side of the `(==)`. For that, we can switch to QuickCheck's operator `(===)` (three equal signs). This is an equality test, but if it fails then it also includes the values being compared in the output.

```
prop_rational_div_mul4 :: Double -> Double -> Property
prop_rational_div_mul4 x y =
  y /= 0  ==>  (x/y)*y === x
```

```
ghci> quickCheck prop_rational_div_mul4
*** Failed! Falsifiable (after 4 tests and 1 shrink):
1.8746163139401988
18.173499848191607
1.874616313940199 /= 1.8746163139401988
```

Now it's easier to see how close the two results are.

Enumeration tests

Types that implement the `Enum` typeclass have a correspondence to integers. You can generate lists of them with the range notation (`[1..5]` or `['a'..'z']`), and they implement `succ` (successor) and `pred` (predecessor).

```
ghci> succ 5
6
ghci> pred 6
5
ghci> succ 'A'
'B'
ghci> pred 'b'
'a'
```

One property we might like to verify is that `pred` and `succ` are inverses! Here is that property expressed on the `Integer` type:

```
prop_integer_pred_succ :: Integer -> Property
prop_integer_pred_succ x = pred (succ x) === x
```

```
ghci> quickCheck prop_integer_pred_succ
+++ OK, passed 100 tests.
```

That was easy! But is it true for other types that implement `Enum`? Here's a generic way to specify it:

```
prop_pred_succ :: (Eq a, Show a, Enum a) => a -> Property
prop_pred_succ x = pred (succ x) === x
```

The type variable `a` must implement `Enum` because we're using `pred`/`succ`, but it must also implement `Eq` so we can compare the results, and `Show` so that we can print the counterexamples. When we test such a generic property, we need to specify the precise signature, so QuickCheck knows what values to generate.

```
ghci> quickCheck (prop_pred_succ :: Integer -> Property)
+++ OK, passed 100 tests.
ghci> quickCheck (prop_pred_succ :: Char -> Property)
+++ OK, passed 100 tests.
ghci> quickCheck (prop_pred_succ :: Int -> Property)
+++ OK, passed 100 tests.
```

Looks pretty good. Unfortunately though, the property *doesn't* hold for every single member of `Char` and `Int`! These are *bounded* types – there's a maximum and minimum. So choosing the maximum makes `succ` fail:

```
ghci> maxBound :: Int
9223372036854775807
ghci> pred (succ (9223372036854775807 :: Int))
*** Exception: Prelude.Enum.succ{Int}: tried to take `succ' of maxBound
```

The default generators for types like `Int` and `Char` do not easily produce some special value like `maxBound` or `minBound`. If you choose smaller types, like 8-bit integers, you can eventually cause it to happen. (It's a one-in-255 chance each time, so your results may differ.)

```
ghci> quickCheck (prop_pred_succ :: Int8 -> Property)
+++ OK, passed 100 tests.
ghci> quickCheck (prop_pred_succ :: Int8 -> Property)
+++ OK, passed 100 tests.
ghci> quickCheck (prop_pred_succ :: Int8 -> Property)
+++ OK, passed 100 tests.
```

```
ghci> quickCheck (prop_pred_succ :: Int8 -> Property)
*** Failed! Exception: 'Enum.succ{Int8}: tried to take `succ' of maxBound' (after 31 tests):
127
Exception thrown while printing test case: 'Enum.succ{Int8}: tried to take `succ' of maxBound'
```

Someone implemented a workaround that would tell QuickCheck to generate 'special' values more often than chance would suggest, exactly because they tend to be edge cases. You just wrap the type with `Special`.

```
ghci> quickCheck (prop_pred_succ :: Special Int -> Property)
*** Failed! Exception: 'Prelude.Enum.succ{Int}: tried to take `succ' of maxBound' (after 35 test
Special {getSpecial = 9223372036854775807}
Exception thrown while printing test case: 'Prelude.Enum.succ{Int}: tried to take `succ' of maxB

ghci> quickCheck (prop_pred_succ :: Special Int -> Property)
+++ OK, passed 100 tests.
```

Sometimes all tests will still succeed, but it produces the failures much more often than the default generator for `Int`.

Encoding tests

Now let's move to testing a more realistic property. Often when you have data representations in a language, you need to export and import them – maybe it's for storing in a binary file format, maybe for transmitting to a web API, or a database. An important property of these representations is that encoded data can be decoded again, and you get the same result.

Here's a simple data type for representing a geographic coordinate:

```haskell
data GeoCoord = GeoCoord
  { latitude :: Double
  , longitude :: Double
  } deriving (Eq, Show)
```

We'd like to convert these coordinates into a Javascript notation (JSON) so we can send and receive them from an API. The 'aeson' package has lots of tools for that, and I imported it using a `Js` qualifier. Here's an example of encoding and decoding a number:

```
ghci> Js.encode pi
"3.141592653589793"
ghci> Js.decode "3.141592653589793" :: Maybe Double
Just 3.141592653589793
```

The type of decode is `FromJSON a => ByteString -> Maybe a`, so when we call it out of context, we need to specify the expected result type. It returns `Maybe` because, of course, decoding can fail if there's a syntax error or type error:

```
ghci> Js.decode "3.141592653589793" :: Maybe Int
Nothing
ghci> Js.decode "3:141592653589793" :: Maybe Double
Nothing
```

We can specify how a `GeoCoord` gets converted to JSON by implemented the type class `ToJSON`.

```
instance Js.ToJSON GeoCoord where
  toJSON coord =
    Js.object [ "lat" .= latitude coord
              , "lon" .= longitude coord
              ]
```

Here, we're saying that it's represented as an object, and we're using the keys `lat` and `lon`. Try it:

```
ghci> g1 = GeoCoord 38.121 (-57.88)
ghci> Js.encode g1
"{\"lat\":38.121,\"lon\":-57.88}"
```

The backslashes only appear because GHCi is trying to display the Javascript strings inside the Haskell string. If you print the representation, it looks cleaner:

```
ghci> LBC.putStrLn $ Js.encode g1
{"lat":38.121,"lon":-57.88}
```

(I had to use `LBC.putStrLn` because the result of `Js.encode` is a lazy bytestring rather than a typical Haskell string. See the qualified import at the top.)

We can specify how to decode a coordinate by implementing `FromJSON`. This uses `withObject` to require an object representation (rather than a number or list), and then it uses the aeson operator `(.:)` and the functor/applicative operators `(<$>)/(<*>)` to sequence the field values into the right slots of the `GeoCoord` constructor.

```
instance Js.FromJSON GeoCoord where
  parseJSON = Js.withObject "GeoCoord" $ \o ->
    GeoCoord <$> o .: "lat" <*> o .: "lon"
```

Here it is in action. The ordering of the fields in the JS object are irrelevant, it picks them up by name. But if one is misspelled, it won't match.

```
ghci> Js.decode "{\"lat\":31.7,\"lon\":-15.2}" :: Maybe GeoCoord
Just (GeoCoord {latitude = 31.7, longitude = -15.2})
ghci> Js.decode "{\"lon\":-15.2,\"lat\":31.7}" :: Maybe GeoCoord
Just (GeoCoord {latitude = 31.7, longitude = -15.2})
ghci> Js.decode "{\"lng\":-15.2,\"lat\":31.7}" :: Maybe GeoCoord
Nothing
```

Now we can write our "round-trip" property: that a coordinate survives being converted to JSON and back:

```
prop_geo_json_roundtrip :: GeoCoord -> Property
prop_geo_json_roundtrip coord =
  Js.decode (Js.encode coord) === Just coord
```

To run it, we need *one* more thing. We need to specify how to generate random coordinates. There are already generators for all the built-in types, so this is just a matter of mapping them into our data structure. Here, we take two 'arbitrary' `Double` values, and then apply the constructor. Again, we're using the functor/applicative operators.

```
instance Arbitrary GeoCoord where
  arbitrary = GeoCoord <$> arbitrary <*> arbitrary
```

There's a function in QuickCheck called `sample` that helps you test an instance of `Arbitrary`:

```
ghci> sample (arbitrary :: Gen Char)
'E'
'C'
'\216'
'*'
'o'
'O'
'J'
'h'
'>'
'S'
'K'
ghci> sample (arbitrary :: Gen GeoCoord)
GeoCoord {latitude = 0.0, longitude = 0.0}
GeoCoord {latitude = 0.7865489931241992, longitude = 1.4375435557173153}
```

```
GeoCoord {latitude = -3.47023894151138, longitude = -4.3709050687064535}
GeoCoord {latitude = 1.7478772773937006, longitude = 4.639094423953339}
GeoCoord {latitude = -1.2547019006984546, longitude = -20.315340131826773}
GeoCoord {latitude = -7.7783335238925595, longitude = -9.688305134324583}
GeoCoord {latitude = 0.48298084974375205, longitude = -2.167132054713652}
GeoCoord {latitude = -8.755126145399208, longitude = 12.885220306489924}
GeoCoord {latitude = 18.748583026919654, longitude = 32.41633975464188}
GeoCoord {latitude = 60.26562342374762, longitude = 39.07470650031688}
GeoCoord {latitude = -421.56426194127044, longitude = 10.081915397957822}
```

Those look like pretty good examples. It even generated the coordinate of "Null Island". Now to test our JSON encoding/decoding: ghci> quickCheck prop_geo_json_roundtrip +++ OK, passed 100 tests.

```
ghci> quickCheck prop_geo_json_roundtrip
+++ OK, passed 100 tests.
```

HUnit

This is a library for doing simple unit tests as assertions. The `Assertion` type is really just a synonym for `IO ()` (the same type as `main`), so test cases can do I/O as needed. There are operators like `@?=` to assert that two things are equal. The mnemonic is that the question mark indicates the result you're unsure about, and then you're comparing it to a known, expected result. But you can do it in either order by switching the question mark with the equals:

```
actual @?= expected
expected @=? actual
```

Here's a silly example that tests the fractional approximation of π:

```haskell
case_approximate_pi :: Assertion
case_approximate_pi = do
  22/7 @?= pi
```

Of course, they're not really equal:

```
ghci> case_approximate_pi
*** Exception: HUnitFailure "expected: 3.141592653589793\n but got: 3.142857142857143"
```

Unlike QuickCheck, we can use HUnit to ensure that specific cases are tried:

```haskell
case_geocoord_decode :: Assertion
case_geocoord_decode = do
  Js.decode "{\"lon\":0.15,\"lat\":0.17}"
    @?= Just (GeoCoord 0.17 0.15)
```

```
ghci> case_geocoord_decode
```

No news is good news.

Tasty

Finally, Tasty is a framework for running lots of test cases and managing the outputs and results. It can easily integrate tests written using HUnit and QuickCheck (and other styles).

Here is one way to use it, explicitly labeling and grouping some of the previous tests we wrote:

```
runTests =
  defaultMain $
    testGroup "My tests"
      [ testCase     "decode coord"   case_geocoord_decode
      , testCase     "approx pi"      case_approximate_pi
      , testProperty "json roundtrip" prop_geo_json_roundtrip
      ]
```

And here is the output:

```
ghci> runTests
My tests
  decode coord:   OK
  approx pi:      FAIL
    expected: 3.141592653589793
     but got: 3.142857142857143
  json roundtrip: OK
    +++ OK, passed 100 tests.

1 out of 3 tests failed (0.01s)
*** Exception: ExitFailure 1
```

Manually labeling and organizing all your tests into trees is painful. There are a few helpful choices for that. One that I discussed in class is `tasty-discover`. If you install that tool, then you can create a Haskell file containing only this comment:

```
{-# OPTIONS_GHC -F -pgmF tasty-discover -optF --tree-display #-}
```

and the compiler will run `tasty-discover` to search subdirectories for test functions that begin with `case_` (for HUnit) or `prop_` (for QuickCheck).

We'll look at another technique that doesn't require a separate source file in assignment 11.

Resource management

TODO

bracket, conduits

open file, read each line count # chars on that line print that # close file

```
countChars = do
  h <- openFile "N20171207.lhs" ReadMode
  putStrLn "Opened file"
  contents <- hGetContents h
  let lineLengths = map length $ lines contents
  mapM_ (print . (100 `div`)) lineLengths
  hClose h
  putStrLn "Closed file"

countChars2 = do
  withFile "N20171207.lhs" ReadMode $ \h -> do
    putStrLn "Opened file"
    contents <- hGetContents h
    let lineLengths = map length $ lines contents
    mapM_ print lineLengths
    putStrLn "(Presumably) Closing file"
  putStrLn "Now we're done using withFile"

countChars3 =
  bracket acquire release calculate
  where
    acquire = do
      h <- openFile "N20171207.lhs" ReadMode
      putStrLn "acquire: file was opened"
      return h
    release h = do
      hClose h
      putStrLn "release: file was closed"
    calculate h = do
      contents <- hGetContents h
      let lineLengths = map length $ lines contents
      mapM_ (print . (100 `div`)) lineLengths
```

Conduits

```
countChars4 :: IO ()
countChars4 = runResourceT $ runConduit
  $  CB.sourceFile "N20171207.lhs"
```

```
  .| CT.decode CT.utf8
  .| CT.lines
  .| CL.map T.length
  .| CL.mapM_ (liftBase . print)

main :: IO ()
main = return ()
```