/*

Haskell meets Java

CIS 194 Week 14 4 December 2014

This week, we will explore the relationship of Haskell to Java, and, in the process, translate code back and forth between the two languages. In so doing, we will explore the more imperative features of Haskell (you *can* actually mutate variables, after all) and the more functional features of Java (you *can* actually write a monad in Java).

Accordingly, this file is *both* a valid literate Haskell file and a valid Java file.

```
import Control.Monad ( when, liftM, guard )
import Data.IORef
import Data.List     ( stripPrefix )
import Data.Maybe    ( maybe, isJust )
import Text.Read     ( readMaybe )

*/
import java.util.*;
import java.util.function.*;
import java.util.stream.*;
public class Lec
{
/*
```

Printing

Comparing a program as basic as printing "Hello, world!" is not terribly enlightening. Instead, we'll start with a program that asks the user for a number and then prints out every number from 1 up to the provided number.

First in idiomatic Haskell:

```
printNums1 :: IO ()
printNums1 = do
  putStr "How high should I count? "
```

```
  answerStr <- getLine
  case readMaybe answerStr of
    Nothing  -> putStrLn "Invalid entry. Please type a number."
    Just num -> mapM_ print [1..num]
```

And now, in idiomatic Java:

```
*/
  public static void printNums1()
  {
    Scanner in = new Scanner(System.in);

    System.out.print("How high should I count? ");
    int answer = in.nextInt();
    for(int i = 1; i <= answer; i++)
    {
      System.out.println(i);
    }

    in.close();
  }
/*
```

But, we can do better than that with a little error checking:

```
*/

  public static void printNums2()
  {
    Scanner in = new Scanner(System.in);

    System.out.print("How high should I count? ");
    if(in.hasNextInt())
    {
      int answer = in.nextInt();
      for(int i = 1; i <= answer; i++)
      {
        System.out.println(i);
      }
    }
    else
    {
      in.nextLine();
      System.out.println("Invalid entry. Please type a number.");
    }
```

```
    in.close();
  }

/*
```

Note how easy it was to forget the error checking in Java! And, in truth, this doesn't really work. If we call `printNum2()` two times in succession, we'll see why. The problem is that `hasNextInt()` doesn't consume any input. So, we could either manually gobble up the input or use exceptions.

Instead of going down this road, though, let's make the Java look more like the Haskell. The first step is to use Java's `Optional` type (directly inspired by Haskell's `Maybe` and OCaml's `option`) to define a safe function to parse an integer:

```
*/

  // No typeclasses. We have to be specific to integers. :(
  public static Optional<Integer> readMaybe(String in)
  {
    try {
      return Optional.of(Integer.parseInt(in));
    } catch(NumberFormatException e) {
      return Optional.empty();
    }
  }

/*
```

The `Integer.parseInt(String)` function throws the `NumberFormatException` when a parse fails. So, we just catch this exception and return `Optional.empty()` in that case. This should work nicely.

Unfortunately, Java doesn't provide a nice pattern-matching mechanism for its `Optional` type. But, it does provide this function:

```
void ifPresent(Consumer<? super T> consumer)
```

What's `Consumer`, you ask?

```
@FunctionalInterface
public interface Consumer<T>
{
  void accept(T t);
  // ...
}
```

`Consumer` is one of Java 8's new *functional interfaces*. A functional interface is an interface with (roughly) one method. (There are various arcane rules around methods whoch overlap those from `Object` and other silly exceptions to the "one method" rule.) A `Consumer<T>` is a function that consumes `T` and produces nothing in

response. In other words,

```
type Consumer a = a -> IO ()
ifPresent :: Optional a -> (a -> IO ()) -> IO ()
```

Haskell achievement unlocked!: You understand something in Haskell more easily than the equivalent Java!

Note how the Haskell equivalent to Java's `Consumer` involves the `IO` monad. That's because *everything* in Java must involve the `IO` monad – you can print, browse the Internet, and launch the missiles from anywhere in a Java program.

You'll see in the Java declaration for `ifPresent`, above, that there is a curious type `? super T`, whereas the Haskell equivalent seems to use just plain old `T` in that spot. (The "spot" is the second appearance of `a` in the Haskell type of `ifPresent`.) This difference has to do with Java's subtyping relation. Haskell has no subtyping, so Haskellers don't have to worry about this. And so, going forward, as Haskellers, we won't worry about it.

In our `printNums` example, though, we'll need to take action even if the optional value *isn't* present. `Optional` doesn't provide the right combinator, so we'll write it ourselves:

```
*/
  public static <A, B> B maybe ( B b
                               , Function<? super A, ? extends B> f
                               , Optional<A> optA )
  {
    if (optA.isPresent())
    {
      return f.apply(optA.get());
    }
    else
    {
      return b;
    }
  }
/*
```

Writing that function makes me cringe, because my compiler isn't checking that I've done a `isPresent()` check before using `get()`. But oh well. Happily, Haskell's standard library provides the same combinator, called `maybe`.

```
printNums3 :: IO ()
printNums3 = do
  putStr "How high should I count? "
  answerStr <- getLine
  maybe (putStrLn "Invalid entry. Please type a number.")
        (\num -> mapM_ (putStrLn . show) [1..num])
        (readMaybe answerStr)
```

And now, the Java version:

```
  public static void printNums3()
  {
    Scanner in = new Scanner(System.in);

    System.out.print("How high should I count? ");
    String answerStr = in.nextLine();
    maybe( System.out.println("Invalid entry. Please type a number.")
         , num -> {
           for(int i = 1; i <= num; i++)
           {
             System.out.println(i);
           }
         }
         , readMaybeInteger(answerStr) );

    in.close();
  }
```

Alas, that doesn't compile, because Java can't abstract over `void`! Let's look at the type variables used in `maybe`. The two pieces passed in both result in `void`, but the type variable `B` can't be `void` in Java. Furthermore, Java is an *eager* language, meaning that parameters to functions are evaluated before the function calls. Given this fact, the call to `System.out.println` in the `Nothing` case will happen *before* we check whether we have `Just` or `Nothing`. This is all wrong. We need to modify our combinator slightly to take two *functions*, so we can control their evaluation better. The function for the `Nothing` case takes no arguments and returns nothing, a use-case seemingly omitted from the `java.util.function` package, so we write it ourselves.

```
*/
  @FunctionalInterface
  public interface Action  // in Haskell, this would be `IO ()`
  {
    void go();
  }

  public static <A> void maybe2 ( Action nothingCase, Consumer<A> justCase
                                , Optional<A> optA )
  {
    if(optA.isPresent())
    {
      justCase.accept(optA.get());
    }
    else
    {
      nothingCase.go();
```

```
    }
  }
/*
```

Now, we're ready to finish the translation to Java:

```
*/
  public static void printNums3()
  {
    Scanner in = new Scanner(System.in);

    System.out.print("How high should I count? ");
    String answerStr = in.nextLine();
    maybe2( () -> System.out.println("Invalid entry. Please type a number.")
          , num -> {
            for (int i = 1; i <= num; i++)
            {
              System.out.println(i);
            }
          }
          , readMaybe(answerStr) );

    in.close();
  }
/*
```

What have we gained here? More compile-time checking. Through the use of `readMaybe`, we now explicitly model the possibility of failure. And, through the use of the carefully-engineered `maybe2` combinator, we are sure to safely access the contents of the `Optional`. By consistently programming this way, our Java code will have fewer errors.

Yet, there is still a small improvement: that `for` loop is really ugly in the middle of otherwise-functional code. Let's get rid of it.

```
*/
  public static void printNums4()
  {
    Scanner in = new Scanner(System.in);

    System.out.print("How high should I count? ");
    String answerStr = in.nextLine();
    maybe2( () -> System.out.println("Invalid entry. Please type a number.")
          , num -> IntStream.rangeClosed(1, num)
                            .forEach(System.out::println)
          , readMaybe(answerStr) );

    in.close();
```

```
  }
/*
```

Ah. That's much better. :)

Note the use of a *method reference* `System.out::println`, which passes a defined function as a functional interface parameter.

Mutable variables in Haskell

The previous section iterated through a program, making a Java program more like Haskell. Now, we'll take the opposite approach and make a Haskell program more like Java, exploring mutation in Haskell.

Let's remind ourselves of the interesting bits of the idiomatic Java program:

```
    System.out.print("How high should I count? ");
    int answer = in.nextInt();
    for(int i = 1; i <= answer; i++)
    {
      System.out.println(i);
    }
```

To do this in Haskell, `for`-loop and all, we'll need variable mutation. Haskell provides this facility in its `IORef`s, imported from `Data.IORef`. The key functions are

```
newIORef    :: a -> IO (IORef a)
readIORef   :: IORef a -> IO a
writeIORef  :: IORef a -> a -> IO ()
modifyIORef :: IORef a -> (a -> a) -> IO ()  -- more efficient than read+write
```

An `IORef` is just a mutable cell that can be read from and written to from within the `IO` monad. Using a monad is necessary because, for example, `readIORef` called on the same `IORef` may return different results at different times, depending on what's in the mutable cell.

We can write a `for`-loop in Haskell, but it's a little clunky. Let's see a cleaner solution first, in terms of `while`, which is much simpler. First, of course, we need to write `while`!

```
while :: IO Bool    -- the condition
      -> IO ()      -- the body
      -> IO ()
while cond body = do
  b <- cond
  when b $ do
    body
    while cond body
```

Then, writing the imperative Haskell version of `printNums` isn't too hard. The monads make it a little clunky

though.

```
printNumsImp1 :: IO ()
printNumsImp1 = do
  putStr "How high should I count? "
  answer <- readLn     -- no error checking, just like Java
  i <- newIORef 1
  while (do ival <- readIORef i
            return (ival <= answer)) $ do
    ival <- readIORef i
    print ival
    modifyIORef i (+1)
```

Can we get rid of the clunkiness? Not without a lot of effort. The problem is that we really *should* know exactly when, in our program, we are reading the value of a mutable cell. For example, do you know what this will print in Java?

```
int i = 5;
int j = i++ + i++;
System.out.println(j+i);
```

The difficulty here is that `i` is mutable and yet being read multiple times. The result is that Java is hard to reason about! In Haskell, similar code would have to be much more explicit.

Now, on to `for`. `for` is more challenging because it brings a new variable into scope. The only real way to do this in Haskell is with a lambda:

```
for :: a                          -- initial value
    -> (IORef a -> ( IO Bool      -- condition
                   , IO ()        -- update
                   , IO () ))     -- body
    -> IO ()
for initial mk_stuff = do
  ref <- newIORef initial
  let (cond, update, body) = mk_stuff ref
  while cond $ do
    body
    update

printNumsImp2 :: IO ()
printNumsImp2 = do
  putStr "How high should I count? "
  answer <- readLn
  for 1 $ \i -> ((<= answer) `liftM` readIORef i, modifyIORef i (+1),
    readIORef i >>= print)
```

Icky, but imperative.

Monads in Java

Can we really write monads in Java? Sadly, no. Let's look closely at the definition of the monad typeclass:

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
```

What's interesting for us now is that `m` is *not* a type, but a type constructor: it needs to be applied to some *other* type to be a proper type. For example, `Maybe` and `IO` are monads, but never `Maybe Int` or `IO ()`.

To model anything like this in Java, we would need to have parameterized type variables in Java, which we don't.

Not all is lost, however. Although we can't write the `Monad` Java interface, we *can* use the concept of monads in Java programming. Indeed, the `Optional` class has `return` and `(>>=)` buried in its methods:

```
public final class Optional<T>
{
  public static <T> Optional<T> of(T value) { ... }
  public <U> Optional<U> flatMap(Function<? super T, Optional<U>> mapper)
    { ... }
  ...
}
```

I claim that `of` is really `return` and `flatMap` is really `(>>=)`.

`of` is a static method, meaning that it takes no implicit object as a parameter. Thus, its only parameter has type `T`. Its return, we can see, has type `Optional<T>`. In other words, its type is `T -> Optional<T>`, which looks awfully like `a -> m a`, when `a` is `T` and `m` is `Optional`. And, we can look at the operation of `return` vs. `of` and see that they're the same.

`flatMap` is a non-static method, meaning that it takes an extra, implicit parameter of type `Optional<T>` (the type of the class it's defined in). `flatMap`'s one explicit parameter has type `Function<? super T, Optional<U>>`. In pseudo-Haskell, that would be `T -> Optional<U>`. And the result type is `Optional<U>`. Putting this all together, we see that the full type of `flatMap` is `Optional<T> -> (T -> Optional<U>) -> Optional<U>`, just like `(>>=)`'s type! And the operation, "If a value is present, apply the provided `Optional`-bearing mapping function to it, return that result, otherwise return an empty `Optional`" matches up exactly. (Quote taken from the Java API documentation viewed **here**.)

Remember `stringFitsFormat`? That function is intended to recognize strings like "1a2aa" and "3aaa", where the string consists of pairs of a one-digit number followed by that many "a"s. Here is its implementation:

```
stringFitsFormat :: String -> Bool
stringFitsFormat = isJust . go
  where go :: String -> Maybe String
        go [] = Just ""
```

```
        go (digit:str) = do
          n <- readMaybe [digit]
          rest <- stripPrefix (replicate n 'a') str
          go rest
```

And now in Java:

```
*/
  public static Optional<String> stripPrefix(String prefix, String s)
  {
    if(s.startsWith(prefix))
    {
      return Optional.of(s.substring(prefix.length()));
    }
    else
    {
      return Optional.empty();
    }
  }

  // like Arrays.fill, but returns its argument!
  public static char[] fill(char[] a, char val)
  {
    Arrays.fill(a, val);
    return a;
  }

  // no local functions. :(
  private static Optional<String> stringFitsFormat_go(String s)
  {
    if(s.isEmpty())
    {
      return Optional.of("");
    }
    else
    {
      char digit = s.charAt(0);
      String str = s.substring(1);
      return readMaybe("" + digit).flatMap( n ->
             stripPrefix(new String(fill(new char[n], 'a')), str).flatMap( rest ->
             stringFitsFormat_go(rest)));
      // I miss `do` notation.
    }
  }

  public static boolean stringFitsFormat(String s)
```

```
  {
    return stringFitsFormat_go(s).isPresent();
  }
/*
```

So, we've seen that Java really does have the `Maybe` monad. But, it's actually not all that useful in Java because Java's exceptions mechanism has essentially the same purpose – allowing a computation to abort in the middle.

On the other hand, Haskell's list monad, which can be thought of as a non-determinism monad, *is* quite useful in Java. Java's new `Stream` interface is the right home for this monad, and that interface provides the same `of` and `flatMap` combinators that `Optional` does. We can do the exact same analysis that we did before to discover that `of` is really `return` and `flatMap` is really `(>>=)`. So, we'll now consider a power use of the non-determinism monad in solving a logic puzzle, taken from **Dinesman 1968**:

"Baker, Cooper, Fletcher, Miller, and Smith live on different floors of an apartment house that contains only five floors. Baker does not live on the top floor. Cooper does not live on the bottom floor. Fletcher does not live on either the top or the bottom floor. Miller lives on a higher floor than does Cooper. Smith does not live on a floor adjacent to Fletcher's. Fletcher does not live on a floor adjacent to Cooper's. Where does everyone live?"

Here is a solution in Haskell:

```
-- Is the given list composed of all distinct elements?
distinct :: Eq a => [a] -> Bool
distinct [] = True
distinct (x:xs) = all (x /=) xs && distinct xs

-- Are the two numbers provided separated by exactly 1?
adjacent :: (Num a, Eq a) => a -> a -> Bool
adjacent x y = abs (x - y) == 1

-- Solves the puzzle, giving the floors of
-- (Baker, Cooper, Fletcher, Miller, and Smith), in that order
multipleDwelling :: [(Int,Int,Int,Int,Int)]
multipleDwelling = do
  baker    <- [1 .. 5]
  cooper   <- [1 .. 5]
  fletcher <- [1 .. 5]
  miller   <- [1 .. 5]
  smith    <- [1 .. 5]
  guard (distinct [baker, cooper, fletcher, miller, smith])
  guard (baker /= 5)
  guard (cooper /= 1)
  guard (fletcher /= 5)
  guard (fletcher /= 1)
  guard (miller > cooper)
  guard (not (adjacent smith fletcher))
  guard (not (adjacent fletcher cooper))
```

```
  return (baker, cooper, fletcher, miller, smith)
```

Let's translate this to Java!

First, we'll need the `guard` combinator. Recall the type of `guard`:

```
guard :: MonadPlus m => Bool -> m ()
```

It takes a Boolean condition and does nothing (that is, returns `()`) if the condition is true but "fails" (in the way appropriate for the `MonadPlus` instance) if the condition is false.

Specializing to lists, here is a suitable implementation:

```
guard :: Bool -> [()]
guard True  = return ()   -- a one-element list
guard False = []          -- a zero-element list
```

Translating this to Java isn't actually so bad. We'll need the `()` type, of course.

```
*/

  private enum Unit { UNIT }   // It's sad I have to define this.

  private static Stream<Unit> guard(boolean b)
  {
    // recall that Java's ?: operator is just like Haskell's if/then/else
    return b ? Stream.of(Unit.UNIT) : Stream.empty();
  }

/*
```

The rest of the translation is rather straightforward. Java doesn't have Haskell's nice `[1..5]` syntax, so we write `range` to help us out. Unfortunately, the naive translation is far too slow to be usable in Java.

```
*/
  private static <A> boolean distinct(A[] as)
  {
    // the Haskell-y way is way too slow in Java
    Set<A> set = new HashSet<>(Arrays.asList(as));
    return set.size() == as.length;
  }

  private static boolean adjacent(Integer a, Integer b)
  {
    return (Math.abs(a - b) == 1);
  }
```

```
  // Alas, the simple version is too slow in Java. Not lazy enough, perhaps?
  /*
  private static Stream<Integer> range(int lo, int hi)
  {
    return Stream.iterate(lo, x -> x + 1);
  }

  public static int[][] multipleDwelling()
  {
    return
    range(1, 5)                          .flatMap( baker    ->
    range(1, 5)                          .flatMap( cooper   ->
    range(1, 5)                          .flatMap( fletcher ->
    range(1, 5)                          .flatMap( miller   ->
    range(1, 5)                          .flatMap( smith    ->
    guard(distinct(new Integer[] {baker, cooper, fletcher, miller, smith}))
                                         .flatMap( u1       ->
    guard(baker != 5)                    .flatMap( u2       ->
    guard(cooper != 1)                   .flatMap( u3       ->
    guard(fletcher != 5)                 .flatMap( u4       ->
    guard(fletcher != 1)                 .flatMap( u5       ->
    guard(miller > cooper)               .flatMap( u6       ->
    guard(!adjacent(smith, fletcher))    .flatMap( u7       ->
    guard(!adjacent(fletcher, cooper)) .flatMap( u8         ->
    Stream.of(new int[] { baker, cooper, fletcher, miller, smith }))))))))))))))
    .toArray(int[][]::new);
  }
  */
/*
```

Instead, we have to use this optimized version, which is careful to always choose distinct floors and tries to minimize backtracking.

```
*/
  private static Stream<Integer> chooseFloor(Integer... floors)
  {
    Set<Integer> avail = new HashSet<>(Arrays.asList(1, 2, 3, 4, 5));
    avail.removeAll(Arrays.asList(floors));
    return avail.stream();
  }

  // This more optimized version works just fine, though.
  public static int[][] multipleDwelling()
  {
    return
    chooseFloor()                                  .flatMap( fletcher ->
```

```
    guard(fletcher != 5)                          .flatMap( u1       ->
    guard(fletcher != 1)                          .flatMap( u2       ->
    chooseFloor(fletcher)                         .flatMap( cooper   ->
    guard(cooper != 1)                            .flatMap( u3       ->
    guard(!adjacent(fletcher, cooper))            .flatMap( u4       ->
    chooseFloor(fletcher, cooper)                 .flatMap( miller   ->
    guard(miller > cooper)                        .flatMap( u5       ->
    chooseFloor(fletcher, cooper, miller)         .flatMap( smith    ->
    guard(!adjacent(smith, fletcher))             .flatMap( u6       ->
    chooseFloor(fletcher, cooper, miller, smith) .flatMap( baker    ->
    guard(baker != 5)                             .flatMap( u7       ->
    Stream.of(new int[] { baker, cooper, fletcher, miller, smith }))))))))))))
    .toArray(int[][]::new);
  }
/*
```

Amazingly, that actually works! It takes rather long to compile though, making me think that programming monadically in Java is far from practical.

So, what is the upshot here? That you can take your knowledge from Haskell and directly apply it in Java. By thinking in terms of the non-determinism monad, you might write an easier solution to a Java problem than you could otherwise. Functional programming is a new way to *think*, and that thinking can translate into any programming language.

But if the language has lambdas, it will be easier!

And now, we end with more obligatory Java boilerplate:

```
*/

  // Why isn't this built-in again?
  public static String intMatToString(int[][] mat)
  {
    return Arrays.toString(Arrays.stream(mat)
                                 .map(Arrays::toString)
                                 .toArray(String[]::new));
  }

  public static void main(String[] args)
  {
    System.out.println(intMatToString(multipleDwelling()));
  }
}

/*

*/
```

```
Generated 2014-12-04 13:54:25.590718
```