Assignment 7

due at 23:59 on Wed Nov 8 (80 points)

For this assignment, we will translate some arithmetic expressions into *monadic* style. This will enable the calculations to be run in different monads to produce different effects.

Start by placing these *language extension codes* at the very top of your `a07.hs` file:

```
{-# LANGUAGE TypeSynonymInstances #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE FlexibleContexts #-}
```

The Haskell language and GHC compiler support many different language extensions that enable particular capabilities. These are not typically enabled by default because they have particular trade-offs that may not be appropriate for every program and development environment. But in most cases, there's no harm in turning on an extension if you need one.

Following those language extensions, add these import statements:

```
import Control.Monad.Writer
import Control.Monad.State
import Control.Monad.Identity
```

You should have all those installed already because they are part of the `mtl` package.

Now we'll define a class for monads that support integer arithmetic. That is, they'll have addition, multiplication, and division; but the answers will be returned within a monad `m`:

```
class Monad m => ArithMonad m where
  add  :: Integer -> Integer -> m Integer
  mult :: Integer -> Integer -> m Integer
  divi :: Integer -> Integer -> m Integer
```

Using these monadic arithmetic operations is a little more tedious than the typical definitions, because each operation needs to be sequenced using "bind" `(>>=)`. Let's see what it takes to implement a monadic function to compute $2x + 1$:

```
twiceAndOne x = do
  twiceX <- mult 2 x
  add twiceX 1
```

We used the do notation to sequence the two monad operations in the right order: `mult` and then add. Note that you can't compose them normally, as in `add (mult 2 x) 1`, because the subexpression `mult 2 x` doesn't produce an integer directly, but rather a monad-wrapped integer. So it requires a monadic bind. Instead of do notation, we could have used `(>>=)` directly:

```
twiceAndOneA x = mult 2 x >>= \y -> add 1 y
```

And you may recognize that the lambda function `\y -> add 1 y` can be eta-reduced to allow this shortcut:

```
twiceAndOneB x = mult 2 x >>= add 1
```

That's pretty clean and convenient. Recall that the bind operator `(>>=)` is sometimes pronounced "andThen", so this function reads pretty easily as "multiply 2 by x andThen add 1". When we have longer expressions to implement though, the do notation may be the clearest option.

Now it's your turn. Write a function called `grak` that takes two arguments `x` and `y`, and calculates $3x^2 + 2xy + 4y^2 - 5$. It must use the `mult` and `add` operators, so that its type is:

```
grak :: ArithMonad m => Integer -> Integer -> m Integer
```

No frills: the identity monad

Now we'd like to test our functions written using `ArithMonad`. It won't really work to call them directly in GHCI, because there are no concrete instances of `ArithMonad`:

```
ghci> twiceAndOne 13
<interactive>:2:1: error:
    • Ambiguous type variable ‘m0’ arising from a use of ‘print’
      prevents the constraint ‘(Show (m0 Integer))’ from being solved.
```

So let's provide the simplest instance. We imported `Control.Monad.Identity`. This defines a monad that adds *nothing* at all to the computation. Return just returns the value; bind just calls the function. But we need to define how to add/multiply/divide within the identity monad, like this:

```
instance ArithMonad Identity where
  add x y = return (x+y)
  mult x y = return (x*y)
  divi x y = return (div x y)
```

Now you can run any generic `ArithMonad` computation within the `Identity` monad like this:

```
ghci> runIdentity $ twiceAndOne 13
27
ghci> runIdentity $ twiceAndOne 82
165
```

So try it also with your `grak` function. Here's what I get:

```
ghci> runIdentity $ grak 3 8
326
ghci> runIdentity $ grak 6 4
215
```

Logging with the writer monad

Now we'll create a new instance of `ArithMonad` that keeps a log of each operation that is performed. Then we can print the log. Recall that the `Writer` monad uses a monoid (in this case, `String`) and an operation `tell` to append something to the log:

```
instance ArithMonad (Writer String) where
  add x y = do
    tell $ show x ++ " + " ++ show y ++ "\n"
    return (x+y)
  mult x y = do
    tell $ show x ++ " * " ++ show y ++ "\n"
    return (x*y)
  divi x y = do
    tell $ show x ++ " / " ++ show y ++ "\n"
    return (div x y)
```

The `return` portion is the same as for the `Identity` instance, but now we're using `tell` to assemble a log entry. We can run our previous computations in the writer monad like this:

```
ghci> runWriter $ twiceAndOne 82 :: (Integer, String)
(165,"2 * 82\n164 + 1\n")
```

We get the same result (165) but it's paired with a string that represents the log of all the calculations: `2*82` and then `164+1`. But there are two inconvenient things about this: first, we had to specify the type `:: (Integer, String)` to get it to work. Second, the format of the log is pretty awkward, with the embedded `\n` newline characters.

Here's a function that will streamline our usage of the writer monad for arithmetic:

```
runLog action = do
  let (result, log) = runWriter action
  putStr log
  return result
```

And its usage:

```
ghci> runLog $ twiceAndOne 82
2 * 82
164 + 1
165
```

Much more convenient! The two lines showing the arithmetic are output by the `putStr` from the log. The last line showing the result is just the answer returned by `runLog`.

Here's a log of the operations executed during a call to `grak`:

```
ghci> runLog $ grak 6 4
6 * 6
3 * 36
6 * 4
2 * 24
4 * 4
4 * 16
108 + 48
156 + 64
220 + -5
215
```

Your code should have the same final answer, although the sequencing of operations may differ slightly – there are several orderings that still produce correct answers.

Exponentiation, fast and slow

We have addition, multiplication, and division, but suppose we want to implement integer exponentiation (x^y). Instead of adding it to `ArithMonad` directly, we can just implement it using the existing operations.

The simplest way to think of exponentiation is just as iterated multiplication. So here's a *non-monadic* version of it:

```
slowExp _ 0 = 1
slowExp x y = x * slowExp x (y-1)
```

So if you trace that to calculate 2^5, you'll see:

```
slowExp 2 5 =
2 * slowExp 2 4 =
2 * 2 * slowExp 2 3 =
2 * 2 * 2 * slowExp 2 2 =
2 * 2 * 2 * 2 * slowExp 2 1 =
2 * 2 * 2 * 2 * 2 * slowExp 2 0 =
2 * 2 * 2 * 2 * 2 * 1 =
2 * 2 * 2 * 2 * 2 =
2 * 2 * 2 * 4 =
2 * 2 * 8 =
2 * 16 =
32
```

If we convert this recursive function into monadic style using `ArithMonad`, it looks like this:

```
slowExpM _ 0 = return 1
slowExpM x y = do
  yMinus1 <- add y (-1)
  recurse <- slowExpM x yMinus1
  mult x recurse
```

and then we can run it in either monad:

```
ghci> runIdentity $ slowExpM 2 5
32
ghci> runLog $ slowExpM 2 5
5 + -1
4 + -1
3 + -1
2 + -1
1 + -1
2 * 1
2 * 2
2 * 4
2 * 8
2 * 16
32
```

The log of operations should match what we calculated in our manual trace of the code.

The `slowExp` functions are *linear* in the size of the exponent, so large exponents will require quite a lot of multiplications. Fortunately, there's a much faster way to do exponentiation. Here's the algorithm in direct (non-monadic) style:

```
fastExp _ 0 = 1
fastExp x y
  | even y = fastExp (x*x) (div y 2)
  | otherwise = x * fastExp x (y-1)
```

And a trace of how it works:

```
fastExp 2 5 =
2 * fastExp 2 4 =
2 * fastExp 4 2 =
2 * fastExp 16 1 =
2 * 16 * fastExp 16 0 =
2 * 16 * 1 =
2 * 16 =
32
```

So whenever the exponent is even, we can bypass a lot of multiplications by squaring the base and dividing the exponent by two.

Now it's your turn. Convert the fast exponentiation algorithm into monadic style using `return` and `do` notation. Call it `fastExpM`. It should work like this:

```
ghci> runIdentity $ fastExpM 2 5
32
ghci> runLog $ fastExpM 2 5
5 + -1
2 * 2
4 / 2
4 * 4
2 / 2
1 + -1
16 * 1
2 * 16
32
ghci> runIdentity $ fastExpM 14 28
1234766956912479358262297818562 56
ghci> runLog $ fastExpM 14 28
14 * 14
28 / 2
196 * 196
14 / 2
7 + -1
38416 * 38416
6 / 2
3 + -1
```

```
1475789056 * 1475789056
2 / 2
1 + -1
2177953337809371136 * 1
1475789056 * 2177953337809371136
38416 * 3214199700417740936751087616
1234766956912479358262297818562 56
```

Counting with state monad

Instead of generating the complete log of operations like we did with `Writer`, we could gauge the efficiency of a numerical algorithm just by *counting* the number of operations. The `State` monad is a good choice for that.

```
instance ArithMonad (State Int) where
```

You should complete this instance. It looks a lot like the instance for `Writer`, except that instead of the `tell` and `show` stuff before the `return`, just simply do `modify succ`.

The `modify` is a convenient operation on the state monad; here is its type specialized to an integer state, like we're using:

```
modify :: (Int -> Int) -> State Int ()
```

Then `succ` is just a function on integers (or any enumerable type) that returns the *next* one. So the overall effect of `modify succ` within the `State` monad is pretty much the same as something like `i++` to increment a counter in C++/Java.

Once you have the `State` instance of `ArithMonad` working, you should be able to do this:

```
runCount :: State Int a -> (a, Int)
runCount action = runState action 0
```

```
ghci> runCount $ slowExpM 2 5
(32,10)
ghci> runCount $ fastExpM 2 5
(32,8)
ghci> runCount $ slowExpM 14 28
(1234766956912479358262297818562 56,56)
ghci> runCount $ fastExpM 14 28
(1234766956912479358262297818562 56,14)
```

So `runCount` returns a pair. The first element is the answer, and the second is the number of operations it took to calculate the answer. You can really see the difference in efficiency between fast and slow exponentiation of 14^{28}.

Submit your `a07.hs` to this dropbox.