HW 21 **Prof. Caldwell**
Due: 22 November 2011 **COSC 3015**

1 The Monad Type Class

Here's the definitions of the type classes `Monad`.

```
class Monad m where
  return :: a -> ma
  (>>=) :: m a -> (a -> m b) -> mb
  (>>) :: m a -> m b -> mb
  m >> n  =  m >>= \_ -> n
```

The monad laws are given as follows:

Left Identity:	`(return x >>= f) = f x`
Right Identity:	`(m >>= return) = m`
Associativity:	`((m >>= f) >>= g) = (m >>= (\x -> f x >>= g))`

1.1 Maybe and List as instances of the Monad Type Class

```
instance Monad Maybe where
  return  = Just
  Nothing >>= f   = Nothing
  (Just x) >>= f  = f x

instance Monad [] where
  return x = [x]
  xs >>= f   = concat (map f xs)
```

1.2 Relating the instances to the Laws

Consider the list monad.

Theorem 1.1 (Left Identity for Lists) `(return x >>= f) = f x`

Proof:

$$\begin{array}{l}\mathtt{return\,x} >>= \mathtt{f} \\ \overset{\langle\langle \mathrm{def.}(>>=)\rangle\rangle}{=} \mathtt{concat(map\,f(return\,x))} \\ \overset{\langle\langle \mathrm{def.}\mathtt{return}\rangle\rangle}{=} \mathtt{concat(map\,f\,[x])} \\ \overset{\langle\langle \mathrm{def.}\mathtt{map}\rangle\rangle}{=} \mathtt{concat[f\,x]}\end{array}$$

Now, by the type of `(>>=)` we know `f :: a -> [b]`. This means `f x = [y]` for some `y`. This gives the following.

$$\mathtt{concat[f\,x]} = \mathtt{concat\,[[y]]} = [\mathtt{y}] = \mathtt{f\,x}$$

□

Exercise 1.1. Use the definition of `(>>=)` to show that `([] >>= f) = []`. Recall that concat can be defined primitively as follows:

```
concat [] = []
concat (xs:xss) = xs ++ (concat xss)
```

2 The Monoid Type Class

Monoids have an associative operator with a left and right identity. The type class `Monoid` is give as follows.

```
class Monoid m where
  mempty  :: m
  mappend :: m -> m -> m
```

The Monoid laws are as follows:

Left Identity: `mempty ‘mappend‘ x = x`
Right Identity: `x ‘mappend‘ mempty = x`
Associativity: `((x ‘mappend‘ y ) ‘mappend‘ z) = (x ‘mappend‘ (y ‘mappend‘ z))`

2.1 Maybe and List as instances of the Monoid Type Class

```
instance Monoid a => Monoid (Maybe a) where
  mempty                    = Nothing
  Nothing  ‘mappend‘ m      = m
  m ‘mappend‘ Nothing       = m
  Just m1 ‘mappend‘ Just m2 = Just (m1 ‘mappend‘ m2)

instance Monad [a] where
  mempty   = []
  mappend  = (++)
```

3 The MonadPlus Type Class

The type class MonadPlus essentially extends monads that have some monadic structure. is defined as follows:

```
class (Monad m) => MonadPlus m where
   mzero :: m a
   mplus :: m a -> m a -> m a
```

The laws relating `mzero` and `mplus` are the monoid laws.

Left Identity: `mzero ‘mplus‘ x = x`
Right Identity: `x ‘mplus‘ mzero = x`
Associativity: `((x ‘mplus‘ y ) ‘mplus‘ z) = (x ‘mplus‘ (y ‘mplus‘ z))`

But also (and this is not discussed in LYAHFGG) there must be a relationship between the MonadPlus operators and the operators in the underlying monad. It turns out that there is some disagreement in the Haskell community about what the correct laws relating the two should be. The following laws relating the `mzero` element of an instance of MonadPlus with the bind operator are universally accepted:

Left Zero : `(mzero >>= m) = mzero`
Right Zero : `(m >> mzero) = mzero`

The following laws are sometimes accepted:

Left Distribution: `(m ‘mplus‘ n) >>= k = (m >>= k) ‘mplus‘ (n >>= k)`
Left Catch: `((return a) ‘mplus‘ b) = return a`

The instantiation of lists as an instance of the MonadPlus type class satisfy the core laws, Left Zero, Right Zero, and Left Distribution. Maybe, IO and the state Monoid satisfy Left and Right Zero, and Left Catch.

3.1 Maybe and List as instances of the MonadPlus Type Class

```
instance MonadPlus Maybe where
  mzero                    = Nothing
  Nothing  ‘mplus‘ m       = m
  m ‘mplus‘ Nothing        = m
  Just m ‘mplus‘ Just n    = Just m

instance MonadPlus [] where
  mzero   = []
  mplus  = (++)
```

3.2 Something about the laws

To see that the left distributive law does not hold for Maybe consider the following Haskell interaction.

```
Prelude> :m + Control.Monad
Prelude Control.Monad>
Prelude Control.Monad> let k b =  if b then Nothing else Just True
Prelude Control.Monad> (Just True ‘mplus‘ Just False) >>= k
Nothing
Prelude Control.Monad> (Just True >>= k ) ‘mplus‘ (Just False >>=  k)
Just True
```

Exercise 3.1. Find an example that shows that lists do not satisfy the Left Catch rule.

4 List Comprehensions, do notation and guards

4.1 do notation

Recall that a sequence of bind operations can be rewritten using the do notion as follows:

```
m1 >>= \x1 ->                     do x1 <- m1
m2 >>= \x2 ->                        x2 <- m2
...                                   ...
mk >>= \xk ->                        xk <- mk
return (f x1 x2 ... xk)              return (f x1 x2 ... xk)
```

Code written using bind

4.2 List comprehensions

Consider the evaluation of the following list comprehension.

```
Prelude> [(x,y) | x <- [1..3], y <- "ab"]
[(1,'a'),(1,'b'),(2,'a'),(2,'b'),(3,'a'),(3,'b')]
```

This code can be rewritten using do notion.

```
Prelude> do {x <- [1..3]; y <- "ab"; return (x,y)}
[(1,'a'),(1,'b'),(2,'a'),(2,'b'),(3,'a'),(3,'b')]
```

Now we can eliminate the do notation.

```
Prelude> [1..3] >>= \x -> "ab" >>= \y -> return (x,y)
[(1,'a'),(1,'b'),(2,'a'),(2,'b'),(3,'a'),(3,'b')]
```

Now, using the definitions of bind (`>>=`) and `return` for the list monad we get the following.

```
Prelude> concat (map (\x -> (concat (map  (\y -> [(x,y)]) "ab")))  [1..3])
[(1,'a'),(1,'b'),(2,'a'),(2,'b'),(3,'a'),(3,'b')]
```

But what about a list comprehension that contains a guard like the following:

```
Prelude> [x | x <- [1..2], even x]
[2]
```

Guards can be implemented as follows:

```
guard :: MonadPlus m => Bool -> m ()
guard True  = return ()
guard False = mzero
```

Using the guard, we can rewrite the list comprehension using do notation as follows:

```
Prelude> do {x <- [1..2]; guard (even x); return x}
[2]
```

Exercise 4.1. Translate the do notation for the expression `do {x <- [1..2]; guard (even x); return x}` into the bind operator and then use the definition of `(>>=))`, `guard`, and `return` to symbolically evaluate the expression to show how the result `[2]` is arrived at. Show every step.