A Counter Monad

Action monads are often implemented by state transformers. Here is a counter monad that illustrates the idea.

The state is the counter value. A state transformer maps an old counter value to a new counter value and a return value.

```
data Counter a = C (Int -> (Int,a))

-- reset the counter
new :: Counter ()
new = C $ \_ -> (0,())

-- increment the counter:
inc :: Counter ()
inc = C $ \n -> (n+1,())
```


A Counter Monad

```
-- returning the current value of the counter:
get :: Counter Int
get = C $ \n -> (n,n)

-- return is nop, >>= is sequential execution
instance Monad Counter where
  return r = C $ \n -> (n,r)
  (>>=) (C f) g = C $ \n0 -> let (n1,r1) = f n0
                                 C g' = g r1
                             in g' n1
```

The “runtime system” for our counter monad may look like this:

```
run :: Counter a -> a
run (C f) = snd (f 0)
```


A Counter Monad: Example of Use

An example “program” using a counter:

```
myprog :: Counter Int
myprog = do new
            inc >> inc >> inc
            c1 <- get
            inc
            c2 <- get
            return (c1*c2)
```

Run the program:

```
run myprog
```

The result is 12.

Counter Monad with Exceptions

An exception is just an ordinary type, e.g.,

```
data Exn = Overflow | Other
```

It is the monad that treats exceptions in a special way. An exception is stored at the place of the return value:

```
data ECounter a = EC (Int -> (Int, Either a Exn))
```

We do this due to the following concerns:

- There is no other good value to return when an exception occurs.
- This does not affect normal return values if we implement the monad operations properly.

Counter Monad with Exceptions

Let's say inc will overflow if the counter exceeds 3:

```
inc :: ECounter ()
inc = EC f where f n | n <= 3    = (n+1, Left ())
                     | otherwise = (n, Right Overflow)
```

The monad operators:

```
instance Monad ECounter where
  return r = EC $ \n -> (n, Left r)
  (EC f) >>= g =
    EC $ \n0 -> let (n1,r1) = f n0
                    EC g' = either g throw r1
                in g' n1
```


Counter Monad with Exceptions

Where throw is defined as a command that throws an exception:

```
throw :: Exn -> ECounter a
throw e = EC $ \n -> (n, Right e)
```

To allow the user to catch and handle exceptions:

```
catch :: ECounter a -> (Exn->ECounter a) -> ECounter a
catch (EC f) h =
  EC $ \n0 -> let (n1,r1) = f n0
                  EC g' = either return h r1
              in g' n1
```


Counter Monad with Exceptions

The runtime system may look like this:

```
run :: ECounter a -> Either a Exn
run (EC f) = snd (f 0)
```

A program that throws an exception due to overflow:

```
errprog = inc >> errprog
```

A program that handles an exception:

```
witprog = errprog `catch` \_ -> return ()
```


(blank)