1. Consider the following Haskell datatypes:

```
data Term =
      NUM    Int
    | PLUS   Term  Term
    | TIMES  Term  Term
    | DIVIDE Term  Term

data Result =
      Ok     Int
    | Error  String
```

Define a function

```
interp :: Term -> Result
```

Make sure to raise an exception when dividing by zero.

Solution:

```
interp :: Term -> Result
interp (NUM n)      = Ok n
interp (PLUS t1 t2) = case interp t1 of
        Ok n    -> case interp t2 of
                Ok m    -> Ok (n + m)
                Error e -> Error e
        Error s -> Error s

interp (Times t1 t2) = case interp t1 of
        Ok n    -> case interp t2 of
                Ok m    -> Ok (n * m)
                Error e -> Error e
        Error s -> Error s
interp (DIVIDE t1 t2) = case interp t1 of
        Ok n    -> case interp t2 of
            Ok m    -> if m /= 0
                         then Ok (n / m)
                         else Error "Division by Zero!"
            Error e -> Error e
        Error s -> Error s
```

□

2. It is most likely that the function you defined in the previoius problem uses nested case statements for pattern matching. We can avoid this and refactor our code using a continuation. Define a function

```
check :: Result -> (Int -> Result) -> Result
```

Solution:

```
check :: Result -> (Int -> Result) -> Result
check (Ok n) f    = f n
check (Error s) f = Error s
```

□

3. define a new function

```
interp' :: Term -> Result
```

Use *check* (write it in infix notation) in your definition.

Solution:

```
interp' :: Term -> Result
interp' (NUM n) = Ok n
interp' (PLUS t1 t2) = (interp' t1) `check`
                       (\n -> (interp' t2) `check`
                       (\m -> Ok (n + m)))
interp' (TIMES t1 t2) = (interp' t1) `check`
                        (\n -> (interp' t2) `check`
                        (\m -> Ok (n * m)))
interp' (DIVIDE t1 t2) = (interp' t1) `check`
                         (\n -> (interp' t2) `check`
                         (\m -> if m /= 0
                                then Ok (n * m)
                                else Error "Division by Zero!"))
```

□

4. As it turns out, Haskell has a built in mechanism to simulate the imperative-looking style of programming used in *interp'*. It is called **do notation**. To use do notation however, we need to define a **monad** instance for our *Result* datatype.

Remember that a **monad** is a type class defined by two functions:

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

That is, in order to define an instance of a monad, two functions have to be defined. **return** and **bind** written as >>=.

Consider this modified datatype for our result type:

```
data RESULT r =
                OK     r
              | ERROR  String
```

Define a monad instance for this result type.

Solution:

```
instance Monad Result where
   return n           = OK n
   (>>=) (OK n) f     = f n
   (>>=) (ERROR s) f = ERROR s
```

□

5. A third function is often added to the list of functions a monad has to define.

```
(>>) :: m a -> m b -> m b
```

From the type of this function, we can infer that it ignores its first argument and only returns the computation done by its second argument. This function is helpful to understand do notation.

The do syntax provides a simple shorthand for chains of monadic operations. The essential translation of a do is captured in the following two rules:

```
do e1 ; e2       =        e1 >> e2
do p <- e1; e2   =        e1 >>= \p -> e2
```

Using do notation, define a function

```
interp'' :: Term -> RESULT Int
```

Solution:

```
interp'' :: Term -> RESULT Int
interp'' (NUM n)      = OK n
interp'' (PLUS t1 t2) = do
                n <- interp'' t1 ;
                m <- interp'' t2 ;
                return (n + m)
interp'' (TIMES t1 t2) = do
                n <- interp'' t1 ;
                m <- interp'' t2 ;
                return (n * m)
interp'' (DIVIDE t1 t2) = do
                n <- interp'' t1 ;
                m <- interp'' t2 ;
                if m /= 0
                   then return (n / m)
                   else ERROR "Division by Zero!"
```

□