Lecture 8

An expression evaluator

Our goal is to build a simple calculator. We will not worry about the parsing as that requires input output for which we are not ready yet. We will only build the expression evaluator. We start by defining a data type that captures the syntax of expressions. Our expressions are simple and do not have variables.

```
> data Expr = Const Double
>           | Plus  Expr Expr
>           | Minus Expr Expr
>           | Mul   Expr Expr
>           | Div   Expr Expr
```

Now the evaluator is easy.

```
> eval :: Expr -> Double
> eval (Const d)      = d
> eval (Plus  a b)  = eval a + eval b
> eval (Minus a b)  = eval a - eval b
> eval (Mul   a b)  = eval a * eval b
> eval (Div   a b)  = eval a / eval b
```

8.1 The `Either` data type

You might have noticed that that we do not handle the division by zero errors So we want to capture functions that evaluates to a value or returns an error.

The Haskell library exports a data type called `Either` which is useful in such a situation. For completeness, we give the definition of `Either`. You can use the standard type instead.

```
data Either a b = Left  a
                | Right b
```

The data type `Either` is used in haskell often is situation where a computation can go wrong like for example expression evaluation. It has two constructors `Left` and `Right`. By convention `Right` is used for the correct value and Left for the error message: A way to remeber this convention is to remember that "Right is always right".

We would want our evaluator to return either an error message or a double. For convenience, we capture this type as `Result`.

```
> type Result a = Either String a
```

The above expression is a *type alias*. This means that `Result a` is the same type as `Either String a`. As far as the compiler is concerned, they are both same. We have already seen a type aliasing in action namely `String` and `[Char]`.

We are now ready to define the new evaluator. We first give its type

```
> eval' :: Expr -> Result Double
```

The definition of `eval'` is similar to eval for constructors `Const`, `Plus`, `Minus` and `Mul` except that it has to ensure.

1. Ensure each of the sub expressions have not resulted a division by zero error.
2. If the previous condition is met has to wrap the result into a `Result` data type using the `Right` constructor.

Instead of explicit pattern matching we define a helper function for this calle `app` that simplify this.

```
> zeroDivision = Left "Division by zero"

> app op (Right a)  (Right b) = Right (op a b)
> app _       _          _     = zeroDivision
```

The constructor `Div` has to be handled seperately as it is the only operator that generates an error. Again we write a helper here.

```
> divOp (Right a) (Right b) | b == 0     = zeroDivision
>                           | otherwise  = Right (a / b)
> divOp    _         _                   = zeroDivision -- problem any way.
```

Now we are ready to define `eval'`.

```
> eval' (Const d)   = Right d
> eval' (Plus  a b) = app (+) (eval' a) (eval' b)
> eval' (Minus a b) = app (-) (eval' a) (eval' b)
> eval' (Mul   a b) = app (*) (eval' a) (eval' b)
> eval' (Div   a b) = divOp   (eval' a) (eval' b)
```

Let us load the code in the interpreter

```
$ ghci src/lectures/An-expression-evaluator.lhs
GHCi, version 7.0.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
[1 of 1] Compiling Main             ( src/lectures/Expression.lhs, interpreted )
Ok, modules loaded: Main.
*Main> let [zero,one,two,three,four,five] = map Const [0..5]
*Main> eval' (Div five zero)
Left "Division by zero"
*Main> eval' (Plus (Div five zero) two)
Left "Division by zero"
*Main> eval' (Plus (Div five two) two)
Right 4.5
*Main>
```