Lecture 10: Maybe and Monads
- Dealing with Exceptions with Maybe
- The Maybe Monad

Next Time:
- Monads as Contextual Computation
- List Monad and List Comprehensions

Reading: Hutton 12.3
Dealing with Errors in Evaluation

The messiest part of any language is always how to deal with fatal errors deep in the middle of some computation.

Let’s use the following language (Hutton p.164) for doing division to see why this is so and how Haskell deals with this...

data Expr = Val Int | Div Expr Expr

eval :: Expr -> Int
eval (Val n) = n
eval (Div x y) = (eval x) `div` (eval y)

Main> eval (Div (Val 8) (Val 4))  -- 8/4 => 2
2

-- (10/3)/2 => 1
Main> eval (Div (Div (Val 10) (Val 3)) (Val 2))
1
Dealing with Errors in Evaluation

But of course it is possible to get an error if we divide by zero:

```haskell
Main> eval (Div (Val 8) (Val 0)) -- 8/0 => error!
*** Exception: divide by zero
```

And this can happen anywhere:

```haskell
Main> eval (Div (Val 8) (Div (Val 6) (Div (Div (Val 12) (Val 0)) (Val 3)) ) )
*** Exception: divide by zero
```
Dealing with Errors in Evaluation

You are deep in the recursion and something goes wrong, what to do?

Exceptions are basically a way of freaking out and bailing on the whole damn thing:

eval(Div(Val 8)(Div(Val 6)(Div(Div(Val 12)(Val 0))(Div(Val 6)(Val 3)))))
  eval (Val 8)
  => 8
  eval (Div(Val 6)(Div(Div(Val 12)(Val 0))(Div(Val 6)(Val 3)))))
  eval (Val 6)
  => 6
  eval (Div(Div(Val 12)(Val 0))(Div(Val 6)(Val 3))))
  eval (Div(Val 12)(Val 0))
  => 12
  eval (Val 0)
  => 0
  eval 12 `div` 0
  exception “divide by 0”

Not very graceful....
Dealing with Errors in Evaluation: Exceptions

These are called exceptions, and most languages have some way of dealing with this, and it is always a complete pain in the neck!

```java
public class ExceptionDemo {

    public static void main (String[] args) {
        divideSafely(args);
    }

    private static void divideSafely(String[] array) {
        try {
            System.out.println(divideArray(array));
        } catch (ArrayIndexOutOfBoundsException e) {
            System.err.println("Usage: ExceptionDemo <num1> <num2>");
        } catch (NumberFormatException e) {
            System.err.println("Args must be integers");
        } catch (ArithmeticException e) {
            System.err.println("Cannot divide by zero");
        }
    }

    private static int divideInts(int i1, int i2) {
        return i1 / i2;
    }
}
```
Dealing with Errors in Evaluation: Exceptions

Haskell also has exceptions, and you can generate them yourself using the function:

```haskell
error :: String -> a
```

```haskell
eval :: Expr -> Int
eval (Val n) = n
eval (Div x y) = case eval x of
  0 -> error "Run Away!!!"
  n -> (eval x) `div` n
```

```
Main> eval (Div (Val 2) (Val 0))
*** Exception: Run Away!!!
CallStack (from HasCallStack):
  error, called at Main.hs:39:28 in main:Main
```

But we would like a graceful way of dealing with errors, where we are in control throughout the crisis...
Dealing with Errors in Evaluation: Maybe

Maybe to the rescue!

data Maybe a = Nothing | Just a

data Expr = Val Int | Div Expr Expr

eval :: Expr -> Maybe Int

eval (Val n) = Just n

eval (Div x y) = case eval x of
    Nothing -> Nothing
    Just n  -> case eval y of
        Nothing -> Nothing
        Just 0  -> Nothing
        Just m  -> Just (n `div` m)

Main> eval (Div (Val 12) (Val 4))
Just 3
Main> eval (Div (Val 12) (Val 0))
Nothing
Dealing with Errors in Evaluation: Maybe

But is this really any better? For each argument to a function you would need to write a nested case and check each time for Nothing:

```
 eval :: Expr -> Maybe Int
 eval (Val n) = Just n
 eval (Div x y) = case eval x of
   Nothing -> Nothing
   Just n  -> case eval y of
     Nothing -> Nothing
     Just 0  -> Nothing
     Just m  -> Just (n `div` m)
```

We call this "cascading cases" or a "staircase of cases"
Dealing with Errors in Evaluation: Maybe

What if you had to check five arguments to a function?

```haskell
addAll a b c d e =
    case eval a of
        Nothing -> Nothing
        Just a' ->
            case eval b of
                Nothing -> Nothing
                Just b' ->
                    case eval c of
                        Nothing -> Nothing
                        Just c' ->
                            case eval d of
                                Nothing -> Nothing
                                Just d' ->
                                    case eval e of
                                        Nothing -> Nothing
                                        Just e' -> Just (a'+b'+c'+d'+e')
```

Not sure this is any better!! Can we abstract away all this syntax? (Yes, of course...)
Dealing with Errors in Evaluation: Maybe

You can think of it this paradigm as wrapping everything in a `Maybe`, and during a computation of expressions $e_1, e_2, e_3$, etc., we pass along correct values using `Just` but jump off and pass `Nothing` back when we get to an error:

$$
\begin{align*}
  e_1 & \Rightarrow f_1 (\text{Just } e_1) \Rightarrow f_2 (\text{Just } e_2) \Rightarrow f_3 (\text{Just } e_3) \Rightarrow f_4 \text{ Nothing} \Rightarrow f_5 \text{ Nothing} \Rightarrow \ldots
\end{align*}
$$

... All good, no division by 0 ...

$e_3$ divides by 0

...Nothing good can come from this computation....
The Maybe Monad

This can be thought of as adding **context** around the referentially transparent "main line" of the computation, the context being a the **Maybe** data type containing the value you are computing.
The Maybe Monad

How to compute with Maybe values? We could write this explicitly:

```haskell
data Maybe a = Just a | Nothing

plus :: Maybe Integer -> Maybe Integer -> Maybe Integer
plus (Just x) (Just y) = Just (x+y)
plus _ _ = Nothing

divide :: Maybe Integer -> Maybe Integer -> Maybe Integer
divide (Just x) (Just y) | y /= 0 = Just (x `div` y)
 divide _ _ = Nothing
```

Main> plus (Just 4) (Just 5)
Just 9

Main> divide (Just 4) (Just 2)
Just 2

Main> divide (Just 4) (Just 0)
Nothing

But why evaluate both arguments?

If the first argument is Nothing, the whole computation return a Nothing, without evaluating the second argument.
The Maybe Monad

So come back to our "cascading cases" or a "staircase of cases":

```haskell
data Maybe a = Just a | Nothing

plus :: Maybe Integer -> Maybe Integer -> Maybe Integer
plus (Just x) (Just y) = Just (x+y)
plus _ _ = Nothing

divide :: Maybe Integer -> Maybe Integer -> Maybe Integer
divide (Just x) (Just y) | y /= 0 = Just (x `div` y)
                        | otherwise = Nothing
divide _ _ = Nothing

plus' :: Maybe Integer -> Maybe Integer -> Maybe Integer
plus' x y = case x of
            Nothing -> Nothing
            Just x' -> case y of
                        Nothing -> Nothing
                        Just y' -> Just (x'+y')

divide' :: Maybe Integer -> Maybe Integer -> Maybe Integer
divide' x y = case x of
            Nothing -> Nothing
            Just x' -> case y of
                        Nothing -> Nothing
                        Just 0  -> Nothing
                        Just y' -> Just (x' `div` y')
```

How to put all the details of Maybe into the background?
The Maybe Monad

How to make this paradigm—defining a data type to pass along relevant information about a computation—into a useful programming tool?

There are two issues:

A. How do we replace a bunch of tedious, almost-identical pieces of code with an abstraction?

B. How do we fit this abstraction into the "Haskell Ecosystem" via a type class?

```
data Maybe a = Just a | Nothing
```
The Maybe Monad

A. How do we replace a sequence of tedious, almost-identical pieces of code with an abstraction?

Here is the problem we want to abstract away:

- Unwrapping values by pattern matching
- Wrapping values back up into a `Just` or a `Nothing`

```haskell
data Maybe a = Just a | Nothing

eval :: Expr -> Maybe Int
eval (Val n) = Just n
eval (Div x y) = case eval x of
  Nothing -> Nothing
  Just n  -> case eval y of
    Nothing -> Nothing
    Just 0  -> Nothing
    Just m  -> Just (n `div` m)
```

---

**Diagram:**

```
Maybe Integer

Just n -> m

Maybe Integer

Just m

unwrap by pattern matching

m

Apply function

div

wrap result with a constructor

Just k
```

**Legend:**
- `Just` and `Nothing` are constructors.
- `div` applies the division function.
- Patterns are matched and unwrapped.

The Maybe Monad

So we have to figure out how to wrap and unwrap a data value held inside a `Maybe` without having to think about it.

We want to focus on the computation of the value in the **foreground**, and keep the details of wrapping and unwrapping in the **background**:

```
data Maybe a = Just a | Nothing
```

```
Maybe a

\[ e_1 \xrightarrow{\text{wrap}} \text{Just } e_1 \xrightarrow{\text{unwrap}} e_1 \xrightarrow{\text{wrap}} f_1 \xrightarrow{\text{unwrap}} \text{Just } e_2 \xrightarrow{\text{unwrap}} e_2 \xrightarrow{\text{wrap}} f_2 \xrightarrow{\text{unwrap}} \text{Nothing} \]

\[ e_2 \xrightarrow{\text{unwrap}} \text{Just } e_2 \xrightarrow{\text{unwrap}} e_2 \xrightarrow{\text{wrap}} g_1 \xrightarrow{\text{wrap}} \text{Just } e_2 \xrightarrow{\text{unwrap}} e_2 \xrightarrow{\text{wrap}} g_2 \xrightarrow{\text{unwrap}} \text{Nothing} \]

Apply function
\[ f_1 :: \text{Maybe } a \to \text{Maybe } b \]
Apply function
\[ f_2 :: \text{Maybe } b \to \text{Maybe } c \]
Apply function
\[ g_1 :: a \to \text{Maybe } b \]
Apply function
\[ g_2 :: b \to \text{Maybe } c \]
```

Put value in Maybe context.
Maybe Monad

But notice that every time we have seen the Maybe type used, it is used as a return type, because something may go wrong with the processing of the inputs. A good example is looking up a key in a map: if the key is not there, you return Nothing to indicate failure:

The Prelude provides a version of lookup that works on a list of key-value pairs:

```
Main> :t lookup
lookup :: Eq a => a -> [(a, b)] -> Maybe b
```

Data.Map provides a more efficient version based on balanced trees:

```
Main> import Data.Map
```

```
Data.Map> :t Data.Map.lookup
Data.Map.lookup :: Ord k => k -> Map k a -> Maybe a
```

**Punchline:** We want to be able to deal with functions that take "normal" values as arguments, but return a Maybe type. We will see that by currying, we really only need to account for functions of the following type:

```
a -> Maybe b
```
**Maybe Monad**

So we really only need (1) a basic function to wrap a value in a Maybe, and (2) a function to apply a function of type \((a \rightarrow \text{Maybe } b)\) to a Maybe value:

(1) The first is called "return":

\[
\text{return} :: a \rightarrow \text{Maybe } a \\
\text{return } x = \text{Just } x
\]

(2) The second is called "bind" and is given as an infix operator:

\[
(\gg=) :: \text{Maybe } a \rightarrow (a \rightarrow \text{Maybe } b) \rightarrow \text{Maybe } b \\
mx \gg= f = \text{case } mx \text{ of} \\
\quad \text{Nothing } \rightarrow \text{Nothing} \\
\quad \text{Just } x \rightarrow f x
\]

\[
\text{incm} :: \text{Integer } \rightarrow \text{Maybe } \text{Integer} \\
\text{incm } x = \text{Just } (x+1)
\]

\[
\text{Main>} (\text{Just } 5) \gg= \text{incm} \\
\text{Just } 6
\]
**Maybe Monad**

\[
\text{return} :: a \to \text{Maybe } a \\
\text{return } x = \text{Just } x
\]

\[
(\gg=) :: \text{Maybe } a \to (a \to \text{Maybe } b) \to \text{Maybe } b \\
\text{mx } \gg= f = \text{case mx of} \\
\quad \text{Nothing } \to \text{Nothing} \\
\quad \text{Just } x \to f x
\]

\[
\text{return } e_1 \quad \gg= \quad g_1 \quad \gg= \quad g_2
\]

- **Put value in Maybe context.**
- **Apply function**
  - \( g_1 :: a \to \text{Maybe } b \)
  - \( g_2 :: b \to \text{Maybe } c \)
The Monad Typeclass: A Clean Interface to Computing in Context

B. How do we fit this abstraction into the "Haskell Ecosystem" via a type class?

The Monad typeclass is defined in the Prelude as follows:

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

Any data type which is an instance of this class must provide implementations of these, so here is Maybe:

```haskell
instance Monad Maybe where
    return x = Just x
    mx >>= f = case mx of
        Nothing -> Nothing
        Just x -> f x
```

m here is a type constructor with one parameter, just as with Functors.

Now let's look at some code to see how this all works in practice.....