INF 212
ANALYSIS OF PROG. LANGS
FUNKTIONAL PROGRAMMING
-- MONADS

Instructors: Crista Lopes

Monads – what is the problem?

- The base problem: how to affect the world
- Problem is more prevalent in pure functional programming style
  - No side-effects
  - That's right: no side-effects!
- But you've all seen it too!

- Consequence: expressing computation declaratively

Example

```
def hypotenuse(x, y):
        return math.sqrt(math.pow(x, 2) + math.pow(y, 2))
```

Now we want to trace it, or affect the world in it:

```
def hypotenuse(x, y):
        h = math.sqrt(math.pow(x, 2) + math.pow(y, 2))
        print "x=" + str(x) + ";y=" + str(y) + ";h=" + str(h)
        return h
```

Example

```
def hypotenuse(x, y):
        h = math.sqrt(math.pow(x, 2) + math.pow(y, 2))
        return h, “x=“ + str(x) + “;y=“ + str(y) + “h=“ + str(h)
```

Signature was	float, float -> float
Signature now is	float, float -> float, string

```
> math.pow(hypotenuse(6, 16), 4);
```

pow is float, float -> float, not (float, string), float -> float

Let's invent monads!

```
def hypotenuse(x, y):
        return math.sqrt(math.pow(x, 2) + math.pow(y, 2))
```

Let's call functions (float, float) -> (float, string) Traceable_f_f

```
#((float, float) -> float) -> ((float, float) -> (float, string))
def makeTraceable_f_f(f):
    def traceable_f_f(x,y):
        h=f(x,y)
        return h, str(f) + " was called, result=" + str(h) + "\n"
    return traceable_f_f

# Now let's make one of these! And call it
>> aTraceableHypo = makeTraceable_f_f(hypotenuse)
>> aTraceableHypo(3,4)
(5.0, '<function hypotenuse at 0xfff42a74> was called, result=5.0\n')
```

Let's invent monads!

```
>>> math.pow(aTraceableHypo(3,4), 2)
TypeError: a float is required
```

It would be nice to trace math.pow too! Let's "lift" it
Let's call functions (float, string), float -> (float, string) Traceable_f_s_f

```
#(((float, float) -> (float, string)), float) ->
#          (((float, float) -> (float, string)) -> (float, string))
def makeTraceable_f_s_f(f, p):
    def traceable_f_s_f(t_f_f):
        r = f(t_f_f[0], p)
        return r, t_f_f[1] + str(f)+" was called, result="+str(r) + "\n"
    return traceable_f_s_f

# Now let's make one of these!
>>> aTraceablePowOf2=makeTraceable_f_s_f(math.pow, 2)
>>> aTraceablePowOf2(aTraceableHypo(3,4))
(25.0, '<function hypotenuse at 0xfff42a74> was called, result=5.0\n<built-in function pow> was
called, result=25.0\n')
```

Let's invent monads!

Still too tightly coupled, let's "bind" them externally instead:

```
>>> bind(aTraceableHypo(3,4), aTraceablePowOf2)
```

Exercise: write the function bind

```
# (t, (t->t')) -> t'
def bind(t, f):
    return f(t)
```

Voila! – our first monad!

What is a monad?

- It's a container

- An active container… it has behavior to:
  - Wrap itself around a type
  - Bind *functions* together

What is a monad?

- A type constructor, m
- A function that builds values of that type
  a -> m a (makeX, previously)
- A function that combines values of that type with computations that produce values of that type to produce a new computation for values of that type
  m a -> (a -> m b) -> m b (bind, previously)

Kinds of monads

- In Haskell
  - Identity
  - Maybe
  - Error
  - List
  - IO
  - State
  - Reader
  - Writer
  - Continuation

Identity monad

- Simple function application

  bind(x, f) = f(x)

Maybe monad

- Functions that return either a value of a certain type (Something) or no value at all (Nothing)
- Nothing values stop the computation; Something values get passed on

- A nice alternative to exceptions!

Maybe monad in C#

Necessary types

```
public interface Maybe<T>{}

public class Nothing<T> : Maybe<T>
{
    public override string ToString()
    {
        return "Nothing";
    }
}

public class Something<T> : Maybe<T>
{
    public T Value { get; private set; }
    public Something (T value)
    {
        Value = value;
    }
    public override string ToString()
    {
        return Value.ToString();
    }
}
```

Maybe monad in C#

- Next, we need 2 operations:
  - One to construct Maybe's
  - One to bind Maybe's to the rest of the computation

Maybe monad in C#

Maybe type constructor:

```
public static Maybe<T> ToMaybe<T>(this T value)
{
        return new Something<T>(value);
}
```

Example:

```
3.ToMaybe();
“hello”.ToMaybe();
```

Maybe monad in C#

Bind:

```
public static Maybe<B> Bind<A, B>(this Maybe<A> a,
                                  Func<A, Maybe<B>> func)
{
    var something = a as Something<A>;
    return something == null ?
                        new Nothing<B>() :
                        func(something.Value);
}
```

Let's use it

```
public static Maybe<int> Div(this int numerator, int denominator)
{
    return denominator == 0
                ? (Maybe<int>)new Nothing<int>()
                : new Something<int>(numerator/denominator);
}
```

```
15.Div(3);
>> 5

15.Div(0);
>> Nothing
```

But there's more

```
36.ToMaybe().Bind(n => Div(n, 3)).Bind(m => Div(m, 0)).Bind(p => Div(p, 9));
>> Nothing
```

Div(9) never happens