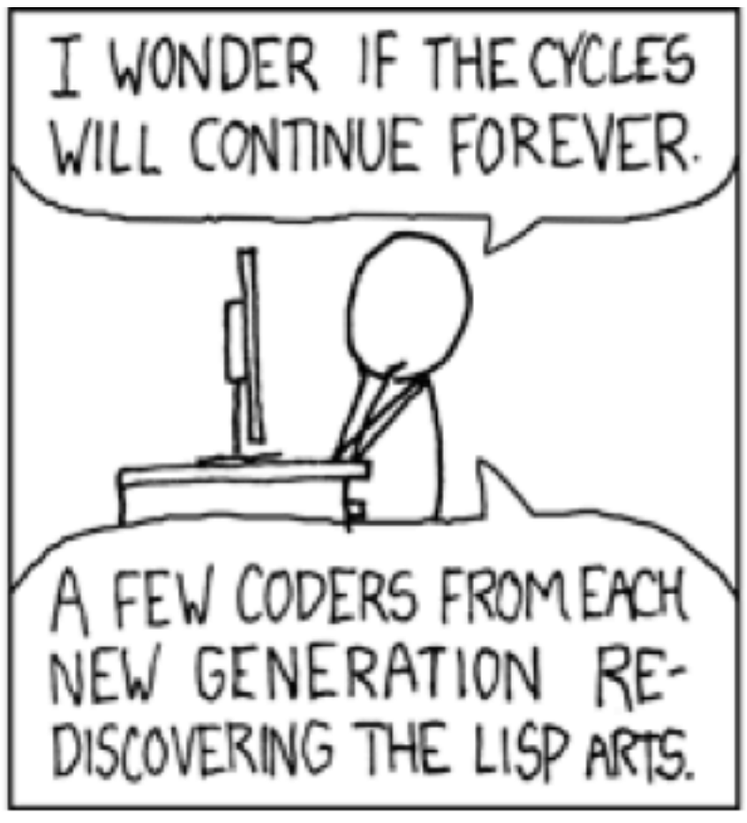
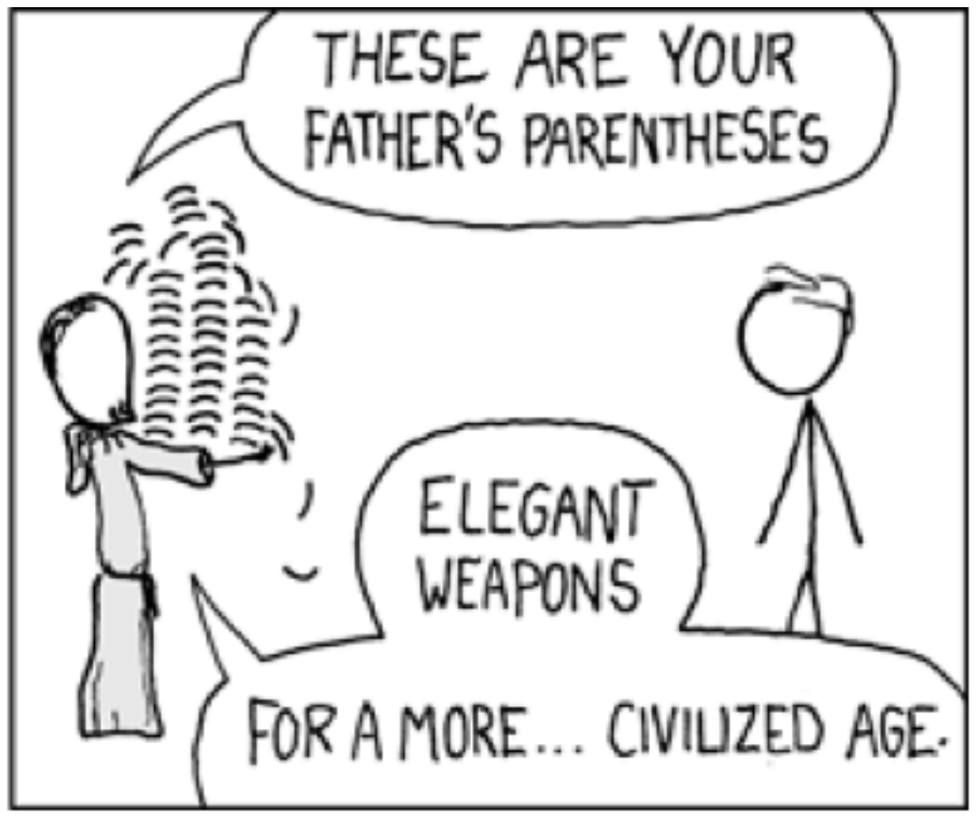

LISP IS OVER HALF A CENTURY OLD AND IT STILL HAS THIS PERFECT, TIMELESS AIR ABOUT IT.


I WONDER IF THE CYCLES WILL CONTINUE FOREVER.
A FEW CODERS FROM EACH NEW GENERATION RE-DISCOVERING THE LISP ARTS.


THESE ARE YOUR FATHER'S PARENTHESES
ELEGANT WEAPONS
FOR A MORE... CIVILIZED AGE.

Functional Programming on the JVM with Scala and Clojure

What they are and why you should care

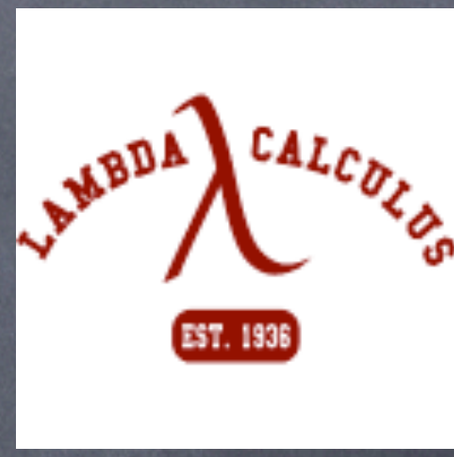
Cristiano Breuel
Advisory Software Engineer
(cbreuel@br.ibm.com)

Thadeu Russo
Advisory Software Engineer
(thadeurc@br.ibm.com)

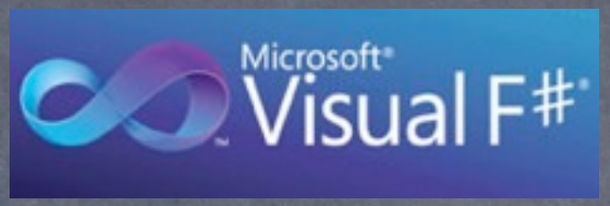
- $Y = \lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))$

What is functional programming about?

- Lambda calculus (a.k.a λ-calculus)
  - function definition
  - function application
  - recursion



- Referential Integrity
  - Avoids state and mutable data
- Higher-Order functions:
  - Functions can receive and return other functions
- Lazy Evaluation:
  - Evaluate an expression only when necessary



F16 $f(x)$ Σ =

	A	B	C	D	E	F
1						
2						
3	Date	Start time	End time	Pause	Sum	Comment
4	2007-05-07	9,25	10,25	0	1	Task 1
5	2007-05-07	10,75	12,50	0	1,75	Task 1
6	2007-05-07	18,00	19,00	0	1	Task 2
7	2007-05-08	9,25	10,25	0	1	Task 2
8	2007-05-08	14,50	15,50	0	1	Task 3
9	2007-05-08	8,75	9,25	0	0,5	Task 3
10	2007-05-14	21,75	22,25	0	0,5	Task 3
11	2007-05-14	22,50	23,00	0	0,5	Task 3
12	2007-05-15	11,75	12,75	0	1	Task 3
13						
14						
15						
16						
17						
18						

FP has been around since the 1950's

Why it is becoming more important now?

Why should I care?

Because of them...

... the multi-core processors

(also, to become a smarter developer!)

On the other side...

The JVM is cool!

- JVM -> Java Language
- Huge investment already on the JVM (libraries, middleware, performance, GC, remote communication, etc..)
- JVM executes byte-code

The question is:

Why not create new (functional) languages on top of the JVM?

(not Closure)

Clojure has...

- ... tight integration with the JVM and Java
- ... compilation for performance
- ... dynamic evaluation
- ... memory-efficient immutable collections
- ... lazy sequences
- ... primitives for concurrency (STM, agents)

Clojure is a LISP

```
(Lots of (Irritating
(Stupid (Parentheses)))))
```

a.k.a. List Processing

LISP

- Homoiconicity - code is data
- Macros
- Multiple dispatch (multi-methods)

Syntax

Operator

Operands (arguments)

```
(str "Hello" "World")
```

Expression delimiters

Syntax

["red" "green" "blue"]	Array
{:color "blue" :length 2}	Map
:color	Keyword
#(even? %)	Anonymous function

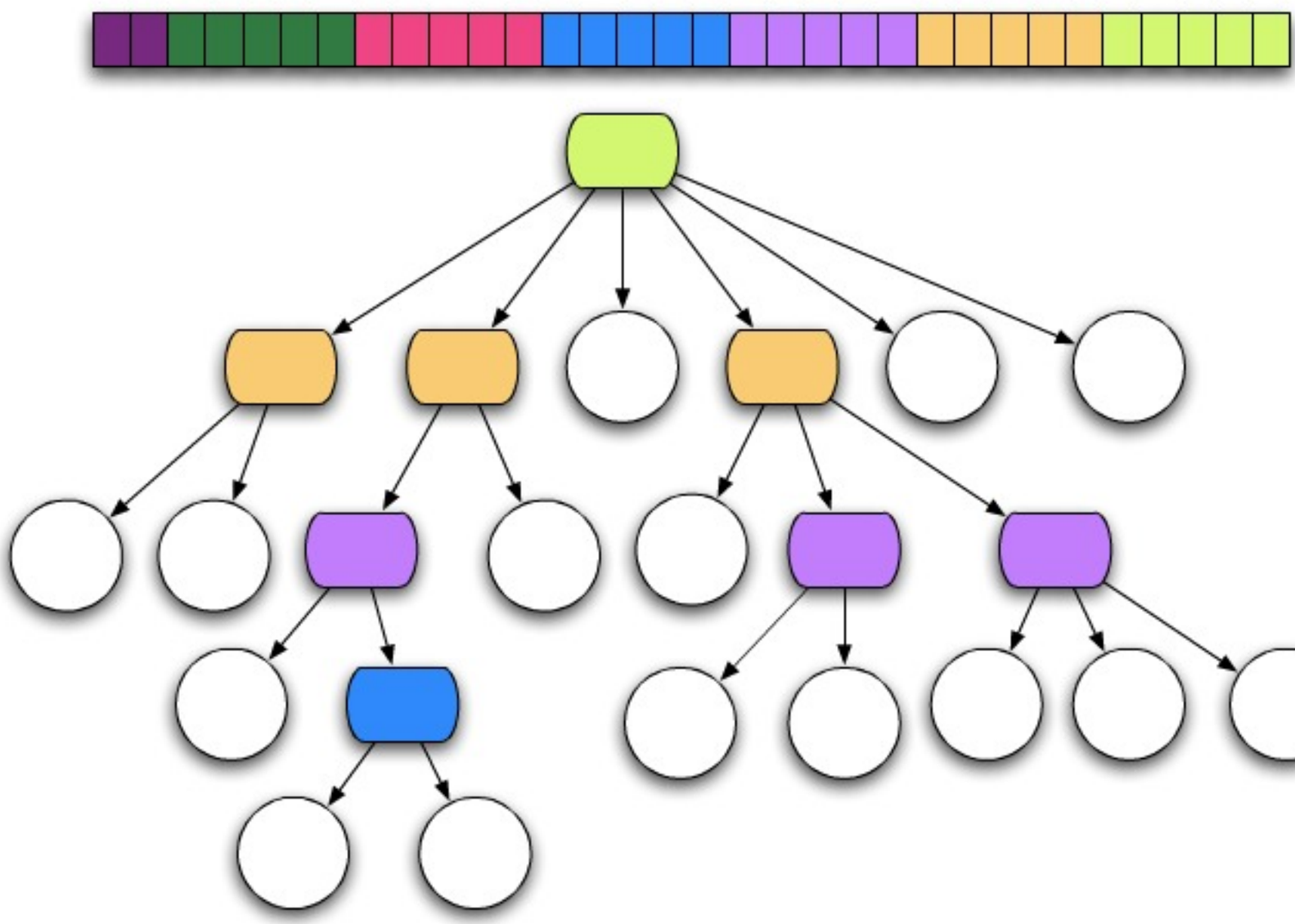
Higher-Order Functions

```
String[] dnaBases = {"AG", "GGATAC", "CTG"};
Arrays.sort(dnaBases, new Comparator<String>() {
    public int compare(String arg0, String arg1) {
        int l1 = arg0.length(), l2 = arg1.length();
        if (l1 > l2) {
            return 1;
        } else if (l1 < l2) {
            return -1;
        } else {
            return 0;
        }
    }
});
```

```
new Thread(new Runnable() {
    public void run() {
        System.out.println("Hello, Thread");
    }
});
```

```
int[] iarray = {1, 2, 3, 4};
int result = 0;
for (int i: iarray) {
    result += i;
}
```

Efficient Immutable Data Structures



Lazy Sequences

```
import java.util.HashMap;
import java.util.Map;


public class ColorMapper {

    private static String[] COLORS = {"white", "blue"};

    public static Map<String, String> mapLineColors(String[] values) {
        Map<String, String> retVal = new HashMap<String, String>();
        int i = 0;
        for (String value: values) {
            retVal.put(value, COLORS[i % COLORS.length]);
            i++;
        }
        return retVal;
    }

    public static void main(String[] args) {

        String[] values = {"Clojure", "Java", "Scala", "Visual Basic"};

        System.out.println(mapLineColors(values));
    }

}
```

Java Integration

```
(def now (new java.util.Date))

(def now (java.util.Date.))

(. now getTime)

(import '(javax.swing JFrame JLabel))

(doto (JFrame.)
  (. add (JLabel. "Hello, Clojure!"))
  (. pack)
  (. show))
```

- Fast
- Functional
- Expressive
- Statically typed (with type inference)
- Concurrent

- every value is an object
- vals / vars
- traits
- companion objects
- currying
- closures / high order functions
- pattern matching / case classes
- XML literals
- Lazy evaluation, and much more...

Syntax

- Java: int i = 0;
- Scala: var i: Int = 0
- Scala: var i = 0

More Syntax

name

return type
(can be inferred too)

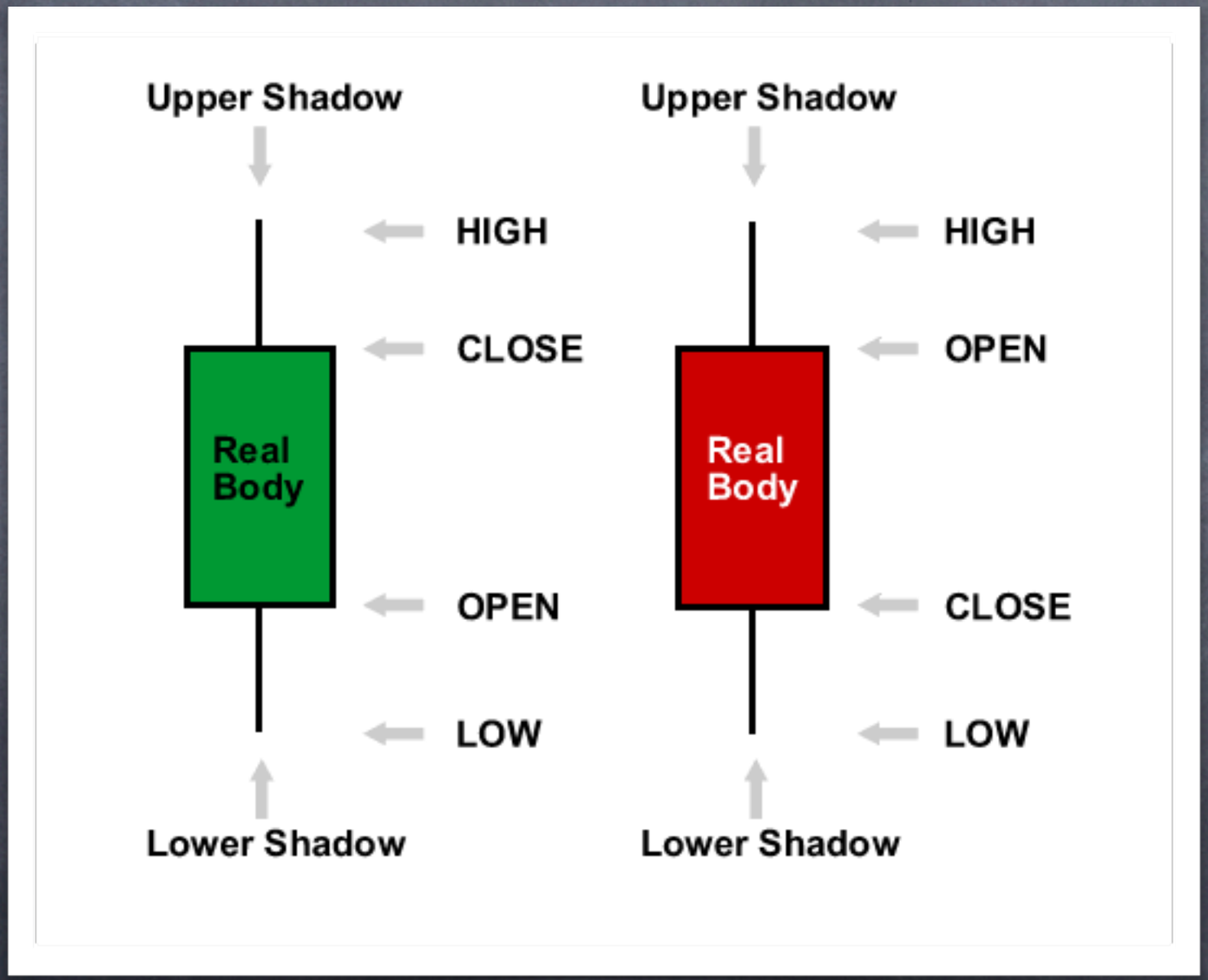
- `def mySum(a: Int, b: Int): Int = { a + b }`

arguments

function body

samples

OO + Functional Sample with Candlesticks



Trade and Candlestick Java code

- PS: I would love to put the code here, but it does not fit - #fail - go to eclipse >

Trade and Candlestick Scala code

```scala
class Trade(val stock: String, val price: Double, val quantity: Int){
    require(stock != null)
    require(price >= 0.0)
    require(quantity >= 0)

    def volume = price * quantity

    override def toString = "stock: %s, price: %.2f, quantity: %d".format(stock,price,quantity)
}

object Trade {
    def apply(stock: String = "", price: Double = 0.0, quantity: Int = 0) =
        new Trade(stock, price, quantity)
}
```

```scala
class Candlestick(val stock: String, val open: Double, val close: Double, val lower: Double,
                  val high: Double, val volume: Double){
    require(stock != null)
    require(open >= 0.0)
    require(close >= 0.0)
    require(lower >= 0.0)
    require(high >= 0.0)
    require(volume >= 0.0)

    def high_? = open < close

    def low_? = !high_?

    override def toString =
    "stock: %s open %.2f, close: %.2f, lower: %.2f, high: %.2f, volume: %.2f".format(stock, open, close, lower,
    high, volume)
}
```

Cont...

```
object Candlestick {
    private lazy val nullCandlestick = newCandlestick()

    def candlesticks(trades: List[Trade]): List[Candlestick] = {
        trades.groupBy(_.stock).map {
            case (_, list) => Candlestick(list)
        }.toList
    }

    def apply(trades: List[Trade]): Candlestick = {
        require(sameStock(trades))
        trades match {
            case Nil => newCandlestick()
            case list => {
                val high = trades.map(_.price).max
                val low = trades.map(_.price).min
                val volume = trades.map(_.volume).sum
                newCandlestick(list.head.stock,list.head.price,list.last.price,high,low,volume)
            }
        }
    }

    private def sameStock(trades: List[Trade]): Boolean = {
        trades.groupBy(_.stock).size == 1
    }

    private def newCandlestick(stock: String = "", open: Double = 0.0,close: Double = 0.0, high: Double = 0.0,
                               lower: Double = 0.0, volume: Double = 0.0): Candlestick = {
        new Candlestick(stock = stock, high = high, lower = lower, close = close, open = open, volume = volume)
    }
}
```

- Understanding and executing the code

To sum up...

We cannot choose our problems....

.... but we can choose our tools.

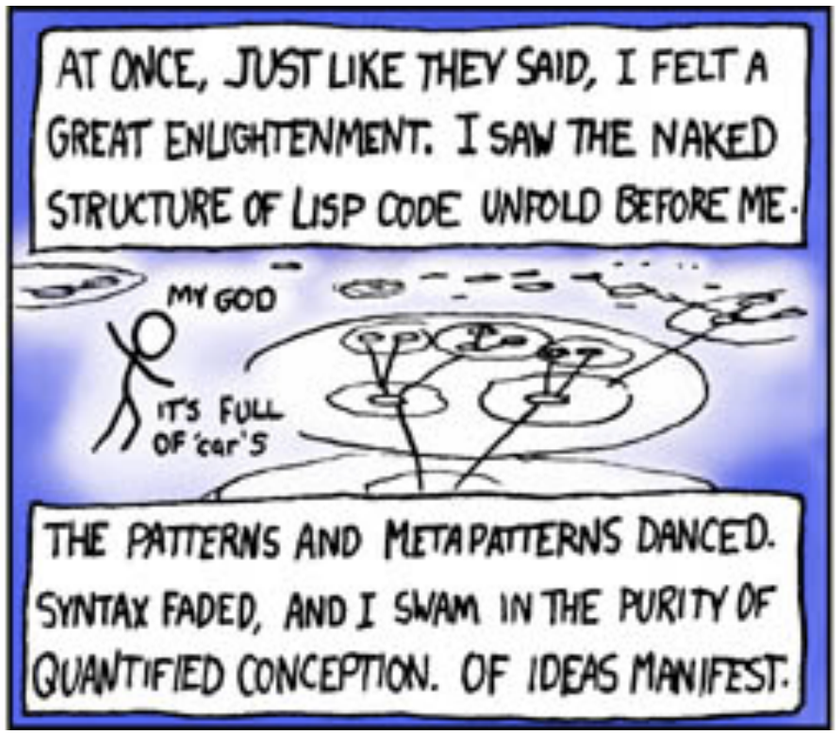
It is our responsibility to figure out the "right" tool.

Even if you can't use these languages in your day job yet, learning them will make you a better programmer!

References

- http://clojure.org/
- http://www.scala-lang.org/


LAST NIGHT I DRIFTED OFF WHILE READING A LISP BOOK.
HUH?
SUDDENLY, I WAS BATHED IN A SUFFUSION OF BLUE.


AT ONCE, JUST LIKE THEY SAID, I FELT A GREAT ENLIGHTENMENT. I SAW THE NAKED STRUCTURE OF LISP CODE UNFOLD BEFORE ME.
MY GOD
IT'S FULL OF 'car'S
THE PATTERNS AND METAPATTERNS DANCED. SYNTAX FADED, AND I SWAM IN THE PURITY OF QUANTIFIED CONCEPTION. OF IDEAS MANIFEST.


TRULY, THIS WAS THE LANGUAGE FROM WHICH THE GODS WROUGHT THE UNIVERSE.


NO, IT'S NOT.
IT'S NOT?
I MEAN, OSTENSIBLY, YES. HONESTLY, WE HACKED MOST OF IT TOGETHER WITH PERL.