

# Phosphorous, The Popular Lisp

Joseph F. Miklojcik III <jfm3@jfm3.org>

## Abstract

We present Phosphorous; a programming language that draws on the power and elegance of traditional Lisps such as Common Lisp and Scheme, yet which brings those languages into the 21st century by ruthless application of our “popular is better” philosophy into all possible areas of programming language design.

## Introduction

Modern and popular writing is not prefaced, overviewed, preambled, foreworded or otherwise introduced. This is just here to let the oldTimers know what is happening. It should also be noted here that, for similar reasons, there will be no conclusion, epilogue, summary, or review at the end<sup>1</sup>.

## Name

The name “Lisp” is far too tarnished for a popular, modern language. We determined that there were three characteristics a good modern programming language name should have. One, that it begin with the letter P. Two, that it name a gemstone. Three, that it be part of the name of a popular comedy group. We therefore named it “Phosphorus<sup>2</sup>”.

## Case

Any even somewhat traditional Lisp separates words in names using hyphens. Most traditional Lisps have very confusing rules regarding the case sensitivity of their names.

Modern and popular programming languages have settled on case sensitive names with words separated by changing case (“CamelCase”). Phosphorous follows suit. To underscore this fact, Phosphorous documentation replaces all uses of the hyphen in English with CamelCase<sup>3</sup>.

## Parenthesis

The most common complaint heard from new Lisp programmers must be “I can’t stand all these parenthesis.” At the same time, the success of XML in the marketplace shows that genericity of syntax is still a desirable feature. Clearly, the parenthesis must go, but can we do better than XML? The answer is emphatically “Yes!”

After careful study, we determined the problem stemmed from the fact that all parenthesis were alike. A programmer faced with the string

```
<table><tr><td><font=fixed><b>Foo</b>.  
<br/></font></tr></td></table>
```

can immediately see the obvious error, whereas in the string  
((((F00))))<sup>4</sup>

there are no such visual clues. In fact XML constructions such as `<b>` and `</b>` are just *named parenthesis*. We therefore experimented with similar named parenthesis in early versions of PhosphorousImpl (AvariciousArmoire<sup>5</sup>).

Early attempts at named parenthesis implementations were like XML minus one interior angle bracket from either side, with arbitrary matching names marking matching angle brackets. This had the visually pleasing effect of markers which appeared balanced, like parenthesis, yet looked enough like XML from a distance<sup>6</sup> to fool your programmerManager. The problem with this approach was that it was possible to write an Emacs macro to allow the programmer to simply use angle brackets like parenthesis, labeling them automatically and hiding the names from view. We found that to force programmers to use the named parenthesis syntax, we had to replace both left and right angle brackets with unmatchable characters, namely the vertical bar, thus foiling easy Elisp program analysis routines. To get the nice balance back, we decided that names had to be numbers that matched with the number that was the same but backwards. This is both pleasing to the eye, and really puts the screws to people who want to thwart the system using Emacs. To allow Phosphorus code to continue to have the look and feel of XML, angle brackets and any characters between them are ignored by the parser. Here are some example PhosphoExprs.

```
|123 321| <a null list>
```

```
<xmlfk:xml17r>|123 321|</xmlfk:xml17r>  
<!-- same, but dressed up to look like xml --!>
```

```
|44 a b c 44| <the list (a b c)>
```

<sup>1</sup> We do not expect the oldTimers will figure it out, but we didn’t want a lot of email from them, hence this section.  
<sup>2</sup> We are aware that Phosphorus is not a precious or semiPrecious stone. Please stop emailing.  
<sup>3</sup> It took some doing, but we also figured out how to stop T<sub>E</sub>X from introducing hyphens into our document. Use the following.

```
\tolerance=10000  
\hbadness=10000  
\hyphenchar\font=-1  
{\tt \hyphenchar\font=-1}
```

<sup>4</sup> This and the preceding example are actual code taken from production working code at an Internet startup we cannot name due to *legal reasons*.

<sup>5</sup> In keeping with other popular software projects, versions are named by adjectiveFurniture pairs, where successive versions (more properly termed “releases”) begin with pairs of successive letters of the English alphabet. We could not come up with the zeroth or negative first letter of the alphabet, so simply began our release names with AvariciousArmoire and documented the reason why.

<sup>6</sup> or squinting

```
|1 anonFunction |2 x 2| |3 numPlus x 1 3| 1|
<like (lambda (x) (+ x 1.0))>
```

## Mathematical Formulas

By typesetting this document in  $\text{T}_{\text{E}}\text{X}$ , we are basically obliged to include some complicated mathematical equations. It is also a well known fact that conference papers are scanned for displayed equations, only those with difficult math then taken seriously enough for primary consideration. Therefore, in keeping with our “popular is better” philosophy, we hereby popularize this document by reproducing one of the author’s favorite calculus problems from high school.<sup>7</sup>

$$\int e^x \sin x \, dx$$

## Fixing COND

As pointed out in the extensive literature published by Paul Graham [CITATION NEEDED], the often used COND Lisp form is too verbose by two parenthesis per case ( $\Theta(2n)$ <sup>8</sup>). Following is an example of the bad old style Scheme cond, followed by an equivalent example of Phosphorous’ multiIf.

```
(cond [(> e 0) (go 'positive e)]
      [(< e 0) (go 'negative e)]
      [t (go 'zero e)])

|12 multiIf |23 gT e 0 32| |32 doP e 23|
|34 1T e 0 43| |43 doN e 34|
|56 doN e 65| 21|
```

Of course, the Phosphorous code only becomes more popularizable when dressed up with some comments that look like XML.<sup>9</sup>

## Virtual Machine Targets

Most modern programmers are smart enough to figure out that there must be something better than Java. Despite its popularity, Java has become associated with oldness, which is never popular. These same programmers are not, however, smart enough to figure out that the Java Virtual Machine (JVM) is equivalently awful. This is because they have to think about and live with the details of the language on a dayToDay basis, whereas the awfulness of the JVM can hide behind the thin facade of Eclipse, or in a pinch, the java command. At the same time, programmerManagers love Java because everybody else is doing it, which means there’s one less thing that can be held against them when their project finally collapses *under the crushing weight of their foolish incompetence*. This confluence of forces has

given rise to a number of programming languages which target the JVM.

The interesting observation here is that because the Java programming language and Java Virtual Machine are (surprise!) so tightlyCoupled, new language designers are compelled to make their languages such that they use only those features they can implement efficiently on the JVM. For example, implementations of Scheme for the JVM either lack call/cc or have a very slow and slightly buggy implementation of it. We call this the *Gosling Tarpit*.

Phosphorous escapes the Gosling Tarpit by means of a VirtualVirtual Machine (V2M). Some legitimately smart person can then figure out how to do continuations on the JVM once and for all and we can package it up into a callWithCurrentContinuation opcode on the V2M. Retargeting the V2M to the CLR is a work in progress.<sup>10</sup>

## Growing a Language

Most new soCalled “scripting” programming languages grow over time, developing new features and incompatibilities fearlessly. In contrast, Lisp has been quite slow to regenerate its various language standards. As a result, we have designed into Phosphorous a sophisticated plan for a pattern of growth. We call it our twoPronged approach to growth.

One, we make our implementation open source. We will make our money by having our language become popular enough that everyone feels compelled to buy the O’Reilly book for it, and we will write the O’Reilly book.

Two, we will undergo a vigorous ISO standardization effort.

## No BoltOns

The worst thing that can be said about a programming language is that its object system is “bolted on.” Let me tell you, our object system is not bolted on. It’s not even close to bolted on.

## Lexical Scoping

All modern popular programming languages got lexical scoping wrong the first few times they tried it<sup>11</sup>, causing unimaginable amounts of old code to break when the problems were fixed, as well as other ills. Phosphorous eliminates this problem by getting lexical scoping exactly right in its written material, but introducing the usual argList bugs in the actual implementation. In this way, we can reap all the benefits of being like popular languages, yet our reputations as language designers can stay pure. We will leave the bugs in until the language takes root as a popular alternative to others.

<sup>7</sup> It has been years since I went to high school, and I don’t remember much more than that, other than that the solution has a clever bit of recursion in it. Hopefully, this will be enough displayed math.

<sup>8</sup> The use of the Master Theorem here allows us to win a bet that I would one day use it for something other than my Google Interview. That’s *two* Dogfish Head IPAs you owe me now, SPH.

<sup>9</sup> Notice how, in the Scheme code, square brackets are used to show brackets that impose structure without being a more usual (foo bar) form. The same pairings can be moved implicitly into the namedParenthesis of Phosphorous by using the same digits in different orders to mark related forms.

<sup>10</sup> It would go a lot faster if we had some research grants. I’m looking at you SPJ.

<sup>11</sup> It is possible that this is why Emacs Lisp has wisely never made the attempt.