

[OT] PostLisp, a language experiment

Source: <http://coding.derkeiler.com/Archive/Lisp/comp.lang.lisp/2005-03/0656.html>

From: Ulrich Hobelmann (*u.hobelmann_at_web.de*)

Date: 03/10/05

Date: Thu, 10 Mar 2005 01:05:47 -0600

(Sorry for crossposting this. If you specifically refer to one language family only, please post the answer only to one group.)

There are several things I (and others) don't really like about Lisp and Forth, but most live with it. Today I set out to try a mix of both to see, how certain things improve.

IMHO the worst thing in Lisp is its many parentheses. While they do a great job in structuring the language and allowing features such as variable numbers of arguments and macros, they turn every program into a ragged tree (while a Forth function typically is one or two lines). I don't say this is terrible, but I thought I might try to improve on it.

The worst thing about Forth IMHO is the stack clutter. When every function has some swaps, dups, overs and nips in it, it takes concentration to keep track of the stack in your mind. Sure, there are stack comments, but when you write those, you might just as well use local variables instead.

The good thing about Lisp is its structure: there are no confusing compile and interpretation modes nor words like postpone, immediate, [, or].

The good thing about Forth is simplicity: functions can be very short. If you want to chain several functions you just concatenate them in Forth: bla baz bar foo; in Lisp this looks like (foo (bar (baz bla))), which is reverse order. And while I could write a macro like (with bla baz bar foo) in Lisp, I just wanted to try things out here.

Please note that this is just the product of a couple of hours of toying around, nothing more. Don't take it too serious.

First here's how some code could look in a simple Lisp (not Common Lisp). The ret is the explicit return continuation. (I smell a ret!) Macros are implicitly quasiquoted, somewhat like in Scheme. That means that most of their definition is just inserted in place of the macro invocation.

The `,` is Lisp syntax to say that that part is not inserted verbatim, but instead the **value** of the unquoted variable is inserted. `,@` does the same, but if it's unquoting a list, the elements of the list (not the list) is inserted.

```
(macro (when cond . body)
  (if cond ,body ()))
```

```
(macro (loop (i a b) . body)
  (let (,i a)
    (when (> ,i b)
      (label start)
      ,@body
      (set i (1+ i))
      (when (< ,i b) (goto start))))))
```

```
(var foo 5)
(fun (fac n ret)
  (if (<= n 1)
    (ret 1)
    (ret (* (fac (- n 1))
            n))))
```

```
; alen = length; a! = store; a@ = fetch
(fun (map fn array ret)
  (let (len (alen array) ar (newarray len))
    (loop (i 0 len)
      (a! ar i (fn (a@ array i))))
    (ret ar)))
```

```
; aprint = print array
(fun (test ret)
  (aprint (map (fn (n ret) (ret (+ n 2))) (array '(1 2 3 4))))))
```

My first attempt to forthify this was the following. Like Forth code it is sequential. Words that end in `:` are prefix and take a fixed number of arguments (something like Forth definition words). To group code, `()` it. `()s` in code that is not used by some prefix word are like a Forth local declaration; they define variables by eating values from the stack. Note that I wanted to get away from stack manipulation; thus, there are no `dup`, `over` etc. The stack is just for implicit parameter passing. I know this makes code longer in general, but I hope it helps readability.

In the macros all code is compiled verbatim; `:variables` are substituted with their value at macro invocation time (like `,` in Lisp), and `::variables` are "spliced in" (like `,@` in Lisp).

```
macro: when (body) (if: :body ())
```

```
macro: loop (i body)
  ((:i b) :i b <=
   when: (label: start ::body :i 1+ (:i) :i b < when: start goto))
```

```
5 var: foo
fun: fac ((n ret) n 1 <= when: (1 ret) n 1 - fac n * ret)
```

```
fun: map ((fn array ret) array alen (len) len newarray (ar)
  0 len loop: i (array i a@ fn call ar i a!) ar ret)
```

```
# note the implicit argument in the anonymous function
# it expects one number to be on the stack for addition
fun: test ((ret) fn: (ret) (2 + ret) array: (1 2 3 4) map aprint)
```

This looks somewhat cluttered, I think. There's less ()s than in Lisp, but :s everywhere. Also, syntax like "fun: fac (...)" looks almost as ugly as C.

So my second (and last so far) attempt turns

```
fun: name (code)
```

```
into
```

```
(FUN name args ... IS code ...)
```

The explicit () and the IS delimiter, so that there can be arguments without using ()s everywhere. Specials are capitalized for better readability. The when macro is useless, since I introduced another delimiter (ELSE) in the IF list, again, saving ()s. -> is used as a special form to bind variables (slightly more verbose than before).

```
(MACRO LOOP i . body IS
```

```
  (-> a b) :i b <=
```

```
  (IF (LABEL start) ::body :i 1+ (-> :i) :i b < (IF start goto)))
```

```
5 (VAR foo)
```

```
(FUN fac var n ret IS n 1 <= (IF 1 ret ELSE n 1 - fac n * ret))
```

```
(FUN map fn array ret IS array alen (-> len) len newarray (-> ar)
```

```
  0 len (LOOP i array i a@ fn call ar i a!) ar ret)
```

```
(FUN test ret IS (FN ret IS 2 + ret) (ARRAY 1 2 3 4) map aprint)
```

I think having everything INSIDE the ()s makes it more readable. The IF ELSE also looks more Forth-like.

I plan for PostLisp to be more array than list-oriented (except for parsing and macros etc.), as the map function suggests, so maybe it should be called POAP (POstfix Array Processing). Don't mix this up with the *simple* SOAP language (Simple Object Access Protocol) that looks like this:

```
<soap:Body xmlns:m="http://www.stock.org/stock">
  <m:GetStockPrice>
    <m:StockName>IBM</m:StockName>
```

```
</m:GetStockPrice>  
</soap:Body>
```

with a result of:

```
<soap:Body xmlns:m="http://www.stock.org/stock">  
  <m:GetStockPriceResponse>  
    <m:Price>34.5</m:Price>  
  </m:GetStockPriceResponse>  
</soap:Body>
```

Isn't it cute? Translating the above example functions into SOAP is left as an exercise to the reader ;)