A Case When EVAL is Necessary

Gene Michael Stover

created 2005 July 17
updated Thursday, 2006 January 19

Contents

1 It's been solved

On Thursday, 2006 January 19, Jörg-Cyril Höhle sent me an idea for a solution which satisfied all three of the Test Cases (Section 5) I originally defined. You can read about Höhle's solution in Appendix A.

I originally wrote the essay in 2005 July. As I'm reading it in 2006 January, I can think of another test case which won't work without EVAL , & I wonder why I didn't choose it as my test case originally.

That new test case would be like the three from Section 5 except that the values of the key/value pairs are mathematical expressions. For example, if one line of the input file was "`a "1 + 2"`", then processing it with LOAD-FUNK should produce a Lisp function called A which returns the numeric value 3.

Since the point of the exercise is not to convert infix mathematical expressions to Lisp, we could simplify the input file format. In the simplified input file format, the values are Lisp expressions.

- For example, one line of the simplified input file might be "`a (+ 1 2)`". After processign it with LOAD-FUNK, we should have a function A which returns 3.
- Another line from the simplified input file might be "`b (reduce #'+ (mapcar #'log '(1 2 3)))`". It would create a function B which returned about 1.79.
- Yet another line from the file might be "`c (/ (get-universal-time) 2)`". It would create a function C which returned different values depending on when you called it.

I'll betcha that requires EVAL .[1]

But Höhle definitely solved the original challenge. And it's a nice solution, too.

The rest of this essay is the original essay, except for Appendix A, which Jörg-Cyril Höhle 's solution.

2 What is this?

Good Lisp programmers avoid using EVAL . I've read many times the heuristic "If you used EVAL , you look at what you wrote & think about doing it with macros instead". It's good advice, but few heuristics are always true.[2]

[1] It could also be done with a special-purpose interpreter, but as the allowed expressions increase, your interpreter grows in complexity until it is effectively a re-implementation of Lisp in its entirety.

[2] I was tempted to write "No heuristic is always true", but that would itself be an absolutist heuristic, surely not always true.

Here I discuss a case in which EVAL is pretty much necessary.

I'm open to the possibility that I'm wrong, that EVAL isn't the only way to do it. In fact, I'd be happy to be wrong. If you come up with a way to do the task I describe here without using EVAL , please send it to me. I'll insert it in this essay to show how it can be done.

But before you send me your solution, make sure it works with the test cases. Those are the requirements. If it doesn't satisfy those requirements, it isn't a solution.

3 The Problem

Let's say we have a data file that does *not* contain Lisp forms, but we want to construct Lisp functions from that data, & we want to do it at run-time. The Lisp programmer's first & worthy instinct is to write a macro, but I claim that a macro won't work in this situation. You must use EVAL directly or indirectly.

Using LOAD counts as using EVAL in my opinion because the main feature of LOAD can be implemented like this:

```
(defun basic-load (pathname)
  "The most basic LOAD possible.  Doesn't do IN-PACKAGE
or other special features of real LOAD."
  (with-open-file (strm pathname)
    (loop for x = (read strm nil strm)
          until (eq x strm)
          do (eval x))))
```

I claim that either you must read data, create Lisp forms, & use EVAL on them, or you must compile the data to Lisp forms in a file, then use LOAD on that file.

4 Examples

Here are some examples of the problem.

4.1 ONC RPC

The interfaces to ONC RPC [1] services are usually described in "rpcgen". Such a file might look like this:

```
/* example service in ONC RPC */
/* from page 82 of "Power Programming with RPC" */
const MAXNAMELEN = 512;
typedef string nametype<MAXNAMELEN>;
typedef struct namenode *namelist;
```

```
struct namenode {
  nametype name;
  namelist pNext;
};

union readdir_res switch (int errno) {
case 0:
  namelist list;
default:
  void;
};

program DIRPROG {
  version DIRVERS {
    readdir_res
    READDIR (nametype) = 1;
  } = 1;
} = 0x20000001;
```

If we had a function or macro called LOAD-RPC, it would be reasonable to expect that we could load the rpcgen file like this:

```
;;; One way we might load the example rpcgen file, above
(load-rpc "dirprog.xdr")
```

or like this:

```
;;; Another way we might load the example rpcgen file, above
(load-rpc (make-pathname :name "dirprog" :type "xdr"))
```

Afterwards, we could expect to have a function called DIRPROG which, given a NAMETYPE & a remote host identifier of some sort[3], would return a list of files in that remote directory.

In this ONC RPC example of the problem3 , we have the non-Lisp data file, & we want to read it at run-time & end up with some Lisp functions & maybe some new data type derived from the ONC RPC file.

4.2 Web Services

Web services are often described in the Web Services Description Language (WSDL, [6]), which is analogous to `rpcgen` for ONC RPC. It would be cool to use a WSDL file from Lisp like this:

```
(load-wsdl "the-service.wsdl")
```

[3]By remote host identifier, I mean a hostname, network address, ONC RPC *client* object, or whatever was required by a Lisp implementation of ONC RPC.

Then we could expect to use that web service by calling regular Lisp functions which handled the details of talking to a web service.

Such a LOAD-WSDL function would be an example of the problem3 because we would have a non-Lisp data file from which we expected Lisp to derive Lisp functions & types.

What's more, web services can be located through UDDI or other registries[4], & those registries identify the WSDL. So the WSDL file might initially be remote, not on the local file system.

5 The Test Cases

For my discussion here, let's use an instance of the problem3 that's simple so we can see the problem at work. If we used a more complicated real-world example, such as loading a WSDL file, the work of parsing & interpreting the WSDL would overshadow the "macro vs. eval" problem.

For the test case, let's say we want a non-Lisp data file to specify key/value pairs. Each key should become a Lisp function of no arguments which returns the associated *constant* value. Here's an example file:

```
a 1
b 2
c 3
```

Let's say we have a function called LOAD-FUNK[5], then after calling LOAD-FUNK on that input file, we could expect it to have defined three Lisp functions called A, B, & C as if we had declared them like this:

```
(defun a () 1)
(defun b () 2)
(defun c () 3)
```

The challenge is to implement LOAD-FUNK.

For the following test cases, assume that there exists a file called `test.funk` with these contents:

```
a 1
b 2
c 3
```

[4] I have only limited experience with web services, & no experience with UDDI & other registries, so if what I've read of them is out of date & UDDI is no longer used, please forgive me. I'm sure you get my point even if UDDI is out of fashion & I should be saying Web Users Service System (WUSS) or whatever.

[5] "Funk" is a mnemonic for "*f*ctions ret*u*rn*i*n*g* *k*onstants". It's not important. It's just the term I'll use for this test case.

5.1 Hard-coded namestring

Start a plain vanilla Lisp session.
Load (or otherwise define) your implementation of LOAD-FUNK.
Evaluate these forms:

```
(load-funk "test.funk")
(list (a) (b) (c))
```

You should encounter no errors.
The result of the second form should be "`(1 2 3)`".

5.2 Evaluated argument

Start a plain vanilla Lisp session.
Load (or otherwise define) your implementation of LOAD-FUNK.
Evaluate these forms:

```
(load-funk (make-pathname :name "test" :type "funk"))
(list (a) (b) (c))
```

You should encounter no errors.
The result of the second form should be "`(1 2 3)`".

5.3 Local variable argument

Start a plain vanilla Lisp session.
Load (or otherwise define) your implementation of LOAD-FUNK.
Evaluate these forms:

```
(let ((pn "test.funk"))
  (load-funk pn))
(list (a) (b) (c))
```

You should encounter no errors.
The result of the second form should be "`(1 2 3)`".

6 READ-FUNK helper function

This function will be useful.[6]

```
(defun read-funk (strm)
  "Read the next function constant pair from STRM.
Return a four-element list (defun NAME () VALUE) if
there is such a next element.  Otherwise, return
```

[6] If you send me your own implementation to show me how it can be done without EVAL, you may use or ignore this function as you choose.

```
    STRM to indicate end-of-input."
      (let* ((name (read strm nil strm))
             (value (read strm nil strm)))
        (if (not (or (eq name strm) (eq value strm)))
            (list 'defun name () value)
          strm))) ; end of file
```

7 Solution Using Macros

So I've made this claim that you can't do it with macros. Here's an implementation that uses macros. I'll call it LOAD-FUNK-NSK because it assumes its argument is a *constant namestring* (nsk).

```
(defmacro load-funk-nsk (nsk)
  (with-open-file (strm nsk)
    `(progn
       ,@(loop for x = (read-funk strm)
               until (eq x strm)
               collect x))))
```

Let's try is on the first test case (Section 5.1):

```
$ lisp

> (defun read-funk (strm)
      "Read the next function constant pair from STRM.
    Return a four-element list (defun NAME () VALUE) if
    there is such a next element.  Otherwise, return
    STRM to indicate end-of-input."
      (let* ((name (read strm nil strm))
             (value (read strm nil strm)))
        (if (not (or (eq name strm) (eq value strm)))
            (list 'defun name () value)
          strm))) ; end of file

READ-FUNK
> (defmacro load-funk-nsk (nsk)
      (with-open-file (strm nsk)
        `(progn
           ,@(loop for x = (read-funk strm)
                   until (eq x strm)
                   collect x))))

LOAD-FUNK-NSK
```

Demonstrate that function A is not already defined:

```
> (a)
; in: LAMBDA NIL
;     (A)
;
; caught STYLE-WARNING:
;   undefined function: A
0] abort

> (load-funk-nsk "test.funk")

C
> (list (a) (b) (c))

(1 2 3)
```

LOAD-FUNK-NSK passed the first test perfectly. Let's try it on the second test:

```
$ lisp

> (defun read-funk (strm) ...) ; details omitted for brevity

READ-FUNK
> (defmacro load-funk-nsk (nsk) ...) ; details again

LOAD-FUNK-NSK
> (load-funk-nsk (make-pathname :name "test" :type "funk"))

debugger invoked on a TYPE-ERROR in thread 24579:
  The value (MAKE-PATHNAME :NAME "test" :TYPE "funk")
  is not of type
    (OR (VECTOR NIL) BASE-STRING PATHNAME STREAM).
0]
```

The LOAD-FUNK-NSK macro fails for the second test case because it assumes is argument is a literal, but the argument in the second test case is a form which evaluates to a pathname.

We could force LOAD-FUNK-NSK to work for the second test case by having it try its argument as a literal, then call EVAL on it (thereby providing support to my claim), but that kind of kluge would make me feel dirty. And it would still fail for the third test case.

8 Other Implementation Ideas

What if LOAD-FUNK were to read the contents of the *.FUNK file into a global variable, & then it used a macro? The macro wouldn't have an argument; it would look at the global variable for the forms.

This won't work, but before I explain why, here's the code for it to explain what I mean.[7]

```
;; yucky LOAD-FUNK-000 implementation

(defvar *forms* ()
  "Private to LOAD-FUNK-000 & its macro.  In
a real system, if you used this technique,
you'd probably make it an unexported global
variable in the package that held LOAD-FUNK-000.")

(defmacro mymacro ()
  "Private to LOAD-FUNK-000.  Note the self-
documenting name.  My imagination took a powder."
  (cons 'progn *forms*))

(defun load-funk-000 (pn)
  ;; Create DEFUN forms from the "funk"
  ;; file & save them in *FORMS*.
  (let ((*forms* (with-open-file (strm pn)
                   (loop for x = (read-funk strm)
                         until (eq x strm)
                         collect x))))
    (mymacro)))
```

It won't work because the MYMACRO will be expanded inside LOAD-FUNK-000, when that function is defined, which is when *FORMS* is empty.

Besides that, even if it worked, I'm not convinced that this use of a macro wouldn't be EVAL by another name.

What about a macro that returned a Lisp expression which opened & read the file (thereby removing the problem with evaluating expressions for the pathname), then that form handed the list of DEFUNs it collected with READ-FUNK to a macro?

I'm almost certain there is no such macro.

I have a few other ideas, but they are twists on things I've already described. I'm sure they won't work. I'm sure you can't do it without EVAL or LOAD.

9 The solution with EVAL

Here's an implementation of LOAD-FUNK which uses EVAL .

```
(defun load-funk-eval (pn)
  (with-open-file (strm pn)
```

[7] When explaining what you mean would take to long, just say what you mean in code. It'll probably take less space than trying to save space by explaining what you mean. Know what I mean?

```
        (loop for x = (read-funk strm)
              until (eq x strm)
              do (eval x))))
```

Let's try it with the three test cases.
Here it is with the first test case:

```
* (defun read-funk (strm) ...) ; details omitted for brevity

READ-FUNK
* (defun load-funk-eval (pn) ...) ; details, again

LOAD-FUNK-EVAL
* (load-funk-eval "test.funk")

NIL
* (list (a) (b) (c))

(1 2 3)
*
```

Now with the second test case:

```
* (defun read-funk (strm) ...) ; details, again

READ-FUNK
* (defun load-funk-eval (pn) ...) ; details, again

LOAD-FUNK-EVAL
* (load-funk-eval (make-pathname :name "test" :type "funk"))

NIL
* (list (a) (b) (c))

(1 2 3)
*
```

So far so good. Now the third test case.

```
* (defun read-funk (strm) ...) ; details, again

READ-FUNK
* (defun load-funk-eval (pn) ...) ; details, again

LOAD-FUNK-EVAL
* (let ((pn "test.funk"))
    (load-funk-eval pn))
```

```
NIL
* (list (a) (b) (c))

(1 2 3)
*
```

All tests pass, & LOAD-FUNK-EVAL is a nice, short function, too. It's only problem is that it uses EVAL .

10 The solution with LOAD

It's also possible to implement a LOAD-FUNK with LOAD.

```
(defun make-tmp-pathname (source)
  ;; In reality, you'd use a better way of
  ;; making a unique or temporary file name.
  ;; Maybe you'd use a technique that allowed
  ;; you to identify the temporary file's
  ;; name from the source file's name.
  (make-pathname :name "abcdefg" :type "lisp"))

(defun load-funk-load (pn)
  (let ((tmp (make-tmp-pathname pn)))
    ;; Compile the name/value pairs file to
    ;; Lisp in a temporary file
    (with-open-file (istrm pn)
      (with-open-file (ostrm tmp :direction :output)
        (loop for x = (read-funk istrm)
              until (eq x istrm)
              do (print x ostrm))))
    ;; Load the compiled Lisp forms from the
    ;; temporary file.
    (load tmp)
    (delete-file tmp))) ; cleanup
```

First test case:

```
* (defun read-funk (strm) ...) ; details, again

READ-FUNK
* (defun make-tmp-pathname (source) ...) ; details, again

MAKE-TMP-PATHNAME
* (defun load-funk-load (pn) ...) ; details, again
```

```
LOAD-FUNK-LOAD
* (load-funk-load "test.funk")

T
* (list (a) (b) (c))

(1 2 3)
*
```

Second test case:

```
* (defun read-funk (strm) ...) ; details, again

READ-FUNK
* (defun make-tmp-pathname (source) ...) ; details, again

MAKE-TMP-PATHNAME
* (defun load-funk-load (pn) ...) ; details, again

LOAD-FUNK-LOAD
* (load-funk-load (make-pathname :name "test" :type "funk"))

T
* (list (a) (b) (c))

(1 2 3)
*
```

Third test case:

```
* (defun read-funk (strm) ...) ; details, again

READ-FUNK
* (defun make-tmp-pathname (source) ...) ; details, again

MAKE-TMP-PATHNAME
* (defun load-funk-load (pn) ...) ; details, again

LOAD-FUNK-LOAD
* (let ((pn "test.funk"))
    (load-funk-load pn))

T
* (list (a) (b) (c))

(1 2 3)
*
```

This technique hides the EVAL within LOAD , but it still might be preferable because it distinguishes between compilcation & evaluating.

The separation of compilation & loading opens some possibilities including:

- Don't delete the temporary Lisp file. Instead, use it as a cache for the compiled code, recompiling only when that file is stale.
- Allow your system builder utility (i.e., ASDF) to create the temporary Lisp file.

11 Conclusion: Why macros didn't work

The reason macros don't work in this case is that macros manipulate Lisp code as the macro is expanded, but we didn't have the Lisp code when we could use the macros. We were deriving Lisp code from non-Lisp data. By the time we had the Lisp code, the macros were already expanded.

So sometimes, you have to use EVAL.

12 Conclusion: Don't use EVAL

Immediately after saying "sometimes, you have to use EVAL", here I am saying that EVAL is best avoided. In fact, we probably wouldn't need it even for the ONC RPC & WSDL examples I gave.

This whole EVAL article came up because I was thinking of how to derive Lisp code for ONC RPC & WSDL interface descriptions. I wanted to use those types of services with local Lisp function calls which hid the remoteness of the remote procedure calls.

For example, assuming the `rpcgen` description of the PORTMAPPER service was in a file called `portmapper.rpc`, I wanted to do something like this:

```
* (load-rpc "portmapper.rpc")

T
* (pmap-getmaps "127.0.0.1")
((ping 100115) (hostmem 100112) (portmapper 100000)
 ...)
```

But the fact that I wanted to type Lisp code which did much of the work statically shows that I was assuming a lot at compile-time. So a compile-time translation step would have been appropriate, whether it was hidden by some kind of `load-rpc` function, hidden by ASDF, or totally explicit & outside of Lisp in an un-Lispy `makefile`. Even if I was using some kind of RPC library that always did things dynamically (but without creating code), & I wanted a more static interface to the services, I would have written wrapper functions around the more dynamic interface of the RPC library. So no matter how I dealt with

it, directly using EVAL would be unnecessary if I wanted a static interface to the remote services.

On the other hand, if I had wanted more dynamic access to, say, web services, a static interface would not have been appropriate in the first place.

So in practice, EVAL is still best avoided.

But if you really wanted to generate Lisp functions from a dynamically named interface description, I still think you'd need EVAL (or LOAD).

A The solution

On Thursday, 2006 January 19, Jörg-Cyril Höhle suggested an implementation using SETF SYMBOL-FUNCTION which passes all three tests. He only supplied pseudocode, not a full implementation. Here's my implementation of his pseudocode.

I made one file, called `hoehle.lisp`, with a READ-FUNK helper function from that of Section 6 & with a LOAD-FUNK implementation which uses Höhle's idea. Here's the code from that file:

```
(defun read-funk (strm)
  "Read the next function constant pair from STRM.
Return a four-element list (defun NAME () VALUE) if
there is such a next element.  Otherwise, return
STRM to indicate end-of-input."
  (let* ((name (read strm nil strm))
         (value (read strm nil strm)))
    (if (or (eq name strm) (eq value strm))
strm ; end of file
        (list name value))))

(defun load-funk (pn)
  "From Hoehle's idea"
  (with-open-file (strm pn)
    (do ((x (read-funk strm) (read-funk strm))
         (lst nil (cons (first x) lst)))
        ((eq x strm) lst)
        ;; Here's the important part!!!
        (setf (symbol-function (first x))
              #'(lambda () (second x))))))
```

Try the first test. It's from Section 5.1:

```
$ lisp
> (load "hoehle.lisp")

T
> (load-funk "test.funk")
```

```
(C B A)
> (list (a) (b) (c))

(1 2 3)
>
```

Nice. Now the second test (Section 5.2):

```
$ lisp

> (load "hoehle.lisp")

T
> (load-funk (make-pathname :name "test" :type "funk"))

(C B A)
> (list (a) (b) (c))

(1 2 3)
>
```

Nicer.

Now the third & final test (Section 5.3):

```
$ lisp

> (load "hoehle.lisp")

T
> (let ((pn "test.funk"))
      (load-funk pn))

(C B A)
> (list (a) (b) (c))

(1 2 3)
>
```

Excellent![8]

It satisfies all three tests which I defined when I originally wrote this essay. Jörg-Cyril Höhle nailed it.

[8] Did you think I was going to say "nicest"?

B Other File Formats

- This document is available in multi-file HTML format at http://cybertiggyr.com/gene/eval/.
- This document is available in Pointless Document Format (PDF) at http://cybertiggyr.com/gene/eval/eval.pdf.

I write almost all of my documents in LaTeX ([5], [3]). I compile to PDF with `latex`, `dvips`, & `ps2pdf`. I compile to HTML with `latex2html` ([2], [4]).

References

[1] John Bloomer. *Power Programming with RPC*. O'Reilly & Associates, Inc., 1992. ISBN 0-937175-77-3.

[2] Nikos Drakos. latex2html.

[3] Michel Goossens and Frank Mittelbach. *The LaTeX Companion*. Addison Wesley Longman, Inc., 1993. ISBN 0201541998.

[4] Michel Goossens and Sebastian Rahtz. *The LaTeX Web Companion: Integrating TeX,* HTML*, and* XML. Addison Wesley Longman, Inc., 1999. ISBN 020143317.

[5] Leslie Lamport. *LaTeX: A Document Preparation System*. Addison-Wesley Publishing Company, Inc., 1986. ISBN 0-201-15790-X.

[6] W3C. Web services description language (wsdl) version 2.0 part 0: Primer. *w3.org*, May 2005. http://www.w3.org/TR/wsdl20-primer/.