Common EVAL

Henry Lieberman

Media Laboratory
Visible Language Workshop
Massachusetts Institute of Technology
Cambridge, Mass. 02139 USA
Henry@AI.AI.MIT.Edu

Christopher Fry

Hip Software Corporation
150 Walnut St.
Somerville, Mass. 02145 USA

Abstract

We propose that the Common Lisp standard be extended by adding to the language specification a short program, itself written in Common Lisp, to implement the EVAL function. We call this *Common* EVAL. The interpreters for every correct implementation of Common Lisp would be required to match the semantics of Common EVAL on valid Common Lisp expressions. It should treat other expressions as errors or as implementation dependent extensions.

There are three cogent reasons for including a Common EVAL in the standard: First, since EVAL definitively specifies the behavior of Lisp programs, Common EVAL would insure uniformity of program semantics across implementations. Second, it would aid validation efforts, since the behavior of a particular implementation could always be compared to the behavior of Common EVAL. Third, it would facilitate the creation of debuggers and other program-manipulating programs that could be ported across Common Lisp implementations.

"Gross internals: Do not look at the code on this page if you value your sanity."

-- Comment found in the code for EVAL in a popular Common Lisp implementation

The Status Quo

One of the marvelous things about Lisp is that the language can be written in itself. In Lisp, programs can be data and data can be programs. The overall operation of the language can be described as a function written in the very same language: the EVAL function. Lisp advocates always point with pride to this fact as a major reason for Lisp's superiority over conventional languages. It is why Lisp has traditionally supported the best debugging environments. It is why Lisp can be used to write the program-manipulating programs which are essential in artificial intelligence work, and in building advanced interactive programming environments.

Sadly, this important advantage is becoming steadily eroded by some of the modern production implementations of Lisp. The quest for efficiency and experimentation with esoteric programming constructs are leading to non-standard implementations of Lisp interpreters that foil attempts to make use of Lisp's self-descriptive capability.

How Did We Get Here?

There are many pressures to deviate from standard Lisp semantics in production implementation of Lisp. Below are some real-life situations.

The primary temptation is to improve efficiency. The interpreters for MIT-descended Lisp Machines by Symbolics, LMI and TI show how production implementations have compromised Lisp semantics. Surprisingly, if you look at the system's definition of the EVAL function, you will find that it only *appears* to be written in Lisp. It calls "subprimitives" which are special-cased by the compiler, compiling into specialized microcode. For example, some subprimitives violate the stack discipline of Lisp. These optimizations undoubtedly make the interpreter more efficient, but the consequence is that the system's definition of EVAL cannot even be interpreted by EVAL itself!

Because of the additional complexity that machine-dependent efficiency hacks add to the evaluator, it is no longer feasible for a user to write an EVAL without subprimitives, and have any confidence that the results will be equivalent to the a particular implementation's EVAL. Worse, if the code for the evaluator relies on subprimitives, it won't even be intelligible to the human reader literate only in Lisp. This is not to say that we are against subprimitives; obviously, they are necessary for such functions as CAR and EQ. But the behavior of CAR and EQ is not problematic; the behavior of EVAL is.

Language Extensions

Another central problem is that "extensions" to the Lisp language may be implemented in the interpreter by low-level constructs that cannot be directly implemented by a Lisp program. While Common Lisp is designed to permit extensions to the language, it should not allow extensions which rely on microcode and other non-Lisp implementation techniques to change the basic semantics of the language. Such extensions effectively prevent any program-manipulating program written in Common Lisp from operating on code containing the extensions.

Spaghetti stacks in Interlisp are an example where an attempt to implement non-standard programming constructs wreaks havoc with the interpreter's semantics. It is impossible for an Interlisp user to write a stepping debugger capable of working on interpreted code that uses spaghetti stacks.

The Symbolics flavors implementation relies on microcode for message reception and method invocation, and has no fully specified Common Lisp description. A significant [though not serious] semantic discrepancy was discovered by one of the authors in the course of porting a Symbolics flavors program to Franz Allegro Common Lisp, of the sort that could be easily settled by a procedural implementation standard.

The recent convergence on CLOS [1] as a standard for an object-oriented programming extension to Common Lisp is commendable. Xerox has made publicly available an implementation called PCL, an object system approaching the proposed CLOS standard. Widespread dissemination of CLOS implementations in Common Lisp will likely result in greater uniformity than the several flavor implementations that have appeared. [PCL does contain, however, both implementation-dependent non-Common-Lisp conditional code, and a code walker, both of which may give rise to discrepancies. This kind of problem might be alleviated by a complete Common `EVAL`.] The Common Lisp CLOS could be viewed as being a piece of a Common `EVAL`, restricted to that part of the interpreter concerning the object-oriented programming constructs.

Why Make Eval Precise?

Many advanced Lisp applications rely on the precise details of the operation of `EVAL`. Implementing a single-stepping debugger for Lisp code, for example, requires imitating `EVAL` on Lisp expressions, while inserting display operations and requesting input at events during evaluation [2, 3].

The lack of a Common `EVAL` means that implementors of programming environment tools must now track internal changes to a particular implementation's evaluator. For example, Symbolics recently changed the function cell of an interpreted function from containing a list beginning with the symbol `LAMBDA` to `SI:DIGESTED-LAMBDA`, which necessitated similar changes in any program which expected to interpret functions. A standard `EVAL` would clarify what representations a user could rely on, and clarify what representations an implementor could change without breaking system code or affecting users.

Lisp's extensibility makes it ideal for defining embedded application-dependent or experimental languages, which may even have a different control structure than Lisp's. Often, these languages need to "use Lisp" as a subset, call Lisp functions from code in the other language, or even invoke foreign language code from Lisp code, to avoid having to duplicate all of the host implementation's facilities. For the interface to be smooth, the language designer must be able to depend on how Lisp code is evaluated, perhaps including details such as variable environments and function definitions.

Many programs which need to analyze Lisp programs statically require a "code walker", a program that determines which subexpressions of a Lisp expression represent code to be evaluated, and which represent data, like expressions which appear inside a QUOTEd list. Such code walkers, which separate the uses of Lisp expressions as programs from uses as data, appear in virtually every Lisp compiler. Smart pretty-printers that print expressions according to their semantics, indexers, or other "programmer's apprentice" tools need this, too. Code walkers anticipate the action of EVAL on an expression, so they are inextricably tied to EVAL's operation.

With Common EVAL, a designer of a program-manipulating program can base their tools on the definition of Common EVAL rather than the details of a particular Lisp implementation. The implementor can then have confidence that the tools will work in all valid implementations of Common Lisp.

Introducing Common EVAL

> "`eval` *form* [Function]
>
> Whatever results from the evaluation is returned from the call to `eval`."
> -- **From** *Common Lisp: The Language*

With Common Lisp, we have a unique opportunity to insure that the program-data equivalence which is one of Lisp's cornerstones remains available as Lisp implementations proliferate. But the English description in the Steele book is not detailed enough to exclude semantic deviations which frustrate serious developers of program-manipulating programs. Specifying the EVAL function as a Common Lisp program can provide a concise, precise, and easily understood description which could serve as a guide for implementors and a means by which to evaluate the results.

Implementors would *not* be required to run the exact code for Common EVAL in their implementation. They could provide another implementation, which may be more efficient or include extra features, but they would be required to assure that their version matched the semantics of Common EVAL. Differences in behavior between a particular Lisp implementation and Common EVAL would be evidence for violations of the Common Lisp standard.

Implementing Common EVAL

How detailed should the Common EVAL implementation be? It is possible to implement a wide range of meta-circular interpreters ranging from a one-page interpreter in the vein of the original Lisp 1.5 book [4], to one that is so detailed it specifies every bit and would probably run to hundreds of pages. Clearly, a middle course is called for. We would aim for a manageable ten-to-twenty pages.

The interpreter should be the minimal size necessary to specify the interpreter unambiguously in terms of calls to "simple" Common Lisp functions. Some judgment is required on which functions are "simple", but we assume that, in the compromising spirit of Common Lisp, some consensus could be reached. The English description of the behavior of the lowest level functions like CAR or SYMBOLP in the Steele book [5] seems adequate to prevent significant divergence among Common Lisp implementations, as is the description of middle level functions like APPEND. It is only when the complexity of something like EVAL is reached that divergence among implementations becomes a real problem. Thus, Common EVAL can use CAR and EQ in its definition without further specification, but must say explicitly how EVAL and APPLY work.

Particular attention should be paid to the behavior of the 26 Common Lisp "special forms". Differences in behavior of the special forms could be one of the worst potential problem areas. Common EVAL should certainly be detailed enough to specify the behavior of all the special forms. At the same time, we do *not* want to make Common EVAL specify the precise format of lexical environments, to give implementors freedom to optimize their representations.

The strategy is to implement EVAL and APPLY in terms of each other, a few helping functions, and list manipulation. One of these functions, NOT-CL:EVAL, provides an explicit way for a user to extend the evaluator for new data types. Most of the other functions are for manipulating lexical environments. The actual data structure used by a lexical environment is not important and is intentionally not specified by Common EVAL. But the behavior of manipulator functions *is* clearly specified.

We propose a few modifications to Common Lisp. First, that lexical environments become first class data objects. Second, that both EVAL and APPLY accept lexical environments as arguments. To minimize the impact of this on existing code, EVAL can be extended to have an optional second argument of a lexical environment [defaulting to the null environment]. APPLY can take a keyword :ENVIRONMENT to enable it to accept a lexical environment.

To illustrate the intent of our proposal more concretely, we present a short segment of Lisp code for a skeleton Common EVAL. See the appendix. Don't take this code too literally -- we mean it only to illustrate the style and the level of detail we would expect of the real Common EVAL, and as a springboard for discussion. Having actual code to discuss facilitates precise communication, one of the key features of Common EVAL.

Summary

When Lisp enthusiasts debate the merits of the language with adherents of other languages, their best ammunition is the superiority of the programming environment tools that have grown up around modern Lisp implementations. The acceptance of Common Lisp in the world will surely be dependent on the quality of its programming environments, including debuggers, editors, and program analyzers. Common Lisp has achieved some success in facilitating the portability of Lisp applications, but making the next generation of Lisp environments truly portable requires taking the next step: Common EVAL.

Appendix: Lisp code for Common EVAL

Some notes about the code:

- The LE package contains the lexical environment manipulator functions, many of which are yet to be written. If the CL community decides to provide advertised support for lexical environment functions, some of the functions here could be moved into the LISP package.
- The NOT-CL package contains miscellaneous support functions for Common EVAL.
- An implementation of eval is permitted to differ semantically from Common EVAL only by redefining NOT-CL:EVAL. This provides a well defined place for modifications to take place. Our default definition here simply errors, as would a pure CL implementation.
- Functions here which are called, not in CL, and intended to be defined by Common EVAL include:
    - APPLY
    - The lexical environment accessors.
    - The functions for handling individual special forms.
    - Closures and lexical functions are not dealt with yet.

A few of the functions for dealing with lexical variables are defined here as an example of how this code can be extended to make a full implementation of Common EVAL.

APPLY should be part of the Common EVAL spec. It does not need to be passed a LEX-ENV. APPLY makes a new LEX-ENV containing only the variables specified by the function's lambda-list. The values of those variables will initially be those values specified in the function call.

```
(in-package 'CE) ;Common Eval package. Just used so that you can
                 ;test this code without munging functions in
                 ;your CL environment.

(shadow '(eval)  'CE)
(export '(user-eval) 'CE)

(make-package 'LE)
(shadow '(boundp symbol-value) 'LE)
(export '(le::boundp le::symbol-value le::set-symbol-value) 'LE)

(make-package 'NOT-CL)

(defun eval (exp &optional lex-env)
  "currently doesn't check for lex-env fns.
   Right now, CL doesn't permit EVAL to take a 2nd arg.
   LEX-ENV defaults to the null lexical environment."
   (cond ((not-cl::self-evaluating-p exp) exp)
         ((symbolp exp) (not-cl::symbol-eval exp lex-env))
         ((consp exp)
          (cond ((symbolp (car exp))
                 (cond ((macro-function (car exp))
                        (eval (macroexpand exp) lex-env))
                       ((special-form-p (car exp))
                        (not-cl::eval-special-function-call
                                              exp lex-env))
                       ((fboundp (car exp))
                        (apply (car exp)
                               (not-cl::list-of-values (cdr exp)
                                                       lex-env)))
                       (t (user-eval exp lex-env))))
                ((and (consp (car exp))
                      (eq (car (car exp)) 'lambda))
                 (apply (car exp)
                        (not-cl::list-of-values (cdr exp)
                                                lex-env)))
                (t (user-eval exp lex-env))))
         (t (user-eval exp lex-env))))
```

```
(defun not-cl::self-evaluating-p (form)
        (cond ((or (numberp form)
                   (stringp form)
                   (characterp form)
                   (keywordp form)
                   (null form))
               form)))


(defun not-cl::symbol-eval (symbol lex-env)
  "If SYMBOL is a variable in LEX-ENV, return its value.
   Else If SYMBOL is bound, return its value,
   else error."
  (cond ((not lex-env)
         (if (boundp symbol) ;check global binding
           (symbol-value symbol)
           (error "Attempt to evaluate an unbound symbol ~S"
                  symbol)))
        ((le:boundp symbol lex-env)
         (le:symbol-value symbol lex-env))
        ((lex-env-parent lex-env)
         (not-cl::symbol-eval symbol (lex-env-parent lex-env)))
        (t (error "Symbol-eval passed bad args ~S ~S"
                  symbol lex-env))))

(defun not-cl::list-of-values (list lex-env)
   (if list
       (cons (eval (car list) lex-env)
             (not-cl::list-of-values (cdr list) lex-env))))
```

```
(defstruct lex-env variables functions tags blocks parent)
  ;only the slots variables and parent are used in the code below
  ;Full Common EVAL may require additional slots.

(defun le:boundp (sym lex-env)
  "Returns non-nil if SYM is a variable in LEX-ENV."
   (assoc sym (lex-env-variables lex-env)))

(defun le:symbol-value (sym lex-env &aux sym-val parent-lex-env)
    (setq sym-val (assoc sym (lex-env-variables lex-env)))
    (cond (sym-val
           (cdr sym-val))
          ((lex-env-parent lex-env)
           (le:symbol-value sym (car (lex-env-parent lex-env))))
          (t (error "Symbol ~S is not bound in lex-env ~S."
                    sym lex-env))))

(defun le:set-symbol-value (sym lex-env new-value &aux sym-val)
  "Finds or creates a lexical variable named SYM in lex-env and
   sets its value to NEW-VALUE. Returns NEW-VALUE."
  (setq sym-val (assoc sym (lex-env-variables lex-env)))
  (cond (sym-val (rplacd sym-val new-value))
        (t (setf (lex-env-variables lex-env)
                 (cons (cons sym new-value)
                       (lex-env-variables lex-env)))))
  new-value)
```

```
(defun not-cl::eval-special-function-call (exp lex-env)
  (case (car exp)
   (let (not-cl::eval-let exp lex-env))
   ;other special forms go here.
   (otherwise (error "not-cl::eval-special-function-call passed
                      non-implemented special form ~S" exp))))

(defun user-eval (exp &optional lex-env)
  "To extend EVAL, the user is permitted only to redefine
   this function."
      (error "Eval passed non-CL form ~S" exp))

(defun not-cl::eval-let (exp lex-env &aux vars new-lex-env
                                                    result)
  (dolist (elt (cadr exp))
    (push   (if (symbolp elt)
              (cons elt nil)
              (cons (car elt) (eval (cadr elt) lex-env)))
            vars))
  (setq new-lex-env (make-lex-env :variables (nreverse vars)
                                  :parent lex-env))
  (dolist (expr (cddr exp))
    (setq result (eval expr new-lex-env)))
  result)
#| TESTS
(= (eval 3) 3)
(eval '(setq a 2)) => Errors with SETQ special form not yet
                                                    implemented
(= (eval '(+ 2 3)) 5)
(equal (eval '(let ((a 1) (b 2) c) (list a b c))) '(1 2 nil))
(equal (eval '(let ((a 1) (b 2))
                (append (list a  b)
                        (let ((c a) (d 4))
                          (list c d)))))
       '(1 2 1 4))
(eval 'no-exist) => Errors with attempt to eval an unbound symbol

|#
```

References

[1] Linda deMichiel and Richard Gabriel.
The Common Lisp Object System.
In *ECOOP-87: European Conference on Object-Oriented Programming*. Springer Verlag, Paris, France, June, 1987.

[2] Henry Lieberman.
Steps Toward Better Debugging Tools for Lisp.
In *Proceedings of the Fourth ACM Conference on Lisp and Functional Programming*. Austin, Texas, USA, August, 1984.

[3] Henry Lieberman.
Reversible Object-Oriented Interpreters.
In *ECOOP-87: European Conference on Object-Oriented Programming*. Springer Verlag, Paris, France, June, 1987.

[4] John McCarthy.
Lisp 1.5 Programmer's Manual.
MIT Press, Cambridge, Mass., 1963.

[5] Guy Steele (and a cast of thousands).
Common Lisp: The Language.
Digital Press, Maynard, Mass., 1985.