ACM *Lisp Pointers* **V**,4 (Oct.-Dec. 1992), 11-20.

Metacircular Semantics for Common Lisp Special Forms

Henry G. Baker
Nimble Computer Corporation, 16231 Meadow Ridge Way, Encino, CA 91436
(818) 986-1436 (818) 986-1360 (FAX)

McCarthy's metacircular interpreter for Lisp has been criticized by Reynolds and others for not providing precise semantics. Unfortunately, the alternative of English prose currently favored by the ANSI X3J13 and ISO committees for the definition of Common Lisp is even less precise than a metacircular interpreter. Thus, while a system of denotational semantics á la Scheme or ML could be developed for Common Lisp, we believe that a carefully fashioned system of metacircular definitions can achieve most of the precision of denotational semantics. Furthermore, a metacircular definition is also more readable and understandable by the average Common Lisp programmer, since it is written in terms he mostly understands. Finally, a metacircular definition for Common Lisp special forms enables us to transparently customize the representation of certain "built-in" mechanisms such as `function` closures, to enable sophisticated systems like "Portable Common Loops" to become truly portable.

INTRODUCTION

Members of software standards committees should be required to take the Hippocratic Oath, which admonishes the neophyte physician to "first, do no harm!".[1] In other words, before any therapy is prescribed, first make sure that it will not make the patient worse off. We believe that the use of natural language (e.g., English) in the currently proposed Lisp standards is much worse than McCarthy's metacircular interpreter, even with its faults.[2] We suggest that a Lisp software standard has much to gain by utilizing its own uniquely powerful tools to make the language standard both readable and precise.

In the following sections, we will develop a series of definitions of various Common Lisp special forms in terms of one another. While these definitions, by themselves, will not pin down the semantics of Common Lisp completely, they can be used in conjunction with a rough understanding of Common Lisp semantics to understand the less usual cases of interactions of the various features.

We use `defmacro` to define a special form in terms of other forms, and perhaps in terms of simpler versions of the form itself. This is done not just because such definitions can be entered and quickly checked using a standard Common Lisp implementation, but also because we feel that the Common Lisp "macro" is the santioned mechanism for adding new "special forms". While this view of special forms is not evident from CLtL2 [Steele90], it should be obvious by the end of this paper. In short, the choice of which "macros" are "special forms" is just as arbitrary as the choice of a axes in a coordinate system for the Cartesian X-Y plane—e.g., some sets of macros are "linearly independent", and some sets of macros "span" the space of special forms.

Some of our emulations may only be approximate, in the sense that certain syntactic variations are not supported, and certain error conditions are not recognized. <u>These emulations are meant to be only a starting point for a serious effort in pinning down the semantics of Common Lisp</u>, and significant additional effort will be required to complete this task.[3]

The "Portable Common Loops" ("PCL") version of the Common Lisp Object System (CLOS) exemplifies the need for a more reflexive view of Common Lisp special forms. PCL does not quite live up to its name, since it needs to diddle the representation of function closures, which is different in every Common Lisp implementation. Through the techniques we exhibit here, a truly portable version of PCL could be produced, thereby eliminating the need to include CLOS in a Common Lisp standard.

---

[1]The ancient Greek's way of saying "if it ain't broke, don't fix it!".

[2]The peculiar penchant of modern software standards committees to couch their pronouncements in English prose is symptomatic of an epidemic of *lawyer envy* which is sweeping the computer field. Lawyers understand the fine art of language *obfuscation*, in which a simple thing is made complex for the single purpose of providing employment for other lawyers who then interpret the language. It's bad enough that there are already more lawyers than engineers in the United States, without having these few remaining engineers talking and acting like lawyers, as well.

[3]The emulations below have not undergone extensive testing; please contact the author regarding any errors.

ACM *Lisp Pointers* **V**,4 (Oct.-Dec. 1992), 11-20.

LAZY EVALUATION OF "IF"

The Common Lisp special form `if` is often thought to be *primitive*, in the sense that it cannot be defined in terms of other special forms. Of course, `if` can be defined in terms of `cond` or `case`, so one of these macros could have been chosen as the primitive conditional special form rather than `if`. Nevertheless, the lazy evaluation of the "then" and the "else" arms of if can be emulated by means of `lambda`, as the following macro shows:

```
(setf (get 't 'select-function) #'(lambda (x y) (funcall x))
      (get 'nil 'select-function) #'(lambda (x y) (funcall y)))
```

```
(defmacro if (be te &optional (ee ''nil))
  `(funcall (get (not (not ,be)) 'select-function)
     #'(lambda () ,te)
     #'(lambda () ,ee)))
```

CIRCULAR ENVIRONMENTS OF "LABELS" EMULATED BY "FLET" AND "SETQ"

It is generally believed that the circular environments of `labels` cannot be obtained by means of `flet`. This is incorrect, as the following emulation (reminiscent of Scheme) shows. With a more sophisticated macro-expansion, this emulation can be optimized into production-quality code.

```
(defmacro labels (fns &body forms)
  (let* ((fnames (mapcar #'car fns))
         (nfnames (mapcar #'(lambda (ignore) (gensym)) fnames))
         (nfbodies (mapcar #'(lambda (f) `#'(lambda ,@(cdr f))) fns)))
    `(let ,(mapcar #'(lambda (nf) `(,nf #'(lambda () ()))) nfnames)
       (flet ,(mapcar #'(lambda (f nf) `(,f (&rest a) (apply ,nf a)))
                      fnames nfnames)
         (flet ,fns
           (progn ,@(mapcar #'(lambda (f nf) `(setq ,nf #',f))
                            fnames nfnames))
           ,@forms)))))
```

CIRCULAR ENVIRONMENTS OF "LABELS" EMULATED BY "FLET" ALONE

Less obvious is the fact that `labels` can be emulated by `flet` without the use of side-effects. Our technique is based on the idea of the "Y combinator", discussed at length in [Gabriel88]. With a more sophisticated macro-expansion based on topologically sorting the `labels` call graph, this technique can approach production quality.

```
(eval-when (compile)
  (defun iota-list (n &optional (m 0))
    (if (zerop n) nil `(,m ,@(iota-list (1- n) (1+ m))))))
```

```
(defmacro labels (fns &body forms)
  (let* ((fnames (mapcar #'car fns))
         (fnvec (gensym))
         (findicies (iota-list (length fns)))
         (fbodies (mapcar #'(lambda (f i)
                              `(,f (&rest a) (apply (svref ,fnvec ,i) a)))
                          fnames findicies))
         (fdecls `(declare (inline ,@fnames)))
         (nfbodies (mapcar #'(lambda (f)
                               `#'(lambda (,fnvec ,@(cadr f))
                                    (flet ,fbodies ,fdecls ,@(cddr f))))
                           fns)))
    `(let ((,fnvec (vector ,@nfbodies)))
       (flet ,fbodies ,fdecls ,@forms))))
```


"FLET" EMULATED BY "MACROLET"

The essence of the lexical scoping of `flet` and `labels` function names is captured by the `macrolet` special form, as the following emulation demonstrates:

```
(defmacro flet (fns &body forms)
  (let* ((fnames (mapcar #'car fns))
         (nfnames (mapcar #'(lambda (ignore) (gensym)) fnames))
         (nfbodies (mapcar #'(lambda (f) `#'(lambda ,@(cdr f))) fns)))
    `(let ,(mapcar #'(lambda (nfn nfb) `(,nfn ,nfb))
                   nfnames nfbodies)
       (macrolet
         ,(mapcar #'(lambda (f nf) `(,f (&rest a) `(apply ,',nf ,a)))
                  fnames nfnames)
         ,@forms))))
```

"LET" EMULATED BY "LAMBDA"

The classical emulation of `let` by `lambda` is mentioned in many Common Lisp textbooks, and we include it here because it illustrates several important points. First, this simulation indicates that the order of evaluation of arguments to a function call is the same as the order of evaluation of the forms of a `let`-expression. However, even though CLtL2 stresses the fact that the order of evaluation of the arguments in a function call is not defined, the order of evaluation of the forms of a `let`-expression *is* defined. Thus, the emulation of `let` by `lambda` pins down argument evaluation order to be the same in both cases. Variable declarations happen to work correctly for this emulation.

```
(defmacro let (vs &body forms)
  `(funcall #'(lambda ,(mapcar #'car vs) ,@forms) ,@(mapcar #'cadr vs)))
```

"LET*" EMULATED BY "LET"

The "sequential" nature of `let*` is achieved by binding only one form at a time and nesting the binding forms, as the following code shows. This emulation does not handle declarations correctly, because the parsing and interpretation of declarations can be a real pain (declarations can be easily and efficiently parsed using the techniques of [Baker91PP]). The correct handling of the declarations is extremely important, however, as it is one of the major sources of confusion in CLtL2 [Steele90].

```
(defmacro let* (vs &body forms)
  (if vs `(let (,(car vs)) (let* ,(cdr vs) ,@forms))
    `(let () ,@forms)))
```

"LET" EMULATED BY "LET*"

The emulation of `let` by `let*` is important, because it proves the sequentiality of the evaluation of the forms.

```
(defmacro let (vs &body forms)
  (let ((nvs (mapcar #'(lambda (ignore) (gensym)) vs)))
    `(let* ,(mapcar #'(lambda (v nv) `(,nv ,(cadr v))) vs nvs)
       (let* ,(mapcar #'(lambda (v nv) `(,(car v) ,nv)) vs nvs)
         ,@forms))))
```

"PROGN" EMULATED BY "LET*"

The emulation of `progn` by `let*` is important, because it proves that each form of a `progn` is completely evaluated to a value before the commencement of the evaluation of the next form:

```
(defmacro progn (&body forms)
  (if forms `(let* ,(mapcar #'(lambda (e) `(,(gensym) ,e)) (butlast forms))
    ''nil))
```


"BLOCK/RETURN-FROM" EMULATED BY "CATCH/THROW"

We will show that only one of the three non-local exit mechanisms `block/return-from`, `tagbody/go`, `catch/throw` is required to be primitive, by showing how to emulate any two in terms of the third.[4] We first emulate `block/return-from` in terms of `catch/throw`. We map the block name into the name of a lexical variable which will hold the unique tag which distinguishes this dynamical block from any other. If trivial `return-from`'s are optimized away, then this emulation can be quite efficient.

```
(defmacro return-from (bname exp)
  (let ((tagname (block-to-tagname bname)))
    `(throw ,tagname ,exp)))

(defmacro block (bname &body forms)
  (let ((tagname (block-to-tagname bname)))
    `(let ((,tagname (list nil))) ; Unique cons cell used as catch tag.
       (catch ,tagname (progn ,@forms)))))
```

"BLOCK/RETURN-FROM" EMULATED BY "TAGBODY/GO"

The emulation of `block/return-from` using `tagbody/go` is more difficult than when using `catch/throw` because we must communicate the returned multiple values using a lexical variable specifically allocated for this purpose:

```
(defmacro return-from (bname exp)
  (let ((vname (block-to-valuesname bname))
        (labelname (block-to-labelname bname)))
    `(progn (setq ,vname (multiple-value-list ,exp))
            (go ,labelname))))

(defmacro block (bname &body forms)
  (let ((vname (block-to-valuesname bname))
        (labelname (block-to-labelname bname)))
    `(let ((,vname nil))
       (tagbody (setq ,vname (multiple-value-list (progn ,@forms)))
         ,labelname)
       (values-list ,vname))))
```

"TAGBODY/GO" EMULATED BY "CATCH/THROW"

The emulation of `tagbody/go` by `catch/throw` is considerably less obvious than the emulation of `block/return-from`. This is because `tagbody` defines a number of different labels rather than a single block name, and because the parsing of the `tagbody` body is considerably more complicated. The various segments of the `tagbody` are emulated by a `labels` nest of mutually recursive functions, which are forced to all execute at the correct dynamic depth by means of a "trampoline". If the implementation implements the "tail recursion" optimization for functions which have no arguments and return no values, and if the simpler cases of `go`'s are optimized away, then this emulation can be quite efficient.

```
(defmacro go (label)
  (let ((name (label-to-functionname label)))
    `(throw ,name #',name)))
```

---

[4]Of the three, we strongly recommend that `catch/throw` be considered the most primitive mechanism, because basing `tagbody/go` and `block/return-from` on `catch/throw` makes absolutely clear the fact that Common Lisp does not and can not have Scheme-like first-class continuations.


```
(defmacro tagbody (&body body)
  (let* ((init-tag (gensym)) (go-tag (gensym)) (return-tag (gensym))
         (functions
          (mapcon
           #'(lambda (seq &aux (label (car seq)) (s (cdr seq)))
               (when (atom label)
                 (let ((p (position-if #'atom s)))
                   `((,(label-to-functionname label) ()
                        ,@(subseq s 0 (or p (length s)))
                        ,(if p `(,(label-to-functionname (elt s p)))
                             `(throw ,return-tag 'nil)))))))
           `(,init-tag ,@body))))
    `(let* ((,go-tag (list nil)) (,return-tag (list nil))
            ,@(mapcar #'(lambda (f) `(,(car f) ,go-tag)) functions))
       (catch ,return-tag
         (labels ,functions
           (let ((nxt-label #',(caar functions)))
             (loop5 (setq nxt-label (catch ,go-tag (funcall nxt-label)))))))))))
```

"CATCH/THROW" EMULATED BY "TAGBODY/GO" AND SPECIAL VARIABLES

The emulation of `catch`/`throw` requires the use of some sort of stack to indicate the dynamic nesting level of the call-return stack. Common Lisp dynamic or "special" variables can be thought of as being "saved" and "restored" from a hidden stack sometimes called the "specpdl". Therefore, we can use the dynamic nesting capabilities of special variables to keep track of the current list of `catch` tags and "catchers", which is organized as a Lisp `assoc` list. Notice that certain issues involved in the implementation of `catch`/`throw` are made completely obvious—that the tag is evaluated, that the `throw` expression is evaluated before the stack is unwound, that the tag is looked up using an `eq` test,[6] that the tag is looked up before the stack is unwound, etc.

```
(defparameter *catchers* nil)

(defmacro throw (tag exp)
  (let ((vtag (gensym)) (vexp (gensym)))
    `(let ((,vtag ,tag) (,vexp (multiple-value-list ,exp)))
       (funcall (cdr (assoc ,vtag *catchers* :test #'eq)) ,vexp))))

(defmacro catch (tag exp)
  (let ((lbl (gensym)) (vals (gensym)))
    `(let ((,vals nil))
       (tagbody
         (setq ,vals (progv '(*catchers*)
                            (list
                             (cons (cons ,tag
                                         #'(lambda (vs) (setq ,vals vs)
                                                          (go ,lbl)))
                                   (symbol-value '*catchers*)))
                       (multiple-value-list ,exp))
         ,lbl)
       (apply #'values ,vals))))
```

"CATCH/THROW" EMULATED BY "BLOCK/RETURN-FROM"

The emulation of `catch`/`throw` by `block`/`return-from` is a bit easier than by `tagbody`/`go`, because we can communicate the returned values a bit more directly.

[5] Many Lisps define `loop` in terms of `tagbody`; we, however, think of it as a trivial tail-recursive function.

[6] `catch`/`throw` is the only mechanism in Common Lisp which defaults to `eq` instead of `eql`; this use violates the "object identity" [Baker93] of the `catch` tag object.


```
(defmacro throw (tag exp)
  (let ((vtag (gensym)) (vexp (gensym)))
    `(let ((,vtag ,tag) (,vexp (multiple-value-list ,exp)))
       (funcall (cdr (assoc ,vtag *catchers* :test #'eq)) ,vexp))))

(defmacro catch (tag exp)
  (let ((tgnm (gensym)))
    `(block ,tgnm
       (progv '(*catchers*)
              (list
               (cons (cons ,tag
                           #'(lambda (vs)
                               (return-from ,tgnm (values-list vs))))
                     (symbol-value '*catchers*)))
         (multiple-value-list ,exp)))))
```

"PROGV" (DYNAMIC BINDING) EMULATED BY "UNWIND-PROTECT"

The Common Lisp special form `unwind-protect` allows the programmer to perform certain actions any time the form is exited, whether by normal exit, or by a non-local exit form such as `throw`, `return-from` or `go`. In particular, the "shallow-binding" [Baker76SB] implementation of `progv` dynamic variables can be emulated using `unwind-protect`.

Unfortunately, our emulation is incomplete, because it cannot handle the case of special variables which are referenced without the use of `symbol-value`. Many of these cases could be handled using `symbol-macrolet`, but not all. In particular, the use of the same form `setq` for both lexical and dynamic variables in Common Lisp is reprehensible.

```
(defconstant *unbound-value* (list nil))

(defun msymbol-value (var)
  (if (boundp var) (symbol-value var) *unbound-value*))

(defun mset (var val)
  (if (eq val *unbound-value*) (makunbound var) (set var val)))

(defmacro progv (syms vals &body forms)
  (let* ((vsyms (gensym)) (vvals (gensym)) (vovals (gensym)))
    `(let* ((,vsyms ,syms)
            (,vvals ,vals)
            (,vovals ,(mapcar #'msymbol-value ,vsyms))
       (unwind-protect
         (progn (mapc #'mset ,vsyms ,vvals)
                (mapc #'makunbound
                      (subseq ,vsyms (min (length ,vsyms) (length ,vvals))))
                ,@forms)
         (mapc #'mset ,vsyms ,vovals)))))
```

EMULATE "UNWIND-PROTECT" AND SPECIAL VARIABLES

The code above shows how to emulate special variables using `unwind-protect`. We now show how to emulate `unwind-protect` itself using an explicit "specpdl" stack.

```
(defparameter *specpdl* (make-array 100 :adjustable t :fill-pointer 0))

(defun unwind-to (n)
  (dotimes (i (- n (fill-pointer *specpdl*)) nil)
    (funcall (vector-pop (symbol-value '*specpdl*)))))
```



```
(defmacro unwind-protect (form &body forms)
  `(multiple-value-prog1
     (progn (vector-push-extend #'(lambda () ,@forms)
                                (symbol-value '*specpdl*))
            ,form)
     (funcall (vector-pop (symbol-value '*specpdl*)))))

(defmacro return-from (bname exp)
  (let ((vexp (gensym)))
    `(let ((,vexp (multiple-value-list ,exp)))
       (unwind-to ,(blockname-to-levelname bname))
       (return-from7 ,bname (values-list ,vexp)))))

(defmacro block (bname &body forms)
  `(let ((,(blockname-to-levelname bname)
          (fill-pointer (symbol-value '*specpdl*))))
     (block ,bname ,@forms)))

;;; catch/throw and tagbody/go are similarly tagged with their dynamic level.
```

EMULATE MULTIPLE VALUES

The emulation of multiple values can be insightful, because it can highlight a possible optimization which avoids the execution of those forms whose values are not wanted. In other words, the `values` function should be a special form which evaluates only those arguments necessary to fulfill the requested number of values.[8]

```
(defparameter *mv-nbr-expected* 1) ; Usually 1 value expected.

(defparameter *mv-vals* (make-array multiple-values-limit))

(defmacro multiple-value-list (form)
  (let ((val1 (gensym)))
    `(progv '(*mv-nbr-expected*) (list multiple-values-limit)
       (let ((,val1 ,form)) ; Receive the first value here.
         (if (= (symbol-value '*mv-nbr-expected*) multiple-values-limit)
           (list ,val1)
           (coerce (subseq *mv-vals* 0 (symbol-value '*mv-nbr-expected*))
                   'list))))))

(defun values (&rest args)
  (dotimes (i (setf (symbol-value '*mv-nbr-expected*)
                    (min (symbol-value '*mv-nbr-expected*) (length args)))
              (car args))
    (setf (aref *mv-vals* i) (elt args i))))

(defmacro multiple-value-prog1 (exp &rest forms)
  (let ((valn (gensym)))
    `(let ((,valn (multiple-value-list ,exp)))
       (progn ,@forms (apply #'values ,valn)))))

(defmacro multiple-value-call (fn &body forms)
  `(apply ,fn
     (append ,@(mapcar #'(lambda (fm) `(multiple-value-list ,fm)) forms))))

(defmacro mvprogn (&body forms)
  (if body `(progn (progv '(*mv-nbr-expected*) '(0)
                     (progn ,@(butlast forms)))
                   ,@(last forms))
    ''nil))
```

[7] I warned you that these definitions were metacircular!

[8] There are other reasons for making `values` into a special form instead of a function—e.g., compiler optimizations.


```
(defmacro mvif (be te &optional (ee ''nil))
  `(if (progv '(*mv-nbr-expected*) '(1) ,be) ,te ,ee))
```

EMULATE "THE" USING "LET" AND "DECLARE"

The emulation of the `the` special form emphasizes the fact that there is a run-time type test which must be passed in order for the program to proceed. Of course, a clever compiler can eliminate the run-time test if it can prove that it will always succeed—e.g., the `gcd` function always returns an integer if it returns at all.

```
(defmacro the (typ exp)
  (if (and (consp typ) (eq (car typ) 'values))
    (let ((vals (gensym)))
      `(let ((,vals (multiple-value-list ,exp)))
         (assert (= (length ,vals) ,(length (cdr typ))))
         ,@(mapcar #'(lambda (typ i) `(assert (typep (elt ,vals ,i) ',typ)))
                   (cdr typ) (iota-list (length (cdr typ))))
         (values-list ,vals)))
    (let ((val (gensym)))
      `(let ((,val ,exp))
         (assert (typep ,val ',typ))
         (let ((,val ,val)) (declare (type ,typ ,val))
           ,val)))))
```

PERFORM "CELL INTRODUCTION" USING "SYMBOL-MACROLET"

In order to correctly handle mutable lexical variables which are captured by `function` closures, many Lisps perform "cell introduction" [Sandewall74] [Kranz86], in which each mutable lexical variable is transformed into another lexical variable which is immutably bound to a newly-consed mutable "cell", which holds the current value of the original variable.

```
(defun make-cell (v &aux (c (gensym)))
  (setf (symbol-value c) v)
  c)

(defmacro setq (pairs)
  `(setf ,@pairs))

(defmacro let (vs &body forms)
  `((lambda ,(mapcar #'(lambda (b) (xform-name (car b))) vs)
      (symbol-macrolet
        ,(mapcar #'(lambda (b)
                     `(,(car b) '(symbol-value ,(xform-name (car b))))
                 vs)
        ,@forms))
    ,@(mapcar #'(lambda (b) `(make-cell ,(cadr b))) vs)))

(defmacro function (fn)
  (if (atom fn) `#',fn
    `#'(lambda ,(cadr fn)
         ((lambda ,(mapcar #'xform-name (cadr fn))
            (symbol-macrolet
              ,(mapcar #'(lambda (v)
                           `(,v '(symbol-value ,(xform-name v))))
                       (cadr fn))
              ,@forms))
          ,@(mapcar #'(lambda (v) `(make-cell ,v)) (cadr fn))))))
```

EMULATE "FUNCTION" USING "MACROLET"

Our final emulation will show how the Common Lisp `function` special form can be customized by the user to provide a specialized representation or specialized storage allocation for `function` closures. Such customization could be used within the "PCL" implementation of CLOS to make this implementation both efficient and portable.

If PCL can be efficiently and portably emulated in Common Lisp-84, then it can become a straight-forward library, and need not be considered part of the essential core of Common Lisp.

Our emulation will utilize some of the previous emulations. In particular, we assume that `block`/`return-from` and `tagbody`/`go` have already been emulated by `catch`/`throw`; these emulations eliminate the need to close over lexical `block` names and `tagbody` labels. Furthermore, we assume that cells have already been introduced for any mutable lexical variables.

We will represent the free variables and free functions of the function closure in a simple vector. We could, however, have constructed a different brand-new function-closure extension of the function structure for each new invocation of the macro `function`; this would allow for a specialized representation for each different occurrence of `function` in the user's program. The `make-function-closure` function which generates new function-closure instances can then be specialized for each occurrence, and may perform different kinds of allocations—e.g., stack allocation [Baker92] versus heap allocation—for each new function-closure instance.

```
(defstruct function ; This defstruct is predefined by the implementation.
  lambda)

(defstruct (function-closure (:include function))
  acquaintances)

(defmacro function (lexp)
  (let* ((gvars (free-globals lexp))
         (fvars (free-lexicals lexp))
         (ffns (free-functions lexp))
         (acql (gensym)))
    `(make-function-closure
      :acquaintances (vector ,@fvars ,@(mapcar #'(lambda (f) `#',v) ffns))
      :lambda
      #'(lambda (,acql ,@(cadr lexp))
          (symbol-macrolet ; handle free lexical and global variable names.
           (,@(mapcar #'(lambda (v i) `(,v '(svref ,acql ,i)))
                      fvars (iota-list (length fvars)))
            ,@(mapcar #'(lambda (v) `(,v '(symbol-value ',v))) gvars ))
           (macrolet ; handle free function names.
            ,(mapcar
              #'(lambda (f i)
                  `(,f (&rest a)
                       (list* 'funcall '(svref ,acql ,(+ i (length fvars))) a)))
              ffns (iota-list (length fns))))
            ,@(cddr lexp))))))
```

CONCLUSIONS

We have shown how a number of Common Lisp special forms can be emulated using other Common Lisp special forms. Taken as a whole, these emulations can provide a relatively precise definition of the Common Lisp language in terms of classical lambda-calculus notions of alpha-renaming, beta-reduction and eta-reduction. Of course, a complete definition would deal with more issues, such as the precise parsing and handling of declarations, and the precise meanings of `symbol-macrolet` and `macrolet`, which define the meaning of lexical variable and lexical function shadowing, respectively.

Although we have shown the mutual interdefinability of three different non-local exit constructs in Common Lisp, we strongly recommend that any formal semantics for Common Lisp utilize `catch`/`throw` as its most primitive non-local exit mechanism, since `catch`/`throw` cannot create Scheme-like first-class continuations, and therefore cannot get into the major semantical and implementational problems (particularly painful in parallel systems) of first-class continuations.

Our emulations raise as many questions as they answer, and provide illumination to some dark corners of Common Lisp. The inability to completely specify certain operations points up some important holes in the semantics of Common Lisp, particularly in the area of macro-expansions, lexical `block` names and lexical `tagbody` labels.

REFERENCES

Baker, Henry G. "Shallow Binding in Lisp 1.5". *CACM 21*,7 (July 1978), 565-569.

Baker, Henry G. "Pragmatic Parsing in Common Lisp". *ACM Lisp Pointers IV*,2 (Apr.-June 1991),3-15.

Baker, Henry G. "Equal Rights for Functional Objects or, The More Things Change, The More They Are the Same". ACM *OOPS Messenger 4*,4 (Oct. 1993), 2-27.

Baker, Henry G. "CONS Should not CONS its Arguments, or, a Lazy Alloc is a Smart Alloc". ACM *Sigplan Not. 27*,3 (March 1992),24-35.

Gabriel, R.P. The Why of Y". *Lisp Pointers 2*,2 (Oct.-Dec. 1988), 15-25.

Kiczales, G., *et al*. *The Art of the Metaobject Protocol*. MIT Press, Camb., MA, 1991.

Kranz, D., *et al*. "Orbit: An Optimizing Compiler for Scheme". *Sigplan'86 Symp. on Compiler Constr.*,219-233.

McCarthy, J. "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I". *CACM 3*,4 (1960), 184-195.

Reynolds, J. "Definitional Interpreters for High-order Programming Languages". *Proc. ACM Nat'l. Conv.* (1972), 717-740.

Steele, Guy L. *Common Lisp, The Language; 2nd Ed*. Digital Press, Bedford, MA, 1990,1029p.

