Expansion-Passing Style:
Beyond Conventional Macros

By

R. K. Dybvig, D. P. Friedman and C. T. Haynes
Computer Science Department
Indiana University
Bloomington, IN 47405

TECHNICAL REPORT NO. 195

Expansion-Passing Style:
Beyond Conventional Macros

by

R. K. Dybvig, D. P. Friedman and C. T. Haynes
Indiana University

May, 1986

This material is based on work supported by the National Science Foundation under grant numbers DCR 83-03325 and DCR 85-01277.

This report to appear in the Proceedings of the 1986 ACM Symposium on LISP and Functional Programming.

Expansion-Passing Style: Beyond Conventional Macros

R. Kent Dybvig, Daniel P. Friedman, Christopher T. Haynes

Computer Science Department
Indiana University
Bloomington, Indiana 47405

Abstract

The traditional macro expansion facility inhibits several important forms of expansion control. These include selective expansion of subexpressions, expansion of subexpressions using modified expansion functions, and expansion of application and identifier forms. Furthermore, the expansion algorithm must treat every special form as a separate case. The result is limited expressive power and poor modularity. We propose an alternate facility that avoids these problems, using a technique called *expansion-passing style* (EPS). The power of this technique is illustrated with several examples, including a set of debugging tools. Most Lisp systems may be easily adapted to employ this technique.

1. Introduction

Lisp systems generally include a facility that allows for convenient extension of the source language syntax. This facility is implemented by expanding *syntactic extensions* (also called *macros*) into the base language (special forms) of the Lisp system. There are several advantages to source-level expansion over the use of a special interpreter to provide new syntatic forms. First, it eliminates the need for extra layers of interpretation; source-code expansion need only be performed once, resulting in greater efficiency. Second, it is easier to make modifications with syntactic extensions than by writing new interpreters. Third, the semantics of a language obtained by "sugaring" the syntax of an existing well understood language with syntactic transformations is more easily understood and verified than the semantics of a language obtained by writing a new interpreter. Finally, the extended language is easily ported to another host that supports the same base language.

In the next section we review the conventional syntactic extension facility. We then present examples of several forms of syntactic extension that are not possible using the traditional mechanism. In the following section we introduce a facility with the flexibility to implement these extensions. Most Lisp systems may be easily adapted to employ this facility. As a substantial and practical example of the new facility's power, we then present debugging tools that are easily obtained using syntactic extension. Finally, we note a few problems with this facility that merit further investigation. The code that follows is expressed in Scheme [2,3].

2. Background and Motivation

Syntactic transformations of Lisp programs are most conveniently and efficiently performed by manipulating expressions prior to evaluation. Provision for this is easily made by adding a preprocessor to eval, which we call expand. Such decoupling of the evaluation mechanism from the syntactic extension mechanism has the advantage of simplifying the underlying compiler or interpreter and making the syntactic transformations independent of the implementation.

Syntactic extension is performed by invoking expansion functions when certain keywords are present in the car position of a form (expression). Such keywords are associated with expansion functions in some manner, such as a `*MACRO*` property. Traditionally, when the expand function encounters a form with a macro keyword in its car position, the entire form is passed to the associated expansion function. The expansion function then returns a new form, obtained by transforming the old one, that is then expanded in place of the old form. See Figure 1 for a typical expand function using this protocol. Each special form (there are 24 in Common Lisp [6]) must in general be treated as a special case.

Consider the expansion of `let` expressions of the form

(`let ((`id_1 exp_1`) ... (`id_n exp_n`))`
  $body_1$ `...` $body_m$`)`

into the equivalent `lambda` applications of the form

`((lambda (`id_1 `...` id_n`)` $body_1$ `...` $body_m$`)`
 exp_1 `...` exp_n`)`.

This material is based on work supported by the National Science Foundation under grant numbers DCR 85-01277 and DCR 83-03325.

To appear in the 1986 ACM Symposium on LISP and Functional Programming.

```
(define old-style-expand
  (lambda (x)
    (cond
      ((symbol? x) x)
      ((not (pair? x)) x)
      ((macro? (car x))
       (old-style-expand ((get (car x) '*MACRO*) x)))
      ((eq? (car x) 'lambda)
       `(lambda ,(cadr x)
          ,@(map old-style-expand (cddr x))))
      ((eq? (car x) 'quote) x)
       other special forms
       ...
      (else (map old-style-expand x)))))

(define macro?
  (lambda (x)
    (and (symbol? x) (get x '*MACRO*))))
```

Figure 1. Traditional syntactic expansion mechanism.

Using put, we define this expansion as follows:

```
(put 'let '*MACRO*
  (lambda (x)
    `((lambda ,(map car (cadr x)) ,@(cddr x))
      ,@(map cadr (cadr x))))).
```

Though this mechanism provides considerable power at low cost, it has several problems. In the first place, it does not permit the expansion of application and identifier forms, which may be useful. For example, we may wish to obtain curried or call-by-name semantics by syntactic transformation of an uncurried call-by-value language. Currying requires that applications and abstractions with more than one argument be transformed into nested applications and abstractions of one argument; thus `(f a b)` would become `((f a) b)` and `(lambda (a b) ...)` would become `(lambda (a) (lambda (b) ...))`. To obtain call-by-name semantics, it suffices to

- replace every application argument *e* that is not an identifier by `(box (lambda () e′))`, where *e′* is obtained by expanding *e*,
- replace every identifier reference *id* that is not an argument by `((unbox id))`, and
- replace every assignment statement of the form `(set! id e)`, where *e* is not an identifier, by

  `(set-box! id (let ((x e′)) (lambda () x)))`.

The functions `box`, `unbox`, and `set-box!` create, dereference, and change one-celled objects. Call-by-need may be achieved with a more complex application expansion.

In the second place, the re-expansion of forms returned by expansion functions is usually, but not always, desirable. Sometimes it is important that either the new top level expression or some of its subexpressions not be expanded further, or that a different expansion function be applied to these expressions. A case in point is the above call-by-name expansion, in which it is not appropriate to perform the expansion on the application introduced by expanding an identifier reference, or on the lambda expression introduced by transforming an assignment expression. In other cases, such as some of the debugging tools to be presented later, it is important that an expansion be performed only on selected subexpressions.

Finally, consider the problem of defining a syntactic extension that allows the definition of new syntactic extensions that are only effective within its lexical scope, in the manner of `macrolet` [6]. This requires that the body be expanded with an augmented expander obtained by extending the current expander so that it recognizes the new form.

3. Expansion-Passing Style

In the last section we demonstrated that expanders should have control over the further expansion of the forms they return. This is analogous to the need for a function to have control over how the value that it returns is used to continue the computation. Continuation-passing style (CPS) may be used to give the function this power [4,5,7].

In the `macro?` line of `old-style-expand` (Figure 1), the recursive call occurs in tail recursive position. Thus if `old-style-expand` were written in CPS, the macro expansion function could simply be passed `old-style-expand` as its continuation argument and the value returned by the expansion function would require no further expansion. This motivates us to modify the traditional macro protocol:

> *Expansion functions take two arguments, the expression to be expanded and an expansion function that must be applied to any form that is to be further expanded.*

We call such expansion functions *expanders* and refer to this protocol as *expansion-passing style* (EPS).

EPS gives expanders control over whether the entire transformed expression is to be expanded further, which proper subexpressions are to be expanded, when the expansions are to be done, and even what expander is to be used for further expansion. In most cases the expander that is passed will be used, but other alternatives are possible.

It is a simple matter to transform a macro expansion function obeying the traditional protocol into an expander.

```
(define macro-to-expander
  (lambda (m)
    (lambda (x e) (e (m x) e))))
```

(Where no ambiguity results, we use the identifiers `x` and `e` for form and expander arguments, respectively.)

The system `expand` function is now defined in terms of an initial expander that dispatches on the type of form to be expanded. We also define `expand-once`, which does only one level of expansion, and is useful for debugging expanders. See Figure 2. Neither `expand` nor `initial-expander` is directly recursive.

```
(define expand
  (lambda (x)
    (initial-expander x initial-expander)))

(define initial-expander
  (lambda (x e)
    (let ((e1 (cond
                ((symbol? x) *identifier-expander*)
                ((not (pair? x)) (lambda (x e) x))
                ((expander? (car x))
                 (get (car x) '*EXPANDER*))
                (else *application-expander*))))
      (e1 x e))))

(define expand-once
  (lambda (x)
    (initial-expander x (lambda (x e) x))))

(define *identifier-expander* (lambda (x e) x))

(define *application-expander*
  (lambda (x e)
     (map (lambda (x) (e x e)) x)))

(define install-expander
  (lambda (keyword function)
    (put keyword '*EXPANDER* function)))

(define expander?
  (lambda (x)
    (and (symbol? x) (get x '*EXPANDER*))))
```

Figure 2. Basic EPS functions.

It is no longer necessary to include each of the special forms, such as `lambda` and `quote`, in the system expand function. It is only necessary to associate expanders with the special form keywords in the same way that new syntactic extensions are defined. With the traditional macro mechanism this is impossible, since it is essential that (1) some sub-parts not be expanded (for example, the formal parameter list of `lambda` or the literal part of `quote`), and (2) the entire form not be re-expanded (as the expansion process would not terminate). But expanders can control further expansion:

```
(install-expander 'lambda
  (lambda (x e)
    `(lambda ,(cadr x)
       ,@(map (lambda (x) (e x e)) (cddr x)))))

(install-expander 'quote (lambda (x e) x)).
```

Factoring the special forms out of the expander increases modularity, encourages custom variations on the expander, and allows redefinition of special form expanders.

Figure 3 illustrates the use of `install-expander` to implement a conventional macro definition interface. The essential features of `defmacro` [6] are supported. (Most of the code is dedicated to destructuring the arguments to the macro.)

```
(install-expander 'defmacro
  (lambda (x e)
    (let ((keyword (cadr x))
          (pattern (caddr x))
          (body (cadddr x)))
      (e `(install-expander ',keyword
             (make-macro ',pattern ',body))
         e))))

(define make-macro
  (lambda (pat body)
    (eval
      `(lambda (x e)
         (e (let ,(destructure pat '(cdr x) '())
                 ,body)
            e)))))

(define destructure
  (lambda (pat arg bindings)
    (cond
      ((null? pat) bindings)
      ((symbol? pat) (cons `(,pat ,arg) bindings))
      ((pair? pat)
       (destructure (car pat) `(car ,arg)
         (destructure (cdr pat) `(cdr ,arg)
           bindings))))))
```

Figure 3. `defmacro` expander.

Currying of applications and lambda abstractions is now straightforward; see Figure 4. We might expect the second `cond` clause in the redefinition of `*application-expander*` to be

```
((null? (cddr x)) (e `(,(car x) ,(cadr x)) e)).
```

However, this would loop indefinitely. Invoking `e` on a two element application will cause the same clause to be reentered.

The call-by-name transformations outlined in the last section may now be obtained by changing the meaning of application, identifier and `set!` forms; see Figure 5. If `let` were used in the expansion of `set!`, the `let` form would have to be further expanded (assuming `let` is implemented as a syntactic extension); but this expansion would result in the by-name application transformation being applied to the application resulting from the `let` expansion, which would be an error. Thus when writing the `set!` expansion it is necessary to manually expand the `let`. Sometimes great care is required when writing expanders!

The factorial function may be defined in a way that dramatizes the normal order evaluation semantics:

```
(define Y
  (lambda (f)
    ((lambda (x) (f (x x)))
     (lambda (x) (f (x x))))))
```

```
(define factorial
  (Y (lambda (f)
       (lambda (x)
         (let ((a (name-zero? x))
               (b (name-* x (f (name-sub1 x)))))
           (if a 1 b))))))
```

where `name-zero?`, `name-*` and `name-sub1` are versions of the `zero?`, `*` and `sub1` primitives that force evaluation of their arguments. This only works because the `let` form expands into an application that is then transformed to delay evaluation of the `let` bindings.

```
(set! *application-expander*
  (lambda (x e)
    (cond
      ((null? (cdr x)) `(,(e (car x) e)))
      ((null? (cddr x))
       `(,(e (car x) e) ,(e (cadr x) e)))
      (else
       (e `((,(car x) ,(cadr x)) ,@(cddr x)) e)))))

(install-expander 'lambda
  (lambda (x e)
    (let ((args (cadr x)) (body (cddr x)))
      (cond
        ((null? args)
         `(lambda ()
            ,@(map (lambda (x) (e x e)) body)))
        ((null? (cdr args))
         `(lambda ,args
            ,@(map (lambda (x) (e x e)) body)))
        (else
         `(lambda (,(car args))
            ,(e `(lambda ,(cdr args) ,@body)
                e)))))))
```

Figure 4. Currying expanders.

```
(define delay
  (lambda (x e)
    `(box (lambda () ,(e x e)))))

(set! *application-expander*
  (lambda (x e)
    `(,(e (car x) e)
      ,@(map (lambda (x)
               (if (symbol? x) x (delay x e)))
             (cdr x)))))

(set! *identifier-expander*
  (lambda (x e)
    `((unbox ,x))))

(install-expander 'set!
  (lambda (x e)
    `(set-box! ,(cadr x)
       ((lambda (v) (lambda () v))
        ,(e (caddr x) e)))))
```

Figure 5. Call-by-name expanders.

```
(define delay
  (let ((g (gensym)))
    (lambda (x e)
      `((lambda (,g)
          (set-box! ,g
            (lambda ()
              ((lambda (v)
                 (set-box! ,g (lambda () v))
                 v)
               ,(e x e))))
          ,g)
        (box '*)))))
```

Figure 6. `delay` for call-by-need.

```
(define extend-expander
  (lambda (current-expander keyword keyword-expander)
    (lambda (x e)
      (if (and (pair? x) (eq? (car x) keyword))
          (keyword-expander x e)
          (current-expander x e)))))

(install-expander 'macrolet
  (lambda (x e)
    (recur loop ((macs (cadr x)) (e e))
      (if (null? macs)
          (e (caddr x) e)
          (let ((key (caar macs))
                (pat (cadar macs))
                (body (caddar macs)))
            (loop (cdr macs)
                  (extend-expander e key
                    (make-macro pat body))))))))
```

Figure 7. `extend-expander` and `macrolet`.

We may similarly obtain call-by-need semantics, in which argument evaluation is delayed as long as possible and performed only once; see Figure 6. Here, if the thunk made from the argument expression is ever invoked, it evaluates the expression, changes its box to hold a new thunk, and returns the value. The new thunk merely returns the value directly.

Next we define `extend-expander`, a function that provides a convenient means of extending an expander so that it recognizes a new syntactic extension with which a keyword and expander are associated; see Figure 7. This extension technique is analogous to that used in denotational semantics to extend environments. A practical use of `extend-expander` (shown in Figure 7) is to define an expander for `macrolet`, which temporarily establishes keyword bindings that are visible only within its body, for example:

```
(macrolet ((foo (x y) `(list ,x ,y))
           (bar ((a) . b) `(list ,a ,b)))
  (append (foo 1 2) (bar (3) . 4)))
```

returns `(1 2 3 4)`.

4. Debugging with syntactic extensions

As a final example of our expansion mechanism's power, we illustrate an approach to the construction of a variety of debugging tools, including tracers, steppers, and inspectors. These tools are obtained using only function definition and the syntactic extension mechanism introduced here. This has a number of advantages over other approaches to implementing debugging tools:

- it is portable to implementations of the same language that support this style of syntactic extension, since it is not dependent on the run time architecture;
- it is independent of the method of implementation (compilation or interpretation);
- its correctness is easily verified, since it is simple and is defined in terms of the existing evaluation mechanism;
- its simplicity encourages experimentation and customization to meet specific needs; and
- the regions of text in which it is effective are easily controlled and there is no efficiency penalty for code outside of these regions.

We begin with a simple trace facility that prints each application before its evaluation and its result after evaluation, with indentation provided to keep track of the applications in the process of evaluation. This is accomplished by redefining the application expander so that applications of the form ($x_1\ \ldots\ x_n$) are expanded into expressions of the form

`(trace-form '(`$x_1\ \ldots\ x_n$`)`
`  (lambda () (`$x'_1\ \ldots\ x'_n$`)))`

where each x'_i is obtained by similarly expanding x_i. See Figure 8. This provides, with remarkable economy, a frequently useful trace facility.

The following trace illustrates two problems with this approach.

```
> (let ((x '(a b))) (car (cdr x)))

((lambda (x) (car (cdr x))) (quote (a b)))
|  (car (cdr x))
|  |  (cdr x)
|  |  (b)
|  b
b

b
```

We would usually like forms other than applications, such as `let` and `quote`, to be traced. Also, we usually do not want to see forms that are not in our souce code, but were instead introduced by syntactic extensions, such as the `lambda` application introduced by the `let` expression above. Finally, we may wish to trace only a small part of a large program, in order to reduce the volume of trace output and improve efficiency.

```
(set! *application-expander*
  (lambda (x e)
    `(trace-form ',x
       (lambda () ,(map (lambda (x) (e x e)) x)))))

(define trace-form
  (let ((level 0))
    (lambda (source thunk)
      (do-times (n level) (display "|  "))
      (printf "~s~%" source)
      (let ((result
              (fluid-let ((level (add1 level)))
                (thunk))))
       (do-times (n level) (display "|  "))
       (printf "~s~%" result)
       result))))
```

Figure 8. Application tracer.

These considerations motivate the `trace-source` syntactic extension that traces all, and only, forms occurring in the source code of its body. For example,

```
> (cons 'c
    (trace-source
      (let ((x '(a b))) (car (cdr x)))))

(let ((x (quote (a b)))) (car (cdr x)))
|  (quote (a b))
|  (a b)
|  (car (cdr x))
|  |  (cdr x)
|  |  (b)
|  b
b

(c . b)
```

See Figure 9 for the implementation of `trace-source`.

```
(install-expander 'trace-source
  (lambda (x e)
    (let ((e1 (trace-expander (cadr x) e)))
      (e1 (cadr x) e1))))

(define trace-expander
  (lambda (source e)
    (lambda (x e1)
      (if (and (pair? x) (subexpression? x source))
          `(trace-form ',x (lambda () ,(e x e1)))
          (e x e1)))))

(define subexpression?
  (lambda (x s)
    (or (eq? x s)
        (and (pair? s)
             (or (subexpression? x (car s))
                 (subexpression? x (cdr s)))))))
```

Figure 9. Source code trace facility.

Note especially the two occurrences of `(e x e1)` in `trace-expander`. If `(e1 x e1)` were used instead, infinite expansion would result, for `trace-expander` would be feeding on its own output. (It is extraordinarily easy to create infinite expansion-time loops using expanders!) Using `e` for the next level of expansion avoids this recursion, but it would not do to use `(e x e)`: in order for subforms to be traced, it is necessary for the expanders at the next level to expand their subforms with the trace expander, `e1`. Also note that the trace expander carries the original source code with it. The full power of EPS is realized only when expanders are closures with local state.

By modifying `trace-form`, we can turn our tracer into a stepper; see Figure 10. `step*` causes the current expression to be evaluated without further stepping. This is achieved by fluidly (or dynamically) rebinding `trace-form` to a function that invokes the thunk it is passed. (The original binding of `trace-form` is automatically restored by `fluid-let` when the value of the current expression, `ans`, is returned.)

Next we endow our trace facility with the powers of an *inspector:* the ability to examine and change the values of lexical variable bindings. In order that `trace-form` have access to the run time environment, we replace every expression of the form `(lambda` ($id_1 \ldots id_n$) $body_1 \ldots body_m$`)` within the scope of `trace-source` with one of the form

```
(lambda ($id_1$ ... $id_n$)
  (trace-lambda-body
    '($id_1$ ... $id_n$)
    (list $locative_1$ ... $locative_n$)
    (lambda () $body'_1$ ... $body'_m$)))
```

where $body'_i$ is obtained by expanding $body_i$, and the locatives give `trace-lambda-body` access to the bindings of the formal parameters. See Figure 11. The locative for identifier *id* is represented as a functional object that responds to the `get` and `put` messages by returning the current value of *id* and a function that assigns a given value to *id*, respectively. `trace-lambda-body` fluidly assigns to the trace environment a list of identifier-locative pairs corresponding to the lambda expression's arguments. The `trace-form` commands `see` and `set!` may then be used to inspect and modify the environment bindings of the local contour.

By redefining `trace-lambda-expander` it is possible to inspect all visible environment bindings; see Figure 12. `*all-vars*` is fluidly bound, *at expansion time,* so that it lists all visible identifiers in the current lexical environment. This list is then used in place of the lambda formals list when constructing `*trace-env*`.

Common Lisp provides a special form, `compiler-let`, to give the programmer limited control over the state in which macro expansion occurs. As this last version of `trace-lambda-expander` demonstrates, the use of fluid variables provides this control without the need for additional machinery.

This technique of implementing a debugger is comparable in run time efficiency to other debuggers, and provides all the advantages mentioned at the beginning of this section. But it does have one drawback: it generates voluminous code. In some cases this would be a serious problem. In other situations, where the region in which debugging is enabled is of moderate size, the advantages of this approach to debugging more than compensate for the increase in code size.

```
(define trace-form
  (lambda (source thunk)
    (recur loop ()
      (printf "~s: " source)
      (case (read)
        (step
         (let ((ans (thunk)))
           (printf "~s returns ~s~%" source ans)
           ans))
        (step*
         (fluid-let ((trace-form (lambda (s f) (f))))
           (let ((ans (thunk)))
             (printf "~s returns ~s~%" source ans)
             ans)))
        (else
         (printf "options: step, step*")
         (loop))))))
```

Figure 10. `trace-form` for a stepper.

5. Conclusion

We have proposed a syntactic extension technique, called expansion-passing style (EPS), that is significantly more flexible than the traditional mechanism. It allows selective expansion of subexpressions, expansion of subexpressions using modified expansion functions or modified state, and expansion of application and identifier forms. It also simplifies the expansion algorithm and improves modularity by factoring out special forms. EPS may be easily incorporated into most Lisp systems without the need for system level programming.

Though EPS is a substantial improvement over the traditional approach to syntactic extension, it does present a few difficulties. We have already shown instances in which programming with expanders is error prone: it is easy to perform unwanted, or even infinite, expansions. Our defense is that (1) powerful tools frequently require great care in their use, and (2) if the power of this mechanism is not required, there need be no danger in its use. (We have shown that traditional macro expansion facilities, such as `defmacro`, can be provided in the more general EPS context.) Another problem is that we have found it necessary to bind `*application-expander*` and `*identifier-expander*` in a different manner than other expanders. Thus `install-expander` and `macrolet` cannot be used to redefine the application and identifier expanders. The root of this problem is that an expander cannot tell until all code has been expanded whether a given form is an application or not. This in turn results directly from the overloading of symbols to denote both identifiers and keywords in traditional Lisp syntax.

```
(install-expander 'trace-source
  (lambda (x e)
    (let ((e1 (extend-expander e 'lambda
                trace-lambda-expander)))
      (let ((e2 (trace-expander (cadr x) e1)))
        (e2 (cadr x) e2)))))

(define trace-lambda-expander
  (lambda (x e)
    `(lambda ,(cadr x)
       (trace-lambda-body
         ',(cadr x)
         (list ,@(map locative (cadr x)))
         (lambda ()
           ,@(map (lambda (x) (e x e)) (cddr x)))))))

(define locative
  (let ((msg (gensym)) (x (gensym)))
    (lambda (id)
      `(lambda (,msg)
         (if (eq? ,msg 'get)
             ,id
             (lambda (,x) (set! ,id ,x)))))))

(define *trace-env* '())

(define trace-lambda-body
  (lambda (vars vals body)
    (fluid-let ((*trace-env* (map cons vars vals)))
      (body))))
```

```
(define trace-form
  (lambda (source thunk)
    (recur loop ()
      (printf "~s: " source)
      (case (read)
        (step
         (let ((ans (thunk)))
           (printf "~s returns ~s~%" source ans)
           ans))
        (step*
         (fluid-let ((trace-form
                       (lambda (s f) (f))))
           (let ((ans (thunk)))
             (printf "~s returns ~s~%" source ans)
             ans)))
        (see
         (for-each
           (lambda (x)
             (printf "~s = ~s~%"
               (car x)
               ((cdr x) 'get)))
           *trace-env*)
         (loop))
        (set!
         (let* ((id (read))
                (val (read))
                (pair (assq id *trace-env*)))
           (if pair
               (begin
                 (((cdr pair) 'put) val)
                 (printf "~s = ~s~%" id val))
               (printf "id ~s not found~%" id)))
         (loop))
        (else
         (printf "options: step, step*, see, set!")
         (loop))))))
```

Figure 11. A stepping inspector.

The critical difference between the facility proposed here and the traditional macro mechanism is that expansion functions are passed not only an expression to be expanded, but also another expansion function. This function may or may not be used to perform further expansion. This is analogous to passing an explicit continuation to a procedure, which is well known to be a powerful programming technique (for example, see [1,7]). In this paper we have demonstrated that the closely related technique of passing expanders explicitly provides a powerful tool for defining syntactic extensions.

Acknowledgment: This material is based in part on work supported by the National Science Foundation under grant numbers MCS 83-04567 and MCS 83-03325.

```
(define *all-vars* '())

(define trace-lambda-expander
  (lambda (x e)
    (fluid-let ((*all-vars*
                  (union (cadr x) *all-vars*)))
      `(lambda ,(cadr x)
         (trace-lambda-body
           ',*all-vars*
           (list ,@(map locative *all-vars*))
           (lambda ()
             ,@(map (lambda (x) (e x e))
                    (cddr x))))))))
```

Figure 12. A visible environment inspector.

References

[1] Charniak, E., Riesbeck, C.K., and McDermott, D.V., *Artificial Intelligence Programming*, Lawrence Erlbaum Associates, 1980.

[2] Clinger, W.C., Ed., The Revised Revised Report on Scheme, Computer Science Department Technical Report No. 174, Indiana University, Bloomington, Indiana, 1985, and Artificial Intelligence Memo No. 848, MIT, Cambridge, Massachusetts, 1985.

[3] Dybvig, R.K., and Smith, B.T.., *The Scheme Programming Language*, Prentice-Hall, 1986, in press.

[4] Fischer, M.J., Lambda calculus schemata, *Proceedings ACM Conference on Proving Assertions about Programs*, Las Cruces, New Mexico, pp. 104–109, 1972.

[5] Plotkin, G., Call-by-name, call-by-value, and the λ-calculus, *Theoretical Computer Science*, **1**, pp. 125–159, 1975.

[6] Steele, G.L., *Common LISP: The Language*, Digital Press, 1984.

[7] Steele, G.L., LAMBDA: the ultimate declarative, Artificial Intelligence Memo No. 379, MIT, Cambridge, Massachusetts, 1976.