Expansion-Passing Style: A General Macro Mechanism*

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

Abstract

The traditional Lisp 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 variable expressions. 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 alternative facility that avoids these problems, using a technique called *expansion-passing style* (EPS). 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. The power of this technique is illustrated with several examples. Most Lisp systems may be adapted to employ this technique.

1. Introduction

Lisp systems generally include a facility that allows convenient extension of the source language syntax. This facility is implemented by expanding *syntactic extensions* (also called *macro calls*) into the core language (special forms) of the Lisp system. There are several advantages to source-level expansion over the use of a specialized interpreter to provide new syntactic forms:

- Source-level expansion eliminates the need for extra layers of interpretation. Expansion need only be performed once, prior to evaluation, resulting in greater efficiency.
- Many language extensions are more easily provided by source-level expansion than by interpretation.
- The semantics of a language obtained by "sugaring" the syntax of an existing well-understood language with syntactic transformations is often more easily understood and verified than the semantics of a language obtained by writing a new interpreter.
- The extended language is easily ported to another host that supports the same core language.

In the next section we briefly review the dialect of Lisp used to express our examples. Section 3 then reviews the traditional Lisp syntactic extension mechanism. We then present examples of several useful forms of syntactic extension that are impossible using the conventional mechanism. This motivates the introduction in Section 4 of a facility with the flexibility to implement these extensions. Most Lisp systems may be adapted to employ this facility. A few simple examples of this facility follow in Section 5. As a more substantial and practical example of its power, in Section 6 we present a set of debugging tools developed with this facility. Finally, a few difficulties with the facility are noted and alternative approaches are discussed. Some familiarity with Lisp and its traditional macro expansion mechanism is assumed. For a survey of macro expansion techniques, see [6].

2. Coding Conventions

The code that follows is expressed in Scheme, a lexically scoped dialect of Lisp [2, 10]. The only *core* syntactic forms—those that are not implemented as syntactic extensions—are given in Figure 1. Superscript "*" and

---

* A preliminary version of this paper was presented at the 1986 ACM Symposium on Lisp and Functional Programming.

This material is based on work supported by the National Science Foundation under grant numbers MCS 83-04567, MCS 83-03325 and DCR 85-01277.

Authors' Address: Computer Science Department, Indiana University, Lindley Hall 101, Bloomington, IN 47405; dybvig,friedman,haynes@iuvax.cs.indiana.edu

Appeared in *Lisp and Symbolic Computation 1*, 1, 53–75 (1988).

⟨expression⟩ ::=
  ⟨variable⟩
 | ⟨self evaluating literal⟩
 | (**quote** ⟨datum⟩)
 | (**lambda** (⟨variable⟩*) ⟨expression⟩$^+$)
 | (**if** ⟨expression⟩ ⟨expression⟩ ⟨expression⟩)
 | (**set!** ⟨variable⟩ ⟨expression⟩)
 | ⟨application⟩

⟨application⟩ ::= (⟨expression⟩$^+$)

Figure 1. Syntax of core forms.

"+" indicate zero or more and one or more of the preceding form, respectively. This core language suffices for Scheme, except that it does not support improper **lambda** lists.

The expression '⟨datum⟩ is equivalent to (**quote** ⟨datum⟩). The expression ‘⟨datum⟩ is similar to '⟨datum⟩, except that each subform preceded by comma (,) is replaced by its value in the resulting structure. An expression preceded by a comma-at (,@) is treated similarly except that the value must be a list, which is spliced into the constructed structure. (There is some variation in the precise semantics of backquote used by Lisp systems, but this should not affect our examples.) For example:[1]

‘(a ,@(*list* (**quote** b) 'c) ,(+ 1 2))

is equivalent to:

(*cons* (**quote** *a*) (*append* (*list* (**quote** *b*) (**quote** *c*)) (*list* (+ 1 2))))

and evaluates to (a b c 3).

In Scheme a name may appear following the **let** keyword, as in:

(**let** *loop* ((*x* *exp1*)) ... (*loop* *exp2*) ...).

Here x is initially bound (within the body of the **let** expression) to the value of *exp1* and *loop* is bound to a function that, when invoked, will rebind x to the value of its argument (*exp2* in the example) and recursively reenter the body of the **let** expression. Other respects in which Scheme differs from traditional Lisp dialects should be clear from context. (For example, Scheme's **set!** and *map* are Lisp's **setq** and *mapcar*.) For our purposes, the only significant semantic difference between Scheme and most other Lisp dialects is that **lambda** expressions close over the current lexical environment, yielding functions that may be freely used as arguments and returned as results.

We use one syntactic form and a few functions that are not part of standard Scheme, though similar features are found in most Scheme (and Lisp) systems. The syntactic form:

(**fluid-let** ((*variable* *expression*)) *body-expression*)

assigns *variable* the value of *expression* during the evaluation of *body-expression* and restores the original value of *variable* before the value of *body-expression* is returned as the value of the **fluid-let** form. (This is referred to as dynamic or fluid binding and is similar to *special* binding in Common Lisp.) The function *eval* is the standard evaluation function. The *gensym* function returns a symbol that is guaranteed to be distinct from all other symbols. Finally, *printf* provides formatted output à la Common Lisp's *format*. These features are described fully in [2].

3. The Traditional Approach to Syntactic Extension

Syntactic transformation of Lisp programs may be performed by manipulating source expressions prior to evaluation. Provision for this is easily made by adding a preprocessor to *eval*, which we call *expand*. Decoupling of the syntactic extension mechanism from the evaluation mechanism simplifies the underlying

[1] Keywords are displayed in bold face, variables in italics, and literals in roman font.

```
(define old-style-expand
  (lambda (x)
    (cond
      ((variable? x) x)
      ((literal? x) x)
      ((macro? (car x))
       (old-style-expand ((macro-function (car x)) x)))
      ((eq? (car x) 'quote) x)
      ((eq? (car x) 'lambda)
       `(lambda ,(cadr x)
          ,@(map old-style-expand (cddr x))))
      ((eq? (car x) 'if)
       `(if ,@(map old-style-expand (cdr x))))
      ((eq? (car x) 'set!)
       `(set! ,(cadr x) ,(old-style-expand (caddr x))))
      (else (map old-style-expand x)))))

(define variable? symbol?)

(define literal?
  (lambda (x)
    (not (or (symbol? x) (pair? x)))))
```

Figure 2. Traditional syntactic expansion mechanism.

compiler or interpreter and makes syntactic transformation independent from the implementation.[2]

Syntactic extensions are indicated by keywords appearing in the first position of a structured form (expression). Each syntactic extension keyword is bound to an expansion function in some manner. (We assume that the functions *install-macro* and *macro-function* make and retrieve these bindings and that the predicate *macro?* returns true when passed a symbol with such a binding.) Traditionally, when *expand* encounters a form whose first position contains a keyword with an associated expansion function, the function is passed the entire form. The expansion function then returns a new form, obtained by transforming the old one, that is then expanded in place of the old form. Because forms returned by expansion functions are themselves expanded, expansion functions can return forms involving other syntactic extensions. See Figure 2 for an expand function using this protocol that might be used with our core language. Each special form (there are 24 in Common Lisp [11]) must be treated as a special case.

For example, **let** expressions of the form:

(**let** $((id_1\ exp_1)\ \ldots\ (id_n\ exp_n))$
  $body_1\ \ldots\ body_m)$

expand into equivalent **lambda** applications of the form:

((**lambda** $(id_1\ \ldots\ id_n)\ body_1\ \ldots\ body_m)$
  $exp_1\ \ldots\ exp_n)$.

[2] This decoupling also means that syntactic transformations are independent of the evaluation environment. In Lisp systems that have coupled evaluation and syntactic transformation mechanisms, syntactic transformations that are dependent on the evaluation environment are sometimes used. This is inconsistent with our view that syntactic transformations should implement (static) syntactic extensions.

This syntactic extension for **let** in terms of **lambda** may be defined as follows:

```
(install-macro 'let
  (lambda (x)
    `((lambda ,(map car (cadr x)) ,@(cddr x))
      ,@(map cadr (cadr x)))))).
```

Although this traditional expansion mechanism provides considerable power at low cost, it lacks flexibility that is sometimes important. Since reexpansion of transformed expressions is automatic, transformation functions do not have control over whether all, or part, of the form they return is expanded further, or how this expansion is done.

As examples of transformations that are not possible with the traditional expansion mechanism, consider the problems of obtaining curried, call-by-name, or call-by-need 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:

$(x_0\ x_1\ x_2)$

would become:

$((x'_0\ x'_1)\ x'_2)$

where x'_0, x'_1, and x'_2 are curried forms of x_0, x_1, and x_2 obtained by recursively applying these transformations, and:

(**lambda** $(v_1\ v_2)\ x$)

would become:

(**lambda** (v_1) (**lambda** $(v_2)\ x'$))

where x' is the curried form of x.

To obtain call-by-name semantics (without **set!**), it suffices to

- replace every application argument x that is not a variable by (**lambda** () x'), where x' is the result of recursively applying the call-by-name transformation to x, and
- replace every variable reference v that is not an application argument by (v).

For example, the expression:

(**lambda** (x) (*name-car* (*name-cdr* x)))

would be transformed into:

```
(lambda (x)
  ((name-car)
   (lambda ()
     ((name-cdr) (lambda () x)))))
```

where *name-car* and *name-cdr* are versions of *car* and *cdr* that, when forced (invoked as functions of no arguments), force their arguments. Call-by-need semantics may be achieved with a more complex application expansion [4].

To perform these transformations, we require a nontraditional expansion algorithm that expands variables and applications. It is also sometimes essential that further expansion be performed only on selected subexpressions of transformed expressions. For example, in the call-by-name expansion, it is not appropriate to perform the expansion on the application introduced by expanding a variable reference.

Another case in which subexpressions require different expansion treatment is the definition of lexical syntactic extensions, in the manner of **macrolet** [11]. Lexical syntactic extensions require that the body of the form that introduces them be expanded in such a way that an additional keyword is recognized and associated with a new syntactic extension, which may override an existing syntactic extension.

In some cases a solution to these problems would be for the transformation functions to fully expand the expressions they are passed so that the expressions they return only contain forms in the core language.

Further expansion of such expressions will then have no effect, since expressions in the core language are fixed points of the *expand* function. There are several problems with this approach:

- It requires that such expansion functions have knowledge of the core language (which may be implementation specific).
- It complicates expansion functions since they must treat each core form as a special case.
- It makes it difficult for expansion functions to treat subexpressions differently at different times. (We will present several examples where this is necessary.)
- It does not permit syntactic extension of core forms in which the expansion contains the same form, as with **lambda** in the currying example. In such cases the expansion process would not terminate.

4. Expansion-Passing Style

We conclude from observations in the last section that expansion functions should have control over the further expansion of the forms they return. This is analogous to a function having 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 control [5, 9, 12].

In the *macro?* line of *old-style-expand* (see Figure 2), the recursive call occurs in a 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, in which case the *macro?* line would be written:

```
((macro? (car x))
 ((macro-function (car x)) x old-style-expand)).
```

The value returned by the expansion function would then require no further expansion.

This motivates the following modification of the traditional macro protocol:

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

We call these expansion functions *expanders* and refer to this protocol as *expansion-passing style* (EPS).[3]

EPS gives expanders control over whether the entire transformed expression is to be expanded further, which subexpressions are to be expanded, when the expansions are to be performed, and 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, as we shall see.

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 variables x and e for form and expander arguments, respectively.) Thus the EPS mechanism strictly extends the power of the traditional macro mechanism, and the simpler traditional facility may still be used when it suffices.[4]

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 3. The functions *install-expander*, *expander-function*, and *expander?* are

---

[3] The analogy between EPS and CPS is useful, but it is not exact. A CPS function has control over how the entire computation continues, whereas an expander only has control over how the forms it is passed are expanded. This is reflected by the non-tail-recursive position of the other calls of *old-style-expand* in Figure 2. Also, continuations are not passed another continuation as a second parameter.

[4] It should also be possible for EPS to coexist with the *macro-by-example* mechanism of Kohlbecker and Wand [6, 8]. This mechanism makes it much easier to express many syntactic extensions, but shares the limitations of traditional macro expansion noted in the last section.

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

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

(define initial-expander
  (lambda (x e)
    (cond
      ((or (variable? x) (literal? x)) x)
      ((expander? (car x)) ((expander-function (car x)) x e))
      (else (map (lambda (x) (e x e)) x)))))

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

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

(install-expander 'if
  (lambda (x e)
    `(if ,@(map (lambda (x) (e x e)) (cdr x)))))

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

Figure 3. Basic EPS functions.

the expander counterparts of *install-macro*, *macro-function* and *macro?*. Neither *expand* nor *initial-expander* is directly recursive.

It might seem that *initial-expander* could be defined using (*e1 x initial-expander*) instead of (*e1 x e*), but this would not be correct. In other uses of *initial-expander*, such as in *expand-once* and in the trace facilities to be presented later, *e* is not always bound to *initial-expander*.

It is no longer necessary to include each of the special forms, such as **lambda** or **quote**, in the system expand function. It is only necessary to associate expanders with special form keywords in the same way that new syntactic extensions are defined. With the traditional macro mechanism this is not possible. In some cases it is essential that certain subparts not be expanded (for example, the formal parameter list of **lambda** or the literal part of **quote** forms) or that the entire form not be reexpanded (for example, **quote** forms expand to themselves, and hence the expansion would not terminate if repeated). However, EPS gives expanders control over further expansion, as the expanders defined in Figure 3 demonstrate. Factoring the special forms out of the expander increases modularity, encourages custom variations on the expander, and allows redefinition of special form expanders. Another definition of the **lambda** expander, to be presented shortly, provides an example of special form redefinition.

```
(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 (pattern body)
    (let ((x (gensym)) (e (gensym)))
      `(lambda (,x ,e)
         (,e (let ,(destructure pattern `(cdr ,x) '())
               ,body)
             ,e)))))

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

Figure 4. **defmacro** expander.

5. Simple Examples of EPS

It is straightforward to provide a conventional macro definition interface using expanders. The expander of Figure 4 supports the essential features of Common Lisp's **defmacro** [11]. (Most of the code is dedicated to destructuring the arguments to the macro.) For example:

```
(defmacro let (decls . body)
  `((lambda ,(map car decls) ,@body)
    ,@(map cadr decls)))
```

expands into:

```
(install-expander 'let
  (lambda (g1 g2)
    (g2 (let ((decls (car (cdr g1))) (body (cdr (cdr g1))))
          `((lambda ,(map car decls) ,@body) ,@(map cadr decls)))
        g2))).
```

The function *make-macro* uses *gensym* to create new symbols for the formal parameters of newly created expanders, such as *g1* and *g2* in the example. This avoids the possibility that identical variables in the body of the new expander be captured by the parameters of the expander. (A more sophisticated solution to the capturing problem is provided by *hygienic* macro expansion [6, 7]. It is possible to combine hygienic expansion and EPS, but we do not address this issue here.)

```
(install-expander 'curry
  (lambda (x e)
    (let ((e1 (lambda (x e2)
                (cond
                  ((literal? x) (e x e2))
                  ((and (lambda? x) (> (length (cadr x)) 1))
                   (e `(lambda (,(caadr x))
                         (lambda ,(cdadr x) ,@(cddr x)))
                       e2))
                  ((and (application? x) (> (length x) 2))
                   (e2 `((,(car x) ,(cadr x)) ,@(cddr x)) e2))
                  (else (e x e2))))))
      (e1 (cadr x) e1))))

(define lambda?
  (lambda (x)
    (and (pair? x)
         (eq? (car x) 'lambda))))

(define application?
  (lambda (x)
    (and (pair? x)
         (not (expander? (car x))))))
```

Figure 5. Currying expanders.

Currying of applications and lambda abstractions as discussed in Section 3 is now possible. See Figure 5. When an expression of the form (**curry** x) is expanded, expander *e1* of the curry expander is passed x and *e1*. If x is a lambda expression of more than one argument, it is transformed into a lambda expression binding the first argument, within which is a lambda expression binding the rest of the arguments. The transformed expression, x', is then passed to the expander e that was originally passed to the curry expander. It is possible for another expander for lambda to be in effect at the time the curry expander is invoked, in which case it would then expand the outer lambda of x'. In any case, expansion of subexpressions of x' are performed by *e1* (then bound to *e2*), thus assuring the full currying of the subexpressions. If x is an application of more than one argument, say (x_0 x_1 x_2 ...), then it is transformed into an application of the form ((x_0 x_1) x_2 ...). The entire transformed application, as well as its subexpressions, are further expanded by *e1*. Thus if x were an application of three or more arguments, *e1* would again transform it. If x is not a lambda expression or application that requires transformation, then the **else** clause passes x to e for further expansion along with *e1*, which curries subexpressions.

See Figure 6 for a solution to the call-by-name implementation technique outlined in Section 3. When an expression of the form (**call-by-name** x) is expanded, *e1* is passed x and *e1*. If x is a variable, then (x) is returned. If x is a application, then the function and argument subexpressions are expanded using *e1* and passed to *call-by-name-application*, which wraps (**lambda** () ...) around each argument that is not a variable. If x is not a variable or application, then it is passed to expander e along with expander *e1*, which is used to expand subexpressions of x.

The factorial function may be defined, in a way that dramatizes the normal order evaluation semantics, using the Y combinator [13]:

```
(install-expander 'call-by-name
  (lambda (x e)
    (let ((e1 (lambda (x e2)
                (cond
                  ((variable? x) `(,x))
                  ((application? x)
                   (call-by-name-application x e2))
                  (else (e x e2))))))
      (e1 (cadr x) e1))))

(define call-by-name-application
  (lambda (x e)
    `(,(e (car x) e)
      ,@(map (lambda (x)
               (if (variable? x)
                   x
                   `(lambda () ,(e x e))))
             (cdr x)))))
```

Figure 6. Call-by-name expanders.

```
(define by-name-factorial
  (call-by-name
    (let ((Y (lambda (f)
               ((lambda (x) (f (x x)))
                (lambda (x) (f (x x))))))
          (test (lambda (tst thn els)
                  (if tst thn els))))
      (Y (lambda (f)
           (lambda (x)
             (test (name= x 0)
                   1
                   (name* x (f (name- x 1))))))))))
```

where *name=*, *name∗* and *name−* are versions of the =, ∗ and − primitives that, when forced, yield functions that force their arguments.

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).

6. Debugging with syntactic extensions

To further illustrate the power of the proposed expansion mechanism, we demonstrate an approach to the construction of a variety of debugging tools, including tracers, steppers, and inspectors. Each of these

```
(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)
    (let loop ((macs (cadr x)) (e e))
      (if (null? macs)
          (e (caddr x) e)
          (let ((key (caar macs))
                (pattern (cadar macs))
                (body (caddar macs)))
            (loop (cdr macs)
                  (extend-expander e key
                    (eval (make-macro pattern body)))))))))
```

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

examples illustrates a different facet of EPS.

The debugging tools developed here are obtained using only function definition and syntactic extension. This has a number of advantages over other approaches to implementing debugging tools:

- It is portable to implementations of the same language that support syntactic extension with EPS, since it is not dependent on the run time architecture.
- It is independent of the method of implementation, e.g., compilation or interpretation.
- Its correctness is easier to assure, since it is simple and is defined in terms of the existing evaluation mechanism.
- It need not "understand" the full language, yet it can distinguish source expressions from expanded expressions.
- Its simplicity encourages experimentation and customization to specific needs.
- The regions of text in which it is effective are easily controlled, and there is no efficiency penalty for code outside of these regions. (Traditional Lisp systems must be run in a slower "debugging" mode when such tools are used.)

We begin with a simple trace facility that prints each application before its evaluation, and its result after evaluation. (Indentation is provided to keep track of the applications in the process of evaluation.) This is accomplished by providing a syntactic extension of the form (**trace-applications** *exp*) that expands each application of the form $(x_1 \ \ldots \ x_n)$ within *exp* into an expression of the form:

$$
\begin{array}{l}
(\mathit{trace\text{-}form}\ \ '(x_1\ \ldots\ x_n) \\
\quad (\mathbf{lambda}\ ()\ (x'_1\ \ldots\ x'_n)))
\end{array}
$$

where each x'_i is obtained by similarly expanding x_i. See Figure 8. This provides, with remarkable economy, a useful trace facility.[5]

For example, consider the expression:

[5] With this approach the scope of tracing is determined statically, since code that is not within a **trace-application** form is not expanded in this way. This contrasts with the usual dynamic approach to tracing control, in which tracing is enabled,

```
(install-expander 'trace-applications
  (lambda (x e)
    (let ((e1 (lambda (x e2)
                (if (application? x)
                    `(trace-form ',x
                       (lambda ()
                         ,(map (lambda (x) (e2 x e2)) x)))
                    (e x e2)))))
      (e1 (cadr x) e1)))

(define print-trace
  (lambda (level object)
    (do ((n level (- n 1)))
        ((zero? n) (printf "~s~%" object))
        (printf "|  "))))

(define trace-form
  (let ((level 0))
    (lambda (source thunk)
      (print-trace level source)
      (let ((value
              (fluid-let ((level (+ level 1)))
                (thunk))))
        (print-trace level value)
        value))))
```

Figure 8. Application tracer.

```
(trace-applications (let ((x '(a b))) (car (cdr x)))).
```

After expansion of the **let** form into the equivalent **lambda** application it becomes:

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

and after expansion is completed by transforming the applications we have:

```
(trace-form '((lambda (x) (car (cdr x))) (quote (a b)))
  (lambda ()
    ((lambda (x)
       (trace-form '(car (cdr x))
         (lambda () (car (trace-form '(cdr x)
                                     (lambda () (cdr x)))))))
     '(a b)))).
```

Execution of this expression yields the trace:

---

or disabled, during the evaluation of given expressions (even when control has temporarily passed to code defined elsewhere). Static or dynamic control of tracing may be preferable, depending on circumstances. Dynamic control could easily be added to the mechanism presented here by fluidly binding a flag which enables printing.

```
(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 (form source)
    (or (eq? form source)
        (and (pair? source)
             (or (subexpression? form (car source))
                 (subexpression? form (cdr source)))))))
```

Figure 9. Source code trace facility.

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

This trace illustrates two problems with the current approach. First, we would usually like forms other than applications, such as **let** and **quote**, to be traced. Second, we usually do not want to see forms that are not in our source code, but were instead introduced by syntactic extensions, such as the **lambda** application introduced by expansion of the **let** form of our example.

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

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

yields the trace:

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

and returns the value (c . b).

```
(define trace-form
  (lambda (source thunk)
    (let 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.

See Figure 9 for the implementation of **trace-source**. Note 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. 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 power of EPS is most easily used when expanders are closures with local state, though in this case the same effect could be obtained with fluid binding (as we illustrate later).

By modifying *trace-form*, we can turn our tracer into a stepper. See Figure 10. Each time *trace-form* is invoked, it prompts for the symbol 'step' or 'step∗'. The symbol 'step' causes the stepper to prompt when entering each (top-level) subexpression of the current expression and to print the value of each of these expressions when its evaluation is complete. The symbol 'step∗' causes the current expression to be evaluated, and its value printed, without further stepping. This is achieved by fluidly binding *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 ... id_n) $body_1$ ... $body_m$)

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 each $locative_i$ gives *trace-lambda-body* access to the binding of the corresponding formal parameter. The locative for variable v is represented as a pair of functional objects, one that returns the current value of v and one that assigns a new value to v. See Figure 11. *trace-lambda-body* fluidly assigns to the trace environment a list of variable-locative pairs corresponding to

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

(define locative
  (let ((x (gensym)))
    (lambda (id)
      `(cons (lambda () ,id)
             (lambda (,x) (set! ,id ,x))))))

(define *trace-env* '())

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

(continued)

Figure 11. A stepping inspector.

```
(define trace-form
  (lambda (source thunk)
    (let 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) ((cadr x))))
           *trace-env*)
         (loop))
        ((set!)
         (let* ((id (read))
                (val (read))
                (pair (assq id *trace-env*)))
           (if pair
               (begin
                 ((cddr pair) val)
                 (printf "~s = ~s~%" id val))
               (printf "~s not found~%" id)))
         (loop))
        (else
         (printf "options: step, step*, see, set!")
         (loop))))))
```

Figure 11. A stepping inspector (continued).

```
(define trace-lambda-expander
  (let ((all-vars '()))
    (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.

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. The variable *all-vars* is fluidly bound, *at expansion time,* to a list of all visible variables 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 environment in which macro expansion occurs. As this last version of *trace-lambda-expander* demonstrates, the use of fluid (or special) variables provides this control without the need for additional machinery.

Another version of *trace-form* not presented here has been used to record the source code and value of each form or application in a circular buffer, rather than printing this information. When an error (or other form of break) occurs, it is then possible to inspect the contents of the buffer to follow the recent history of the computation. Such a history mechanism has proven useful for debugging student assignments [3].

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 results in the generation of voluminous code. In some cases this would be a serious problem. In other situations, where the regions in which debugging is enabled are of moderate size, the advantages of this approach to debugging more than compensate for the increase in code size.

7. Discussion and conclusion

We have proposed a syntactic extension technique, called expansion-passing style (EPS), that is significantly more flexible than the traditional Lisp macro mechanism. It allows selective expansion of subexpressions, expansion of subexpressions using modified expansion functions or modified state, and expansion of application and variable forms. It also simplifies the expansion algorithm and improves modularity by factoring out special forms. EPS may be incorporated into most Lisp systems. (It generally suffices to redefine *eval* to first invoke *expand* on its argument.)

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 function, which is well known to be a powerful programming technique (for example, see [1, 12]). In this paper we have demonstrated that the related technique of passing expanders explicitly provides a powerful tool for defining syntactic extensions.

Using EPS, it is possible to implement several special forms typically found in Lisp systems, avoiding the need to build these forms into the compiler or interpreter. These include **eval-when**, which requires the ability to control when subexpressions are evaluated, and **macrolet** and **compiler-let**, which require control over the environment in which compilation takes place. In other cases, existing macros may be simplified or made more efficient using EPS. For example, Common Lisp's **defun** is typically implemented as a macro that must fully expand its body to resolve ambiguities related to declarations and documentation strings. With EPS, **defun** may be implemented with an expander that first expands the body with another expander and then inspects the result, thereby avoiding reexpansion of the body.

Although EPS is a substantial improvement over the traditional approach to syntactic extension, it does present a few difficulties. We have shown instances in which programming with expanders is error prone; it is easy to perform unwanted, or even infinite, expansions. However, we have also shown that traditional macro expansion facilities, such as **defmacro**, can be provided in the more general EPS context. Thus there is no harm in making this power available, for it need only be used when its added expressiveness is needed.

A more important problem is that expanders cannot determine which subexpressions are syntactic extensions, because the internal representation (as lists) is the same for syntactic extensions and applications. A straightforward solution is to pass expansion functions a syntax table that binds keywords to expansion functions. Syntax table operations could then include determining whether a given symbol is bound in the table and retrieving the expansion function bound to a symbol if there is one. However, EPS expanders have two significant advantages over syntax tables alone:

- Expanders can naturally examine (and expand if necessary) forms which do not begin with keywords, such as variables, applications, and literals. Hooks for the expansion of these forms could be added to a mechanism based on syntax tables, but this would not be as natural or as elegant as the expander mechanism.
- Expanders may lexically or dynamically bind information used to control expansion of subexpressions.

The best solution may be to pass both syntax tables and expanders, in order to give the full power of both.

References

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

[2] Dybvig, R. K., *The Scheme Programming Language*, Prentice-Hall, 1987.

[3] Liongosari, Edy S., "Scheme Debugger User's Manual" Undergraduate Honors Thesis, Indiana University (July, 1987).

[4] Dybvig, R. K., Friedman, D. P., and Haynes, C. T., "Expansion-passing style: beyond conventional macros," *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming,* Cambridge, Massachusetts, pp. 143–150, 1986.

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

[6] Kohlbecker, E., *Syntactic Extensions in the Programming Language Lisp,* PhD dissertation, Indiana University, August, 1986.

[7] Kohlbecker, E., Friedman, D., Felleisen, M., and Duba, B., "Hygienic macro expansion," *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming,* Cambridge, Massachusetts, pp. 151–161, 1986.

[8] Kohlbecker, E., and Wand, M., "Macro-by-example: deriving syntactic transformations from their specifications," *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages,* Munich, Germany, pp. 77–84, 1987.

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

[10] Rees, J., and Clinger, W. (Eds.), Revised^3 Report on the Algorithmic Language Scheme, *SIGPLAN Notices,* **21**, 12, pp. 37–79, 1986.

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

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

[13] Stoy, J., *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory,* MIT Press, 1977.