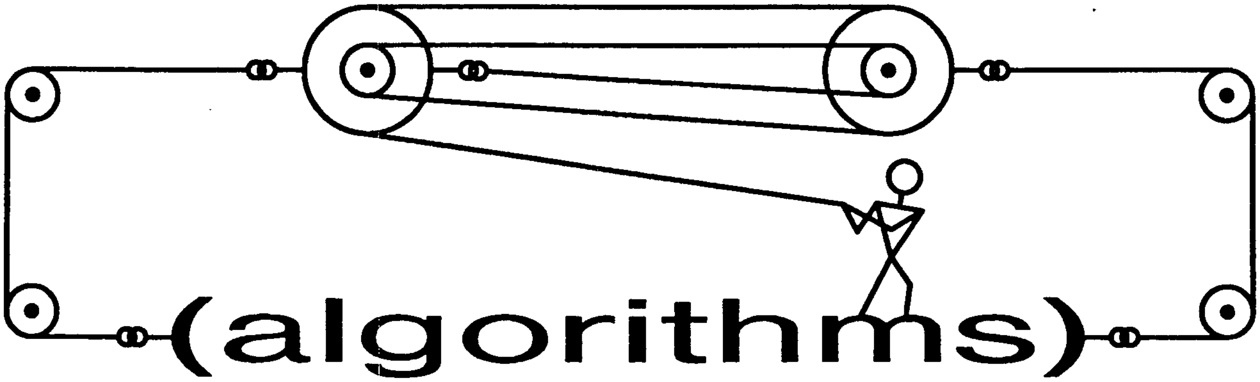

(algorithms)

Pavel Curtis, editor

Xerox PARC
3333 Coyote Hill Rd.
Palo Alto, CA 94304

Pavel.pa@Xerox.Com

I have not been deluged with submissions for this column. I've gotten a few, but not enough to make it easy to put out another issue each quarter. Let me therefore repeat my charter:

The `(algorithms)` department consists of articles that fit into one or more of three broad categories:

- Annotated implementations of interesting and relevant algorithms; they should make particularly good or novel use of the unique features of the Lisp family of programming languages (e.g., closures, continuations, code as data, polymorphism),
- Annotated implementations of algorithms whose subject matter is the Lisp family of languages (e.g., code analysis tools, iteration facilities, generic arithmetic), and
- Discussion of performance issues, benchmarking, or implementation experiences for interesting algorithms written in or about the Lisp family of languages.

If you've got a piece of code that seems like it might be appropriate for inclusion here, or if you've written an article on such an algorithm or piece of code, please send it along to one of the addresses given above. If I agree that it's appropriate and it's a complete article, I'll print it in place of the column, as I did last issue. If it's not so polished, or even if it's simply a modestly commented piece of raw seething Lisp code, I'll write a column around it, as I did two issues ago and do this issue. In fact, a major portion of this issue's column was generated from an electronic mail discussion; the ideas weren't even polished enough to be real working code yet!

So give your code, ideas, or article a chance at a spot in this column; we could be reading about *your* code next issue!

Our topic for this issue is the `setf` facility of Common Lisp, the mechanism by which so-called "generalized variables" can be read and written.

Surely all Common Lisp programmers are aware of the existence of the `setf` macro itself and most of its cousins, such as `incf`, `push`, and pop, but dramatically fewer hackers have ventured much further into the details of the facility. Probably, several of you have used the short form of `defsetf` to associate a "setting" function with a particular "getting" function. A somewhat smaller group have used either the longer form of `defsetf`, to define a more complex setter macro, or `define-modify-macro`, to create new macros like `incf` and `decf`. It seems, however, that *very* few people have plunged into the depths of true "`setf` methods", those groups of five values passed around by constructs like `define-setf-method` and `get-setf-method`.

In this column, we take a trip into the detailed depths of `setf` methods, noting along the way just a few of the odd and marvelous things that can be accomplished with them.

We'll start with a couple of new macros, implemented in Common Lisp by Rick Harris, that use generalized variables in new ways. `Locf` creates "locatives", anonymous handles through which you can get and set the contents of any generalized-variable reference. `Letf` binds generalized variables in much the same way that `let` binds special variables.

Then we'll move on to creating new kinds of generalized variables. I'll show useful `setf` methods for expressions involving such operators as `cons`, `list`, and `quote`; these enable, among other things, a pattern-matching assignment statement using backquote. Finally, we'll see how a proper `setf` method for `values` would yield a kind of "`multiple-value-setf`" if only `setf` itself had been slightly more broadly defined.

Before we see our first sights, however, let's recall the major points of the discussion on pages 104–107 of *Common LISP: The Language*. Given a particular generalized-variable reference, an expression that extracts a value from some location, we can derive a corresponding "`setf` method". The `setf` method explains how to store into that location and also how to evaluate the subexpressions of the reference form. The subexpression information is needed to properly implement expressions like this:

```
(incf (car (foo)))
```

A naive way to expand this expression is

```
(rplaca (foo) (1+ (car (foo))))
```

but this is incorrect if the function `foo` has side-effects; `foo` is called too many times. Thus, the `setf` method for `(car (foo))` must explain somehow that `(foo)` is an expression to be evaluated only once. A correct expansion would look something like this:

```
(let* ((tv (foo))
       (sv (1+ (car tv))))
  (progn
    (rplaca tv sv)
    sv))
```

The information used by the implementation of `incf` to produce such an expansion is the `setf` method of the form `(car (foo))`.

One of the things that is "generalized" in a "generalized variable" is that is need not contain just one value. Since the value of a generalized-variable reference is provided by a more-or-less arbitrary expression, and since expression can return multiple values, generalized variables can "contain" multiple values. The structure of `setf` methods is defined so as to account for this possibility. A `setf` method is represented as a set of five values:

- A list of *temporary variables.*
- A list of *value forms*, subexpressions of the generalized-variable reference.
- A list of *store variables*, more temporary variables.
- A *storing form*, evaluated to update the generalized variable.
- An *accessing form*, evaluated to fetch the value(s) of the generalized variable.

The idea is that, to do anything with the generalized variable, one should first bind the *temporary variables* to the results of the *value forms*; there must be the same number of each. Within those bindings, then, one can evaluate the *accessing form* as often as desired to fetch the (then current) value(s) of the generalized variable. In order to update the generalized variable, one must also bind the *store variables* to the value(s) one wishes to store; within all of the bindings, then, one evaluates the *storing form* to actually perform the modification. The *storing form* should return the value(s) that was (were) stored.

I realize that this all sounds very complex, but it's not really that bad. Let's look at a couple of concrete examples, beginning with the expression `(car (foo))` from above. The only subexpression is `(foo)`, so we'll only need a single *temporary variable*; call it `tv`. This generalized variable, the `car` slot of whatever `(foo)` returns, can only accept a single value, so we'll have just one *store variable*; call it `sv`. The *storing form* should put `sv` into the `car` of `tv` and then return `sv`; the expression

```
(progn (rplaca tv sv)
       sv)
```

does that. Finally, the *accessing form* simply fetches the car of `tv`:

```
(car tv)
```

Thus, a correct `setf` method for the generalized-variable reference `(car (foo))` is the following:

- `(tv)`
- `((foo))`
- `(sv)`

-
  ```
  (progn (rplaca tv sv)
         sv)
  ```
- `(car tv)`

If it looks like there are too many parentheses in the second item above, recall that it is the *value forms*, a *list* of subexpressions. By looking for these items in the expansion given for `incf` above, one can begin to get an idea of how such macros are implemented.

It should be noted that `setf` methods usually use `gensym`ed variable names, rather than `tv` and `sv`, to avoid the possibility of naming conflicts. I'll continue to use more readable names in the examples, though, so as not to further complicate things.

For a second example of a `setf` method, suppose that in a particular window system the function `window-size` takes a window as an argument and returns two values, real numbers representing the height and width of that window, respectively. Suppose further that windows are represented by `defstruct`-defined structures with two fields named `width` and `height`, among others. Thus, `window-size` could be implemented as follows:

```
(defun window-size (w)
  (values
    (window-height w)
    (window-width w)))
```

Let's now design the `setf` method for an example use of `window-size`, say `(window-size (frotz))`. As before, there is only one subexpression, so we have just one *temporary variable*, called `tv` as before. For symmetry with what `window-size` returns, however, we assume that we get *two* values to store, the new height and width. Thus, we need two *store variables* this time, one to hold each value; call them `sv1` and `sv2`. The *storing form* should set both height and width appropriately and return those same values:

```
(values
  (setf (window-height tv) sv1)
  (setf (window-width tv) sv2))
```

Finally, the *accessing form* is again simple, just `(window-size tv)`. The complete `setf` method for `(window-size (frotz))` is thus these five values:

- `(tv)`
- `((frotz))`

```
(defmacro setf (place expr &rest more-pairs)
   (if (not (null more-pairs))
      '(progn
          (setf ,place ,expr)
          (setf ,@more-pairs))
      (multiple-value-bind (tvars vals svars store access)
                           (get-setf-method place)
         '(let* (,@(mapcar #'list tvars vals)
                 (,(car svars) ,expr))
             ,store))))
```

Figure 1: An implementation of the `setf` macro of Common Lisp

- `(sv1 sv2)`
- `(values (setf (window-height tv) sv1) (setf (window-width tv) sv2))`
- `(window-size (frotz))`

Now that we have some understanding of what's in a `setf` method, let's look at some macros that use them, beginning with `setf` itself. There are two functions in Common Lisp that return `setf` methods:

- `get-setf-method`
- `get-setf-method-multiple-value`

Both of these take a single argument, the generalized-variable reference whose `setf` method is desired.[1] They each return the five values comprising the `setf` method for the given form. The only difference between the two is in their treatment of `setf` methods with more than one *store variable*. Because for many macros it only makes sense for a single value to be expected (`incf` is such a macro), it was decided that there was a need for a function that checked the `setf` method before returning it, checking that exactly one *store variable* was provided; `get-setf-method` is that function. The other function, `get-setf-method-multiple-value`, is less discriminating; it returns the `setf` method regardless of the number of *store variables*.

The convenience of `get-setf-method`'s built-in check is nice, but unfortunately the designers went one step further: they erroneously decided that there *were* no macros in Common Lisp itself for which multiple *store variables* made sense. This means that none of the `setf`-like macros in Common Lisp accept generalized variables containing more than one value, like our `window-size` function above. Even the simple expression

```
(setf (window-size win)
      (values h w))
```

is illegal in Common Lisp as defined in the silver book. I'll have more to say about this near the end of the article; for now, let's look at `setf` as currently defined.

The implementation of `setf` is quite simple; see Figure 1. After dealing with the case where more than one place/value pair was given,[2] we merely call `get-setf-method` to analyze the

---

[1] Actually, in the version of Common Lisp being standardized by ANSI, these functions may have an optional second argument, the `&environment` in which any macros should be expanded during the search for a `setf`-able generalized-variable reference. I will ignore this issue in this article.

[2] The astute (or perhaps obscure) reader will have noticed that this implementation does not arrange for `(setf)` to evaluate to `nil`, as required by the specification. I leave this "generalization" to the reader.

```
(defmacro locf (place)
   (multiple-value-bind (tvars vals svars store-form access-form)
                        (get-setf-method place)
      '(let ,(mapcar #'list tvars vals)
          (make-locative
             :access-fn #'(lambda ()
                             ,access-form)
             :modify-fn #'(lambda ,svars
                             ,store-form)))))
```

Figure 2: Implementation of `locf`.

generalized-variable reference and then put the pieces of the `setf` method together in a straightforward manner. We construct a `let*` in which the *temporary variables* are bound to the *value forms* and the single *store variable* is bound to the given expression. Within these bindings, we simply evaluate the *storing form* to effect the assignment. As an example, the form `(setf (car (foo)) (bar))` would macro-expand into this:

```
(let* ((tv (foo))
       (sv (bar)))
   (progn
      (rplaca tv sv)
      sv))
```

Let's move on now to the first of our new uses for this machinery, a Common Lisp implementation of the `locf` macro from Symbolics Zetalisp, sent to me by Rick Harris from the Rensselaer Polytechnic Institute. The idea of `locf` is that it maps a generalized-variable reference into a so-called "locative" for that location. The function `location-contents` takes a locative and returns the contents of the corresponding generalized variable. To set that generalized variable, one uses `setf` of `location-contents`. With such a facility, one could, for example, rewrite a macro like `incf` as a function:

```
(defun increment (loc)
   (incf (location-contents loc))
```

Let `x` and `loc` be defined as follows:

```
(setq x   (cons 1 2)
      loc (locf (car x)))
```

After evaluating `(increment loc)` a couple of times, the value of `x` would be `(3 . 2)`. In effect, `locf` creates an anonymous handle on a particular generalized variable, in this case the `car` of `x`. This sort of thing is handy when your program must work with many types of data and you don't want to write a lot of special-purpose code. Instead of passing the actual data objects around, you create and pass locatives to the relevant slots in them. The receiving code can manipulate the slot without knowing what kind of object it's in.

Enough motivation, let's look at the implementation. The essential idea is that `locf` creates a structure containing two functions, one that returns the value of the generalized variable and one that sets that value. These functions are closures created from the information in the `setf` method of the given form; see Figure 2. Note that the *value forms* are evaluated at the time the locative is created, not every time it's used; this ensures that any side-effects they might have are not duplicated. The `locative` structure type is defined by `defstruct` in the obvious way:

```
(defstruct locative
   access-fn
   modify-fn)
```

The function `location-contents` merely `funcalls` the access function of the given locative.

```
(defun location-contents (locative)
  (funcall (locative-access-fn locative)))

(defsetf location-contents (locative) (new-value)
  (funcall (locative-modify-fn locative) new-value))
```

Figure 3: Implementation of the operations on locatives.

```
(defun execute-process (process)
  (let ((machine (find-free-machine)))
    (letf (( (current-process machine)  (process-id process) ))
      (run process machine))))
```

Figure 4: A more-or-less typical use of `letf`.

The `setf` method for `location-contents` is about as simple; we `funcall` the `modify-fn` instead, passing along the new value for the location. See Figure 3 for the details.

I'm honor-bound to mention a few things about this implementation. The most important is that I've left out some pieces of Rick's code in order to simplify the presentation. There are a few ways in which what he wrote is more efficient and more debuggable than what I've shown here. Also, for compatibility with the original Zetalisp construct, he treats places of the form `(cdr ...)` as a non-consing special case. His complete source is given at the end of the article.

It also bears mentioning that a Common Lisp implementation of locatives cannot be as efficient as in Zetalisp, in which they are represented as immediate pointers to memory cells. On the other hand, the Zetalisp implementation can't handle every kind of generalized variable, so perhaps the advantages balance out somewhat. In any case, before using this Common Lisp `locf` one should be aware that the performance characteristics will be radically different from Zetalisp's construct.

The other `setf` method client we'll look at was also programmed by Rick Harris and is also taken in part from Zetalisp; it is the `letf` construct for "dynamically binding" arbitrary generalized variables.

You've almost certainly had a use for this at some point, even if you weren't aware of it at the time. There's some slot in a structure or some such that you want to have a different value during the execution of some piece of code and you want to restore the old value when you're done. So what do you do? You bind a new variable to the old value, put the new value in the location, and execute the relevant code inside an `unwind-protect` that puts the old value back at the end. `Letf` encapsulates this idiom in a convenient notation through the use of `setf` methods. Figure 4 shows a typical use of the macro.

As with other binding forms, like `let`, the `letf` construct can take an arbitrary number of place/value pairs to bind. All of the values to be bound are computed before any of them are assigned, as with `let`. The implementation is somewhat more complex than that of `locf`, almost entirely because of the arbitrary number of generalized-variable references; see Figure 5.

For each binding in the `letf` form, we call `get-setf-method` to analyze the generalized-variable reference and accumulate the results in

```
(defmacro letf (bindings &body body)
   (let ((tvar-list nil)
         (val-list nil)
         (svar-list nil)
         (store-list nil)
         (access-list nil)
         (bound-exprs (mapcar #'cadr bindings))
         (save-vars (mapcar #'(lambda (ignore) (gensym)) bindings)))

      (dolist (binding bindings)
         (multiple-value-bind (tvars vals svars store access)
                              (get-setf-method (car binding))
            (setq tvar-list (nconc tvar-list tvars))
            (setq val-list (nconc val-list vals))
            (setq svar-list (nconc svar-list svars))
            (setq store-list (nconc store-list (list store)))
            (setq access-list (nconc access-list (list access)))))

      '(let* (,@(mapcar #'list tvar-list val-list)
              ,@(mapcar #'list save-vars access-list))
          (unwind-protect
                (let ,(mapcar #'list svar-list bound-exprs)
                   ,@store-list
                   ,@body)
             (let ,(mapcar #'list svar-list save-vars)
                ,@store-list)))))
```

Figure 5: The implementation of `letf`.

several lists. The expansion first binds all of the *temporary variables* to all of their respective *value forms*, and then saves the initial values of all of the generalized variables in `gensym`ed variables. Within an `unwind-protect`, the bound value expressions are evaluated and bound to the various *store variables* and all of the *storing forms* executed. This gets everything ready to evaluate the body of the original `letf`. Finally, the clean-up part of the `unwind-protect` again binds the *store variables*, this time to the variables holding the saved values of the bound locations, and within those bindings evaluates the *storing forms* again to restore the old values of the generalized variables.

Well, that's a bit of a mouthful, so let's look at the expansion of an example that uses our `setf` method for `(car (foo))` from before. The `letf` form

```
(letf (( (car (foo))  (bar) ))
  (body-stuff))
```

expands into a form like this:

```
(let* ((tv   (foo))
       (#:g1 (car tv)))
  (unwind-protect
      (let ((sv (bar)))
         (progn (rplaca tv sv) sv)
         (body-stuff))
    (let ((sv #:g1))
      (progn (rplaca tv sv) sv))))
```

Sure enough, this saves the old value of the `car` of `(foo)`, puts `(bar)` in there during the execution of the body, and restores the old value upon exit, just like the doctor ordered.

Again, I've changed Rick's code to simplify the presentation and, again, you can see his more complete implementation of the Zetalisp original at the end of the article. There is one significant difference between this `letf` and the one in Zetalisp, when used in a multiprocessing Lisp implementation. In Zetalisp, the old value is restored every time there's a context switch to allow some other process to run and the new value is put back when control returns to the binding process. In this way, the bindings made by `letf` are identical to normal special variable bindings. Since Common Lisp doesn't say anything about multiple processes, we can't portably implement that special behavior. For single-process applications, though, this `letf` can be used pretty much everywhere that Zetalisp's can.

Let's move on now to consider new kinds of generalized variables. All of those defined in Common Lisp are simple data structure accessors, like `car`, `gethash`, `symbol-plist`, etc. This isn't a requirement for *all* generalized variables, though. The contract of a generalized variable is simply that it must behave like a variable: after storing a set of values into it, evaluating the access form must yield the "same" values. I put "same" in quotes here because there isn't a consistent view of the kind of equality to use here. For most of the built-in generalized variables, like `car` and `gethash`, a predicate like `eql` is probably intended, but for others, like `subseq`, none of the Common Lisp equality predicates is appropriate. The notion of "sameness" is dependent on the kind of generalized variable.

This "variable-like" behavior, though, is the entire requirement on a new kind of generalized variable. As Jonathan Rees mentioned to me, for example, such variables might not have anything to do with the memory of the running Lisp system. Here are some compelling examples of this sort of generalized variable:

```
(setf (file-length "index.hash")
      4096)

(setf (host-address "gidney")
      (generate-new-ip-address))
```

Operations on generalized variables also need not be computationally cheap. For example, suppose that the function `mvmult` multiplies a matrix by a vector, yielding a new vector. Then the form

```
(setf (mvmult A x) b)
```

might solve systems of linear equations in order to find `x`, given `A` and `b`. The possibilities are truly endless.

One interesting new kind of generalized variable, suggested to me by Kent Pitman, is *value constructors*, like `cons`. What should an expression like `(cons a b)` *mean* as a generalized-variable reference? Well, since it behaves like a variable, we know that after evaluating an expression like

```
(setf (cons a b) (foo))
```

the value of `(cons a b)` will be the "same" as whatever `foo` returned. If we let "same" mean `equal` here, then `a` must now hold the `car` of `foo`'s result and `b` must hold the `cdr`. Thus, assigning to a generalized-variable reference of the form

(cons *place*$_1$ *place*$_2$)

should *destructure* the assigned value into the two *places*. How handy! But can we implement this behavior in Common Lisp? It turns out that it isn't very difficult at all.

The only direct way to add a new `setf` method to Common Lisp is with the `define-setf-method` form. It has this syntax:

(define-setf-method *name pattern*
  body)

```
(define-setf-method cons (x y)
   (let ((svar (gensym)))
      (values
         '()
         '()
         (list svar)
         `(progn
             (setq ,x (car ,svar))
             (setq ,y (cdr ,svar))
             ,svar)
         `(cons ,x ,y))))
```

Figure 6: A simplified **setf** method definition for cons.

```
(define-setf-method cons (x y)
   (let ((svar (gensym)))
      (multiple-value-bind (x-tvars x-vals x-svars x-store x-access)
                           (get-setf-method x)
         (multiple-value-bind (y-tvars y-vals y-svars y-store y-access)
                              (get-setf-method y)
            (values
               (append x-tvars y-vars)               ; temporary variables
               (append x-vals y-vals)               ; value forms
               (list svar)                           ; store variables
               `(let ((,(car x-svars) (car ,svar))   ; storing form
                      (,(car y-svars) (cdr ,svar)))
                   ,x-store
                   ,y-store
                   ,svar)
               `(cons ,x-access ,y-access))))))     ; accessing form
```

Figure 7: The complete **setf** method definition for cons.

where *pattern* is a `defmacro`-style argument list. This tells the `setf` facility to call this code whenever it needs the `setf` method for a generalized-variable reference whose `car` is the symbol *name*. The *body* should return five values, the ones we've been using all along to represent `setf` methods.

We'll begin with a simplification of the real `setf` method for cons; we'll assume for the moment that *place*$_1$ and *place*$_2$ have to be simple variables. Thus, we're only going to deal with generalized-variable references like `(cons a b)` and not more complex uses like

```
(cons (aref x 12)
      (gethash 17 ht))
```

For this simple case, the `setf` method definition is easy to write; it appears in Figure 6.

There can't be any subexpressions with side-effects, so we don't need any *temporary variables* or *value forms*. We use `gensym` to get a fresh *store variable* and then it's easy to write the *storing form*; it simply assigns the car and cdr of the value to x and y. We also have to remember to return the stored value from the *storing form*. The *accessing form* is trivial, identical to the original reference.

So the easy case is easy, but what must be done to accommodate the general case, hairy references like the one involving `aref` and `gethash` above? Since we've constructed a new generalized-variable reference (using `cons`) out of two others (using `aref` and `gethash`), we should expect that we'll construct a new `setf` method out of two others. We'll need to call `get-setf-method` to analyze the two subforms of `cons` for us; the details appear in Figure 7.

The subexpressions of a generalized-variable reference using cons are those of the two arguments. Thus, the *temporary variables* and *value forms* for our `setf` method are simply the concatenation of those for the subforms x and y. The *store variable* is as before and, in a sense, so is the *storing form*. We must first set up the bindings of the *store variables* for the subforms, but then it's just as in the simple case above; we store into x and y (we have to use their *storing forms* instead of simple `setq`s) and finally return the value that was stored. Similarly, our *accessing form* is like the one in the simple case except that we use the *accessing forms* of x and y.

This fully general `setf` method for `cons` allows us to do some complex kinds of destructuring:

```
(setf (cons a (cons b (cons c d)))
      '(1 2 3 4))
```

assigns 1 to `a`, 2 to `b`, 3 to `c`, and the list `(4)` to `d`. We could go further and define `setf` methods for the other value constructors of Common Lisp, like `vector` and the constructor functions defined by `defstruct`. There's one constructor that's particularly amusing in this context: backquote.

Consider the expression `` `(a ,b c) `` as a generalized-variable reference for a moment. After evaluating the odd-looking assignment

```
(setf `(a ,b c) (foo))
```

the expression `` `(a ,b c) `` should evaluate to the result of calling `foo`. For this to be true, `foo` must have returned a list of length 3 whose first element is the symbol `a`, whose second element was stored into the variable `b` and whose third element is the symbol `c`. But what if the result of `foo` isn't a list, or that list is not three elements long, or the first and third elements aren't `a` and `c`? Then surely an error should be signalled since the assignment cannot be carried out correctly, right? In a way, this would amount to a kind of "pattern-matching" assignment statement, a useful addition to the language. Let's look at how we could achieve this.

First, what is the expansion of an expression like `` `(a ,b c) ``? Common Lisp does not specify this exactly, but on most systems backquoted expressions expand into normal expressions made out of operators like `cons`, `list`, `list*`, and `quote`. For example, in Xerox Lisp, `` `(a ,b c) `` expands into `(list 'a b 'c)`. Thus, in that particular implementation, we would be concerned with the `setf` methods of `list` and `quote`.

Any call to `list` can be rewritten as a set of nested calls to `cons`, whose `setf` method we've already defined. Fortunately, it's easy to take advantage of this to define a `setf` method for `list`, as shown in Figure 8. We simply return the `setf` method for the rewritten expression instead of computing one ourselves. There's a problem here, though. When you rewrite `(list 'a b 'c)` in this way, you get

```
(cons (quote a)
      (cons b
            (cons (quote c)
                  nil)))
```

We were about to define a `setf` method for `quote`, but what about that `nil` down there at the end? We will end up asking for a `setf` method for it and `setf` is likely to think it's just a simple variable, which is just the wrong thing. We'll come back to this in a moment.

```
(define-setf-method list (&rest args)
   (get-setf-method (reduce #'(lambda (arg form)
                                 `(cons ,arg ,form))
                             args
                             :from-end t
                             :initial-value nil)))
```

Figure 8: The setf method definition for list.

```
(defun fancy-get-setf-method (form)
   (if (constantp form)
       (let ((svar (gensym)))
          (values
             nil                       ; temporary variables
             nil                       ; value forms
             (list svar)               ; store variables
             `(progn                   ; storing form
                 (assert (equal ,form ,svar) ()
                    "pattern-matching failed: ~S should have been ~S"
                    ,svar ,form)
                 ,form)
             form))                    ; accessing form
       (get-setf-method form)))
```

Figure 9: An enhanced version of get-setf-method

The `setf` method for `quote` is a bit stranger. After evaluating

```
(setf (quote a) (foo))
```

we need `(quote a)` to yield the same value that `foo` returned. That is, we need `foo` to return the symbol `a`. We don't need to do any actual assignments to make this true, we just have to check the given value. If `sv` were our *store variable*, then this would make a good *storing form*:

```
(progn
  (assert (equal sv 'a) ()
     "~S should be ~S" sv 'a)
  'a)
```

This `setf` behavior shouldn't be peculiar to `quote`, though; the `setf` method for *every* constant expression should be like this, including that pesky `nil` from above. The problem with this observation is that Common Lisp doesn't have a way for us to specify a `setf` method for expressions like `17` and `"a string"`.

In order to accomplish our desired `setf` behavior for backquoted expressions, we'll have to substitute our own function in place of `get-setf-method`. Fortunately, constant expressions are only useful as generalized-variable references when they're arguments to value constructors. Since we're writing all of those `setf` method definitions, we can simply have them call our function to do their analysis instead of `get-setf-method`. I've called this new function `fancy-get-setf-method`; its definition is simple and appears in Figure 9.

We now have everything in place to compute

```
(let* ((#:g32 (foo)))
  (let ((#:g33 (car #:g32))
        (#:g34 (cdr #:g32)))
    (progn
      (assert (equal 'a #:g33) ()
              "pattern-matching failed: ~S should have been ~S"
              #:g33 'a)
      #:g33)
    (let ((#:g35 (car #:g34))
          (#:g36 (cdr #:g34)))
      (setq b #:g35)
      (let ((#:g37 (car #:g36))
            (#:g38 (cdr #:g36)))
        (progn
          (assert (equal 'c #:g37) ()
                  "pattern-matching failed: ~S should have been ~S"
                  #:g37 'c)
          #:g37)
        (progn
          (assert (equal nil #:g38) ()
                  "pattern-matching failed: ~S should have been ~S"
                  #:g38 nil)
          #:g38)
        #:g36)
      #:g34)
    #:g32))
```

Figure 10: The macroexpansion of ``(setf `(a ,b c) (foo))``.

the expansion of our original pattern-matching assignment statement; see Figure 10. Pretty amazing, eh? Looking at it makes one realize that it might be worth putting that `assert` expression into a separate function. If you squint hard enough, you can actually find the assignment to `b` there in the middle of the code.

For our final stop of this whirlwind tour of `setf` method applications, let's consider one last kind of constructor, the `values` function. This is somewhat like the others, but it has an arbitrary number of subforms, each of which should be a generalized-variable reference, and it involves the use of multiple *store variables.* The code appears in Figure 11.

The part that deals with analyzing the arbitrary number of argument places is almost identical to that part of the implementation of `letf`; the five values of each constituent `setf` method are accumulated in lists for later use. The `setf` method itself has all of the *temporary variables* and *value forms* from the arguments, as in the definition for `cons`.

We need as many *store variables* as there are arguments to the `values` form, since we'll be doing that many assignments. The *storing form* simply sets up all of the bindings needed by the *storing forms* for all of the arguments and then executes those forms one after another, finally returning all of the values that were stored. As in the `cons` case, the *accessing form* is straight-

```
(define-setf-method values (&rest places)
  (let ((tvar-list nil)
        (val-list nil)
        (svar-list nil)
        (store-list nil)
        (access-list nil)
        (my-svars (mapcar #'(lambda (x) (gensym)) places)))

     (dolist (place places)
        (multiple-value-bind (tvars vals svars store access)
                             (fancy-get-setf-method place)
           (setq tvar-list (nconc tvar-list tvars))
           (setq val-list (nconc val-list vals))
           (setq svar-list (nconc svar-list svars))
           (setq store-list (nconc store-list (list store)))
           (setq access-list (nconc access-list (list access)))))

     (values
        tvar-list                        ; temporary variables
        val-list                         ; value forms
        my-svars                         ; store variables
        '(let* ,(mapcar #'list           ; storing form
                    svar-list my-svars)
            ,@store-list
            (values ,my-svars))
        '(values ,@access-list)))))      ; accessing form
```

Figure 11: A complete `setf` method definition for `values`.

forward.

Unfortunately, as was mentioned much earlier, we can't use this nice `setf` method with any of the macros in Common Lisp, since it usually includes more than one *store variable.* There is some reason to hope that this will be changed in the version of the language being standardized by ANSI, though, so it may yet find acceptance. If so, we would be able to use it to say

```
(setf (values (car a)
              (gethash b 'c)
              (aref d 13))
      (some-hairy-computation))
```

which stores the first value returned by `some-hairy-computation` into the `car` of `a`, the second into the hash table `b`, and the third into the array `d`. In effect, this gives us a general `multiple-value-setf` form all as a part of the normal `setf` macro with which we're so familiar.

Well, I hope I've led you on an interesting and comprehensible trip into the possibilities inherent in the `setf` facility of Common Lisp. As a final, intriguing example, I leave you with this piece of code, enabled by our labors here:

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

After writing the `setf` method for `list*`, you might enjoy figuring out just what this sometimes useful idiom does. $\boxed{()}$

```
;;;; LOCF Implementation by Richard Harris, RPI

;;; "locf access-form                          Macro
;;;     Takes a form that accesses some cell and produces a corresponding
;;;     form to create a locative pointer to that cell.  Examples:
;;;
;;;         (locf a)          ==> #<Locative to A>
;;;         (locf (aref q 2)) ==> #<Locative to (AREF Q 2)>"

(defstruct (locative
             (:constructor make-locative (access modify name))
             (:print-function print-locative))
  access
  modify
  name)

(defun print-locative (locative stream depth)
  (declare (ignore depth))
  (format stream "#<Locative to ~S>" (locative-name locative)))

(defmacro locf (setf-form)
  (if (and (consp setf-form)
           (eq 'cdr (car setf-form)))
      (cadr setf-form)
      (multiple-value-bind (vars vals stores store-form access-form)
                           (get-setf-method setf-form)
        `(let ,(mapcar #'list vars vals)
           (make-locative
             #'(lambda ()
                 ,access-form)
             #'(lambda ,stores
                 ,store-form
                 ,(car stores))
             ',setf-form)))))

(proclaim '(inline location-contents))
(defun location-contents (locative)
  (typecase locative
    (cons
      (cdr locative))
    (locative
      (funcall (locative-access locative)))
    (t
      (error "~S is not a locative" locative))))

(defsetf location-contents (locative) (value)
  `(typecase ,locative
     (cons
       (setf (cdr ,locative) ,value))
     (locative
       (funcall (locative-modify ,locative) ,value))
     (t
       (error "~S is not a locative" ,locative))))
```

```
;;;; LETF Implementation by Richard Harris, RPI

;;; The following is really a cross between the (Symbolics) functions letf
;;; and let-globally:
;;;
;;; "letf places-and-values body...             Special form
;;;     Just like let, except that it can bind any storage cells rather than
;;;     just variables."
;;;
;;; "let-globally ((var value)...) body...      Special form
;;;     Similar in form to let.  The difference is that let-globally does not
;;;     bind the variables; instead, it saves the old values and sets the
;;;     variables, and sets up an unwind-protect to set them back."
;;;
;;; This difference is important (only) in a multiple-process Lisp system.

(defmacro letf (bindings &body forms)
  (let ((tvars nil)
        (tvals nil)
        (store-vars nil)
        (store-forms nil)
        (access-forms nil)
        (value-forms nil)
        (save-vars nil))

    (dolist (binding bindings)
      (let ((setf-form  (if (atom binding) binding (car binding)))
            (value-form (if (atom binding) nil     (cadr binding))))
        (multiple-value-bind (vars vals stores store-form access-form)
                             (get-setf-method setf-form)
          (setq tvars (nconc tvars vars))
          (setq tvals (nconc tvals vals))
          (setq store-vars (nconc store-vars stores))
          (setq store-forms (nconc store-forms (list store-form)))
          (setq access-forms (nconc access-forms (list access-form)))
          (setq value-forms (nconc value-forms (list value-form)))
          (setq save-vars (nconc save-vars (list (gensym)))))))

    `(let ,(mapcar #'list tvars tvals)
       (let ,(mapcar #'list save-vars access-forms)
         (unwind-protect
              (progn
                (let ,(mapcar #'list store-vars value-forms)
                  ,@store-forms)
                ,@forms)
           (let ,(mapcar #'list store-vars save-vars)
             ,@store-forms))))))

(defmacro letf* (bindings &body forms)
  (if (null (cdr bindings))
      `(letf ,bindings
         ,@forms)
      `(letf (,(car bindings))
         (letf* ,(cdr bindings)
           ,@forms))))
```