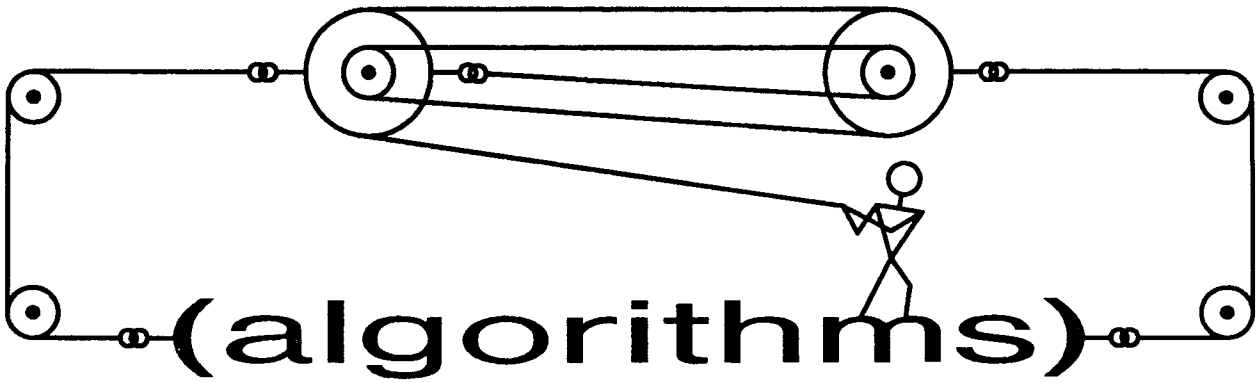

Pavel Curtis, editor

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

Pavel.pa@Xerox.Com

Now that Lisp Pointers is being sponsored by SIGPLAN, it seems worthwhile to repeat the charter of this department. 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.

So far, (algorithms) has included articles on the `extend-syntax` macro definition facility, an elegant hidden-line elimination program, Common Lisp `setf` methods, and a higher-order programming solution to the infamous `xpl` puzzle. Of these, two were written by me and two by contributing authors.

If you've been hacking on an interesting piece of code that might fit into the (algorithms) department, please send me a note at one of the addresses above. If I agree that the code is appropriate, then either you or I (or the two of us working together) will put an article about it in this space. Don't be shy, send me your ideas!

One of the most well-known and widely-used pieces of Common Lisp code is Gregor Kiczales' portable implementation of the Common Lisp Object System, PCL.[1] Among the many interesting algorithms and implementation tricks employed in that code is a very general program-analysis tool known as the "code walker". Though it is distributed along with PCL, the walker is a separate utility, and the subject of this issue's column. Gregor's walker is based, in part, on earlier code-analysis tools written by Larry Masinter, David Moon and Gary Drescher.

[1] The name PCL originally stood for "Portable CommonLoops", one of the precursors to CLOS.

The code walker is perhaps best understood in comparison to the Common Lisp function `mapcar`. `Mapcar` takes a function and a list as arguments and returns a list of the results of applying that function to the elements of the input list. The nice thing about `mapcar` is that it abstracts away all of the details of visiting each element of the input list and constructing the result list.

The main entry point into the code-walker, a function named `walk-form`, is quite similar. It takes a function (called the *walk function*) and a Common Lisp expression as arguments and returns a new expression whose subexpressions are the results of applying the walk function to each of the subexpressions of the input expression. That is, the walker recursively visits all of the subexpressions in a piece of code, applying the walk function to each of them, and puts the results of the walk function back together into new expressions. The nice thing about the walker is that it abstracts away the complex details of Common Lisp expression syntax and semantics and the reconstruction of the resulting expression.

For example, suppose that the walk function mapped every variable reference into the string `"Foo"`, but left other expressions alone. Then the walker would transform

```
(let ((x (list y)))
   (tagbody
    top
      (setq x (cons 'x x))
      (if (< (length x) 8)
         (go top)))
   (print x))
```

into the new expression

```
(let ((x (list "Foo")))
   (tagbody
    top
      (setq x (cons 'x "Foo"))
      (if (< (length "Foo") 8)
         (go top)))
   (print "Foo"))
```

Note that it didn't change *every* symbol into the string `"Foo"`, just the ones that were variable references.

This is a nice abstract model of what the walker does, but some questions arise to muddy the water a bit. First, there is the question of just what constitutes a "subexpression". Consider this expression:

```
(let ((x 17))
   (setq z (cons 1 2))
   ((lambda (y) (funcall f g))
    #'foo))
```

I think it is easy to agree on most cases; certainly `17`, `(cons 1 2)`, and the entire `let`-expression itself should be considered subexpressions. Similarly, it is pretty clear that the syntactic keywords `let` and `setq` are *not* subexpressions by themselves.

Some other cases are less clear-cut, though. What about the binding uses of `x` and `y`? In the walker, these are not considered subexpressions because no evaluation takes place when the interpreter encounters them; binding occurrences serve only to establish meanings for identifiers as opposed to making use of those meanings.

On the other hand, the use of `z` in the `setq`-expression *is* considered a subexpression by the walker, albeit in a different sense than, say, the use of `f`.

Finally, what about those forms in functional context: `cons`, `funcall`, `foo`, and the `lambda`-expression? Certainly they involve some evaluation, but once again in a different sense than other subexpressions.

The walker handles the variety of kinds of subexpressions by passing a "context" argument to the walk function along with each subexpression, noting whether it appeared in `:eval`, `:set`, or `:call` position.

There is a more significant difference, though, between the lists manipulated by `mapcar` and the expressions given to `walk-form`. The elements of a list are completely separate; while they may share structure, elements are not nested within each other. Thus, if the input

function has no side-effects, it does not matter in what order `mapcar` examines the elements. This is not the case for Common Lisp expressions, the subexpressions of which are frequently and necessarily nested. There are two interesting orders in which `walk-form` might apply the walk function to the subexpressions.

The walk function might be applied to each expression *before* any of its subexpressions. Let E be the original expression and let E' be the result of applying the walk function to E. Since the walker is supposed to return the results of applying the walk function to every expression, it must traverse the subexpressions of E', not the original, E.

For example, suppose that E was `(car x)` and E' was `(+ a b)`. If the walker were to apply the walk function to `car` and `x`, what could it do with the results? It should instead walk `+`, `a`, and `b`; it is easy to see how to combine those results to return them.

Alternatively, the walk function might be applied to each expression only *after* visiting each of its subexpressions. As in the other case, we need a way for *all* of the results of the walk function to have some effect on the final result of the walk.

Suppose that the walk function maps the expressions E_1 and E_2 into E_1' and E_2', and maps `cons` into `list` when it appears in `:call` position. If the walker is presented with the expression `(cons` E_1 E_2`)` it must first visit `cons`, E_1 and E_2. Having done so, it would be useless to apply the walk function to `(cons` E_1 E_2`)`. Instead, of course, it must be applied to `(list` E_1' E_2'`)`.

Neither of these orders is obviously best for all possible uses of a code walker. As it turns out, the first order is most convenient for the uses Gregor had in mind when he wrote the walker, so that is the one implemented. At the end of the article, we examine a possible change, suggested by Gregor, that enables both orders (and, indeed, any combination of the two) in a simple and natural fashion.

Speaking of possible uses for a walker, it behooves me to show you one or two. Before doing so, though, we need some more of the details of the walker's contract.

The function `walk-form` takes three arguments: a Common Lisp expression to walk, an environment (such as is provided by an `&environment` argument to a macro expander), and the walk function itself; it returns the new expression resulting from the walk. The environment argument is necessary so that the walker can correctly recognize and expand any macros it might encounter.

The walk function takes three arguments: the current expression in the walk, the evaluation context of that expression (one of `:eval`, `:set`, or `:call`), and the environment of the expression. The environment is provided so that the walk function can discover certain facts about the bindings around the expression. We'll come back to this possibility later. The walk function should return two values: the new expression and a flag indicating whether or not the walker should continue walking this expression. If the flag is true, the walker will not continue; this makes the usual case a bit more concise since the walk function can simply return the new expression, letting the flag default to `nil`.

The first example is a simpleminded (and not quite correct) macro implementation of a new special form, planned for inclusion in the ANSI Common Lisp standard when it appears. The form is called `symbol-macrolet` and has the following syntax:

```
(symbol-macrolet ((name form)
                  ...)
  body)
```

The idea is that the symbol *name* should be replaced by the expression *form* wherever it is referenced within *body*; you can think of it as a kind of "inline" definition of a new lexical variable *name*. If *name* appears in an assignment expression, `(setq` *name expr*`)`, that expression should be interpreted as `(setf` *form expr*`)`. A similar transformation should be

```
(defmacro symbol-macrolet (bindings &body body &environment env)
   (walk-form `(progn ,@body) env
      #'(lambda (form context env)
           (sm-walk-fn form context env bindings))))

(defun sm-walk-fn (form context env bindings)
   (cond ((not (eq context ':eval))
          form)
         ((symbolp form)
          (let ((entry (assoc form bindings)))
             (if entry
                (cadr entry)
                form)))
         ((atom form)
          form)
         ((member (car form) '(setq setf))
          (let ((kind (car form)))
             (labels ((scan-pairs (tail)
                         (if (null tail)
                            nil
                            (let ((entry (assoc (car tail) bindings)))
                               (list*
                                  (if entry
                                     (progn (setq kind 'setf)
                                            (cadr entry))
                                     (car tail))
                                  (cadr tail)
                                  (scan-pairs (cddr tail)))))))
                (let ((new-tail (scan-pairs (cdr form))))
                   (cons kind new-tail)))))
         ((eq (car form) 'multiple-value-setq)
          (let* ((vars (cadr form))
                 (gensyms (mapcar #'(lambda (i)
                                      (declare (ignore i))
                                      (gensym))
                                  vars)))
             `(multiple-value-bind ,gensyms
                                   ,(caddr form)
                 ,@(mapcar #'(lambda (v g) `(setf ,v ,g))
                           vars
                           gensyms))))
         (t form)))
```

Figure 1: A simpleminded macro implementation of `symbol-macrolet`.

```
(defmacro symbol-macrolet (bindings &body body &environment env)
   (let ((bindings (mapcar
                    #'(lambda (binding)
                        (list (car binding)
                              (cadr binding)
                              (variable-binding env (car binding))))
                    bindings)))
      (walk-form '(progn ,@body) env
         #'(lambda (form context env)
             (sm-walk-fn form context env bindings)))))
```

Figure 2: A more correct implementation of `symbol-macrolet`.

made if it appears in a `multiple-value-setq` form. The implementation appears in Figure 1.

This code is pretty straightforward; the walk function only transforms expressions in `:eval` context, and not all of those. It simply tests for the ones it affects and returns the others unchanged. There are a few problems with this code however:

- The *form* should only be substituted for *name* over the scope of a normal lexical binding. Thus

```
(symbol-macrolet ((x (car y)))
   (list x
      (let ((x (1+ x)))
         x)))
```

  should be equivalent to

```
(progn
   (list (car y)
      (let ((x (1+ (car y))))
         x)))
```

  but this code would return

```
(progn
   (list (car y)
      (let ((x (1+ (car y))))
         (car y))))
```

- If the Common Lisp implementation expands uses of `multiple-value-bind` into a use of `multiple-value-setq`, this code will loop forever, walking and rewalking the same expression. This doesn't seem very likely, however.

- According to the official specification of `symbol-macrolet`, the environment used for macro expansion inside the body of the form is supposed to contain the bindings of the new symbol macros, so that other macros used in the body can correctly expand them.

Solving the last two problems is beyond the scope of this article, but the first one is more relevant; this kind of issue is the reason that the environment is passed to the walk function.

We would like to be able to tell if one of the names in the `symbol-macrolet` form is rebound during the walk and then avoid any substitutions inside that new binding. The walker provides a utility function, `variable-binding`, that takes an environment and a symbol and returns a representation of the binding of that symbol in that environment. Nothing is guaranteed about the representation except that it is different from the representation of any other bindings of that variable.

Figure 2 shows a new definition of `symbol-macrolet` which uses this facility to help fix the problem described earlier. The function `sm-walk-fn` must also change, of course, but the differences are small. In both places where

```
(defmacro with-constant-folding (&body body &environment env)
  (walk-form '(progn ,@body) env
    #'(lambda (form context env)
        (if (and (eq context ':eval)
                 (consp form)
                 (member (car form) '(+ - * /))
                 (every #'constantp (cdr form)))
            (eval form)
            form))))
```

Figure 3: A naive implementation of `with-constant-folding`.

`(if entry ...)` is tested, the predicate should change to

```
(and entry
  (eq (caddr entry)
      (variable-binding env form)))
```

We thus arrange to perform the substitution only when the name is in the same binding contour as the `symbol-macrolet` form, fixing the bug.

Let's move on to another example. Consider a macro named `with-constant-folding` whose meaning is the same as **progn** except that arithmetic subexpressions involving constants are evaluated at expansion time. Thus,

```
(with-constant-folding
   (setq x (+ 2 (* 3 4)))
   (/ (- y (* 24 7))
      (* 60 60)))
```

should expand into

```
(progn (setq x 14)
       (/ (- y 168) 3600))
```

One might have a use for such a macro in some particularly lazy implementation of Common Lisp. For simplicity, we assume that the macro only affects calls to the four functions used in the example.

At first sight, it might appear that the walker cannot be used for this purpose because of the order in which it walks subexpressions. A naive implementation, shown in Figure 3, indeed fails to produce the desired expansion. The example above expands into

```
(progn (setq x (+ 2 12))
       (/ (- y 168) 3600))
```

in which the expression in the `setq` is not fully folded. The problem is, when the walk function is applied to the form `(+ 2 (* 3 4))`, not all of the arguments are constants yet, so nothing gets folded. The walk function never gets to look at the form after the arguments have been walked and so never performs the second level of folding.

We can, however, fix the problem even in the current walker. The trick is to have the walk function do its own recursive walk of the arguments before deciding whether or not to fold.

Of course, the easiest way to do the recursive walk is simply to call `walk-form` again. This solution is in Figure 4. Note that, for efficiency's sake, the walk function in this implementation returns a second value of `t` whenever it has already walked the arguments. Walking the arguments again wouldn't hurt, but it won't help either.

Now that we have a feel for how one might use the walker, let's move on to consider its implementation.

The walker examines the form recursively, keeping track of the appropriate environment and evaluation context. This recursive function is `walk-form-internal`, which `walk-form` calls

```
(defmacro with-constant-folding (&body body &environment env)
  (labels ((each-form (form context env)
             (if (and (eq context ':eval)
                      (consp form)
                      (member (car form) '(+ - * /)))
                 (let* ((args (mapcar #'(lambda (expr)
                                          (walk-form expr env #'each-form))
                                      (cdr form)))
                        (new-form (cons (car form) args)))
                   (values (if (every #'constantp args)
                               (eval new-form)
                               new-form)
                           t))
                 (values form nil))))
    (walk-form `(progn ,@body) env #'each-form)))
```

Figure 4: A better implementation of `with-constant-folding`.

```
(defun walk-form-internal (form context env walk-fn)
  (multiple-value-bind (new-form walk-no-more?)
                       (funcall walk-fn form context env)
    (cond (walk-no-more? new-form)
          ((not (eq form new-form))
           (walk-form-internal new-form context env walk-fn))
          ((atom new-form)
           new-form)
          (t
           ;; Walk the subexpressions of new-form
           ...))))
```

Figure 5: The easy part of `walk-form-internal`.

immediately, establishing `:eval` as the initial context:

```
(defun walk-form (form env walk-fn)
  (walk-form-internal
    form ':eval env walk-fn))
```

The first part of `walk-form-internal` is pretty straightforward and appears in Figure 5. If the walk function somehow did not return the form it was given, the walker starts over again with the new form. This allows the walk function to perform its mapping in convenient cases; when one case reduces to the input of another, the walk function can simply return the new form, knowing that it will get another look at the result.

There are many ways one could imagine for finding the subexpressions of a given form. In particular, the simplest way might be to `case` on the `car` of the form and invoke a special-form-specific walking routine.

The problem is that this leads to a lot of fairly tedious code, since there are many similarities

```
(define-walker-template  function  (nil :call))
(define-walker-template  go        (nil nil))
(define-walker-template  throw     (nil :eval :eval))

(define-walker-template  setq      (nil :repeat (:set :eval)))
(define-walker-template  block     (nil nil :repeat (:eval)))
(define-walker-template  progv     (nil :eval :eval :repeat (:eval)))

(define-walker-template  if        walk-if)
(define-walker-template  tagbody   walk-tagbody)
(define-walker-template  let       walk-let)
```

Figure 6: The walker templates for some of the special forms.

in the syntax of the various Common Lisp expressions. Instead, the walker employs a simple but reasonably powerful language of *walker templates*, patterns that can describe the syntax of most special forms.

A walker template acts as a kind of road map to a particular kind of expression. For each subform, it specifies the evaluation context of that sub-form, or `nil` if the sub-form is not evaluated. For example, Figure 6 shows the template definitions for some of the Common Lisp special forms. The `nil` at the front of several of these represents the fact that the syntactic keywords at the front of each of these forms (i.e., the symbols `function`, `setq`, etc.) are not themselves evaluated.

The syntax of several of the special forms allow certain kinds of sub-forms can be repeated arbitrarily many times. To handle this, walker templates may contain the construction

`:repeat` (*template template* ...)

meaning that the sequence of templates in the parentheses may be repeated zero or more times. Of course, only one `:repeat` is allowed at each level of parenthesis nesting; if there were more, it wouldn't be clear when to stop repeating the first one and go on to the next.

As expressive as this template language is, it is insufficient to describe a few of the Common Lisp special forms. There are three kinds of forms not handled this way:

- The last sub-form of `if` and `return-from` expressions is optional. One could imagine adding a `:optional` construct to the template language to handle these two cases, but it is probably easier to handle them specially. Neither form is very complicated, after all.
- The `tagbody` special form has a very idiosyncratic syntax, definitely easier to handle in a special way.
- While inside the various binding forms, such as `let`, `lambda`, `macrolet`, and `labels`, the walker must arrange to augment the syntactic environment that is passed to the walk function; the environment must reflect the fact that new bindings of one kind or another are in effect.

For these special forms that are undescribable in the template language (there are only ten of them, out of the original 24), the code specifies a form-specific walker function.

We can now understand the rest of `walk-form-internal`, whose complete definition appears in Figure 7.

`Get-walker-template` takes the `car` of a form and returns the appropriate template, or

```
(defun walk-form-internal (form context env walk-fn &aux template fn)
   (multiple-value-bind (new-form walk-no-more?)
                        (funcall walk-fn form context env)
      (cond (walk-no-more? new-form)
            ((not (eq form new-form))
             (walk-form-internal new-form context env walk-fn))
            ((atom new-form)
             new-form)
            ((setq template (get-walker-template (car new-form)))
             (if (symbolp template)
                (funcall template new-form context env walk-fn)
                (walk-template new-form template env walk-fn)))
            (t
             (multiple-value-bind (exp-form expanded?)
                                  (macroexpand-1 new-form env)
                (if expanded?
                   (walk-form-internal exp-form context env walk-fn)
                   (walk-template new-form '(:call :repeat (:eval))
                                  env walk-fn)))))))
```

Figure 7: All of the function `walk-form-internal`.

`nil` if none was defined. If the argument is a `lambda`-expression, it returns the template `(:call :repeat (:eval))`, representing a function call. `Walk-template` is the walker function that interprets the template language; we'll consider it in a moment.

If an expression has no template, the walker checks to see if it is a call to a macro; if so, the process starts over again with the expansion.

Finally, for normal function calls, the walker calls `walk-template` with the form and an appropriate template.

The code for `walk-template` is very nice; it appears in Figure 8. The simple recursive structure of the template language is reflected in the very simple structure of its interpreter. Note how the final clause of the `cond` reconstructs the results of the various sub-walks in a concise way.

The only non-trivial part of the template interpreter is the handling of the `:repeat` construct. We must allow for the possibility that there will be template pieces following a repetition, so the tail of the form that must match those pieces is computed and passed along to `walk-repeat-template`, an auxilliary function of the interpreter. The interpreter checks at each step to see if it has reached this "stop-form"; if so, it abandons the repetition and returns to normal processing.

`Walk-repeat-template` is responsible for iterating through the repetition for as long as necessary. To do so, it keeps track of the template pieces yet to be used in the current iteration (the "repeat-template") as well as the template as a whole. When the repeat-template runs out, a new iteration is begun if appropriate, the repeat-template being reinitialized from the full template.

I think this template mechanism is the most elegant part of the whole walker implementation.

Given the template walker as a model, it is easy to see how to write the special-purpose walker functions for the `if`, `return-from`, and `tagbody` special forms. Of considerably more

```
(defun walk-template (form template env walk-fn)
   (cond ((atom template)
          (ecase template
             ((nil) form)
             ((:eval :set :call)
              (walk-form-internal form template env walk-fn))))
         ((eq (car template) ':repeat)
          (walk-repeat-template form (cdr template) '()
                                (nthcdr (- (length form)
                                           (length (cddr template)))
                                        form)
                                env
                                walk-fn))
         ((atom form)
          (error "While walking template:~%~
                  The template ~S is longer than the form ~S."
                 template form))
         (t
          (cons (walk-template (car form) (car template) env walk-fn)
                (walk-template (cdr form) (cdr template) env walk-fn)))))

(defun walk-repeat-template (form template repeat-template
                             stop-form env walk-fn)
   (cond ((null form)
          (if (and (null repeat-template)
                   (null stop-form))
              '()
              (error "While handling :repeat:~%~
                      The form is shorter than the template.")))
         ((eq form stop-form)
          (if (null repeat-template)
              (walk-template form (cdr template) env walk-fn)
              (error "While handling :repeat:~%~
                      Ran into stop while still in repeat template.")))
         ((null repeat-template)
          (walk-repeat-template form template (car template)
                                stop-form env walk-fn))
         (t
          (cons (walk-repeat-template (car form) template
                                      (car repeat-template) env walk-fn)
                (walk-repeat-template (cdr form) template
                                      (cdr repeat-template) env walk-fn)))))
```

Figure 8: The functions `walk-template` and `walk-repeat-template`.

```
(defvar *walker-data-name* (gensym))

(defun env-walker-data (env)
   (let ((fn (macro-function *walker-data-name* env)))
     (and fn (funcall fn '() env))))

(defun augment-walker-environment (env &rest key-args
                                       &key macro variable)
   (let* ((old-data (env-walker-data env))
          (new-data (append variable (car old-data))))
     (apply #'augment-environment env
        :macro (cons (list *walker-data-name*
                           #'(lambda (form env) new-data))
                     macro)
        key-args)))

(defun variable-binding (env var)
   (member var (env-walker-data env)))
```

Figure 9: Environment-hacking trickery.

interest are the walk functions for forms that affect the environment. I'll take the function handling `let` forms as a representative example; the others are more-or-less straightforward derivatives. First, though, there is the matter of environments.

Code walkers and other programs that manipulate syntactic environments cannot be written portably in the current definition of Common Lisp. This is because no procedures are provided for interrogating environments or for constructing new ones. In fact, all one can do with an environment is pass it to either `macroexpand-1` or `macroexpand`. This situation is quite different in the new standard being developed by the ANSI committee X3J13.[2]

In the new standard, a number of functions have been extended to accept an optional environment argument, `macro-function` being a notable example. In addition, a suite of new functions have been defined for interrogating and constructing environments. For the purposes of this article, only one of these new functions is necessary.

The function `augment-environment` takes an existing environment and a number of keyword arguments and returns a new environment layering the information given in the keyword arguments on top of the given environment. The three keywords we'll need here are `:variable`, `:declare`, and `:macro`; their associated arguments are as follows:

`:variable` A list of symbols to be considered as bound variables in the new environment. All of the bindings are considered to be lexical except those with a corresponding `special` proclamation recorded in the environment or a `special` declaration given with the `:declare` keyword.

`:declare` A list of *decl-specs*, the items that can appear in the `declare` special form.

`:macro` A list of lists, each of which has two elements: a symbol naming a macro and

[2] No ANSI standard has yet been published, so all details described here are subject to change before the draft standard appears.

```
(defun walk-let (form context env walk-fn)
   (multiple-value-bind (remaining-body decl-specs)
                        (parse-declarations (cddr form))
      (let* ((bindings (cadr form))
             (bindings-env (augment-environment env :declare decl-specs))
             (walked-bindings
                (mapcar #'(lambda (binding)
                            (if (symbolp binding)
                               binding
                               (cons (car binding)
                                     (walk-form-internal (cadr binding)
                                                         ':eval
                                                         bindings-env
                                                         walk-fn))))
                        bindings))
             (names (mapcar #'(lambda (binding)
                                (if (symbolp binding)
                                   binding
                                   (car binding)))
                            bindings)))
             (body-env (augment-walker-environment bindings-env
                                                   :declare decl-specs
                                                   :variable names))
             (walked-body (walk-template remaining-body '(:repeat (:eval))
                                         body-env walk-fn))
         `(let ,walked-bindings
             (declare ,@decl-specs)
             ,@walked-body))))
```

Figure 10: The special-purpose walker function for `let`-expressions.

an associated expansion function.

Functions have also been defined allowing programs to discover information about variable, function, and macro bindings, as well as the declarations in force, all in any given environment. Thus, the walker goes to some trouble to ensure that the environments passed to the walk function reflect the correct syntactic context of the current form. Walk functions can then base their actions on the contents of the environment.

Unfortunately, the `variable-binding` function used in the walk function given earlier for `symbol-macrolet` cannot be written in terms of the new facilities in the standard. The walker must therefore implement that functionality itself. All that's needed is a unique value associated with every variable binding. The walker, though some subtle trickery, stores a list of the currently-bound variables in all of the environments it manipulates; the `cons`-cell whose `car` contains a given variable is the unique value for that binding. The code implementing the trickery is in Figure 9.

The idea involves defining a macro in each environment with a "secret" `gensym`-ed name, stored in `*walker-data-name*`. The expansion

```
(defun parse-declarations (body &optional doc-string-allowed?)
   (labels ((scan (body decl-specs doc-string)
              (let ((form (car body)))
                 (cond ((and (stringp form)
                             doc-string-allowed?
                             (null doc-string)
                             (not (null (cdr body))))
                        (scan (cdr body) decl-specs form))
                       ((and (consp form)
                             (eq (car form) 'declare))
                        (scan (cdr body)
                              (append (cdr form) decl-specs)
                              doc-string))
                       (t
                        (values body decl-specs doc-string))))))
     (scan body '() '())))
```

Figure 11: The helper function `parse-declarations`.

function for this secret macro ignores its arguments and returns a piece of "walker data"; in this case, the data is the list of bound-variable names. Given an environment, the function `env-walker-data` returns this piece of data by looking up the macro-function for the secret name and then invoking it on some arguments.

The function `augment-walker-environment` is layered on top of `augment-environment`; it adds any newly-bound variables to the list in the walker data and redefines the secret macro in addition to any other new macros. The new secret macro definition will automatically shadow the old one, just as we desire.

Given this machinery, we can finally understand the special-purpose walker for `let`- expressions; it appears in Figure 10. The function `parse-declarations` takes the body of a `let`- or `lambda`-form and splits it up into declarations, documentation-string (if one is allowed) and the body itself. The code for it is simple[3] and appears in Figure 11. Because of the (somewhat unfortunate) semantics of Common Lisp declarations, the binding-expressions of a `let` must be walked in an environment in which those declarations are in effect; `bindings-env` is that environment. A second new environment, with both declarations and the new variable bindings, is constructed to walk to body. `Walk-let` returns a reconstructed `let`-expression, incorporating the results of all the walking.

As a final point concerning the walker, I want to go back to the issue of the order in which a given expression is walked. As we saw much earlier, there are good uses for both orders (parent before children and vice-versa) but the walker currently provides only one. During our conversations on the walker while I was writing this article, Gregor came up with an interesting change that would not only allow both orders, but an arbitrary mixture of the two.

Under the new idea, the interface of walk functions would change slightly. Along with the form, context, and environment, walk functions would receive a "continuation" argument. The continuation, when passed an expression, would recursively walk all of the immediate subexpressions of that form, returning the reconstructed

[3]The code is made even simpler by the fact that, in the new standard, macros may not expand into declarations.

```
(defmacro with-constant-folding (&body body &environment env)
  (walk-form '(progn ,@body) env
     #'(lambda (form context env continuation)
         (let ((walked-form (funcall continuation form)))
           (if (and (eq context ':eval)
                    (consp walked-form)
                    (member (car walked-form) '(+ - * /))
                    (every #'constantp (cdr walked-form)))
               (eval walked-form)
               walked-form)))))
```

Figure 12: Implementing `with-constant-folding` for a continuation-passing walker.

result. Also, walk functions would no longer return a second value indicating whether or not to continue walking; if more walking should be done, the walk function would have to call the continuation argument.

In this way, walk functions have complete control over the order in which subexpressions are walked; it is completely determined by when, whether, and even how often the continuation argument is invoked. Figure 12 shows how the correct implementation of `with-constant-folding` could be written for such a walker. Note that the recursive calls to `walk-form` are gone; the code more closely resembles the simplicity of the old, naive implementation.

You might find it interesting to try your hand at changing the walker to work in this new way; I think that `walk-form` and `walk-form-internal` are the only functions needing modification. I'm given to understand that this change will probably be working its way into the distributed code sometime in the near future.

That just about covers the most important ideas in the walker. It's an interesting tool that can serve as the basis for a wide variety of other code-analysis applications. The next time you're writing a hairy macro, you may very well find yourself wishing you had the walker to help out; fortunately, it's freely available.

If you'd like to get a copy of the PCL code walker, or even the entire PCL system, send either electronic mail to "CommonLoops-Coordinator.pa@Xerox.Com" or normal mail to

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

They can send you information on the options available to you for receiving the code.

Next issue, the `(algorithms)` department will cover one of the largest clients of the code walker: a clever implementation by Bill van Melle of an elegant and powerful iteration facility. If you've got an idea for a article in this department for issues after that, please send them along. ()