Implementation of an Iteration Macro

Bill van Melle
Xerox PARC
3333 Coyote Hill Rd.
Palo Alto, CA 94304
vanMelle@Xerox.Com

April 19, 1990

This article describes the implementation of a set of iteration macros. These macros are found in the Portable Common Loops (PCL) implementation of the Common Lisp Object System, though they are not dependent on PCL in any way. The macros were designed to be more functional than the small set of iteration macros found in Common Lisp (e.g., `do`), while still having a "Lisp-like" syntax that is easy to understand and extend. Pavel Curtis is responsible for the essential insights about program transformation that make feasible an efficient implementation of this conceptually simple iteration form. The implementation makes extensive use of the PCL code walker, which was described in the previous issue of *Lisp Pointers*.

There are two major macros: `iterate`, which defines an iteration, and `gathering`, which defines a context in which to accumulate one or more results. For example,

```
(gathering ((result (collecting)))
  (iterate
        ((e (list-elements mylist))
         (i (interval :from 1)))
    (when (interesting-p e)
      (gather e result)
      (format t
              "~D: collected ~S~%"
              i e))))
```

maps over the elements of a list, collecting the interesting ones into a new list and reporting the index of each one collected.

The first subform to `iterate` is a list of clauses, looking much like `let` bindings, each of which specifies an *iteration variable* and a form that is evaluated to produce a *generator*. A generator is a function that produces the next in a sequence of values each time it is called; it indicates it is exhausted by calling its first and only argument, a closure that terminates the iteration. The `iterate` form binds the iteration variables, and executes the remainder of the `iterate` form in a loop, at the beginning of which it sets each variable to the output of the corresponding generator.

The first subform to `gathering` is also a list of clauses, but in this case, the variable in each pair is thought of as an abstract accumulation site, and the second element of the pair is a *gatherer* expression. This expression returns two values: the first is a function that when called with an argument incorporates the argument into its accumulating value, the second is a function that when called returns the accumulated result. `gather` is a function that applies the gatherer associated with its second argument to its first argument. The body of the `gathering` form can thus call `gather` in as many places as it likes, as long as they are lexically apparent. This is in contrast to such alternatives as ZetaLisp `loop`, which permits an accumulation in only one place, the top-level of the iteration construct. In fact, `gathering` need not appear within an iteration at all. The `gathering` form returns (as multiple values) the accumulated re-

```
(multiple-value-bind (result %finisher)
    (collecting)
  (flet ((gather (value accumulator) (funcall accumulator value)))
    (block %itloop
      (let* ((%lstgen (list-elements mylist))
             (%intgen (interval :from 1))
             e i)
        (loop (setq e (funcall %lstgen #'(lambda () (return-from %itloop))))
              (setq i (funcall %intgen #'(lambda () (return-from %itloop))))
              (when (interesting-p e)
                (gather e result)
                (format t "~D: collected ~S~%" i e)))))
    (values (funcall %finisher))))
```

Figure 1: The formal expansion of an `iterate/gathering` form.

```
(defmacro list-elements (list &key (by '#'cdr))
       `(let ((tail ,list))
             #'(lambda (finish)
                   (prog1 (if (endp tail)
                              (funcall finish)
                              (first tail))
                          (setq tail (funcall ,by tail))))))
```

Figure 2: The definition of the generator `list-elements`.

sults of all of the gatherers.

Thus, the formal meaning of the example above is shown in Figure 1 (the "%" identifiers in this and other examples should be thought of as gensyms that have been given suggestive names for readability). And that is all you need to know to understand what an `iterate` or `gathering` form does, and nearly all you need to know to write new generators and gatherers.

A generator expression can be any Lisp form that returns a funcallable object, but is typically written as a macro that produces a closure. For example, the definition of `list-elements` is shown in Figure 2.

The macro binds some state (the tail of the list being iterated over), and returns a function closed over that state. When the function is called, it checks whether the list is exhausted. If so, it calls its argument (the exit function); otherwise, it returns the head of the list and sets the state to the rest of the list. The macro takes an optional `:by` keyword argument that lets you walk down the list by some other accessor than `cdr`. The `iterate` module also supplies definitions for `interval`, `list-tails`, and `elements` (of a generic sequence); it is clearly easy for a user to define additional ones.

Similarly, generators are typically written as macros that produce two closures as values. The definition of `collecting` is shown in Figure 3. Again, the macro binds some state and returns two functions closed over that state. Additional gatherers already defined are `summing`, `maximizing`, and `minimizing`.

If you expand these two macros (and the obvious one for `interval` as well) in the "formal meaning" example given above, you get some obviously inefficient code. However, the actual

```
(defmacro collecting ()
 `(let (head tail)
    (values #'(lambda (value)
                (if (null head)
                    (setq head (setq tail (list value)))
                   (setq tail (cdr (rplacd tail (list value))))))
            #'(lambda () head))))
```

Figure 3: The definition of the gatherer `collecting`.

```
(let (%lsthead %lsttail)
  (block %itloop
    (let* ((%lststate mylist)
           (%intstate 1)
           e i)
      (loop (setq e (prog1 (if (endp %lststate)
                               (return-from %itloop)
                              (first %lststate))
                           (setq %lststate (cdr %lststate))))
            (setq i (prog1 %intstate (setq %intstate (+ %intstate 1))))
            (when (interesting-p e)
                (let ((value e))
                   (if (null %lsthead)
                       (setq %lsthead (setq %lsttail (list value)))
                      (setq %lsttail (cdr (rplacd %lsttail (list value))))))
                (format t "~d: collected ~s~%" i e)))))
   %lsthead)
```

Figure 4: The actual expansion of the `iterate/gathering` example.

implementation of `iterate`, which uses program transformation techniques, produces code more like that shown in Figure 4. This is about as good as the code you might produce by hand, or that might be produced by other iteration facilities, such as `loop`. And that is what gives us anything interesting to write in this article.

To understand the program transformations involved, let us work through a subset of the previous example:

```
(iterate ((e (list-elements l)))
   body)
```

whose naive expansion is shown in Figure 5.

The first transformation is something we call *let eversion*. Abstractly, it is permissible to replace

```
(let ((var (let (bindings) value)))
   body)
```

with

```
(let (bindings)
   (let ((var value))
      body))
```

as long as none of the variables bound by *bindings* is special, and none is used in *body*. Given the former, it is easy to insure the latter: the variables bound by *bindings* are only known within the lexical scope of the internal let, so

```
(block %itloop
   (let* ((%lstgen (let ((tail l))
                     #'(lambda (finish)
                         (prog1 (if (endp tail)
                                    (funcall finish)
                                    (first tail))
                                (setq tail (funcall #'cdr tail))))))
          e)
     (loop (setq e (funcall %lstgen #'(lambda () (return-from %itloop))))
           body)))
```

Figure 5: The naive expansion of a one-generator `iterate` example.

```
(block %itloop
   (let* ((%lsttail l)
          (%lstgen #'(lambda (finish)
                       (prog1 (if (endp %lsttail)
                                  (funcall finish)
                                  (first %lsttail))
                              (setq %lsttail (funcall #'cdr %lsttail)))))
          e)
     (loop (setq e (funcall %lstgen #'(lambda () (return-from %itloop))))
           body)))
```

Figure 6: Example after applying let eversion.

we can just rename them all to gensyms, a compiler transformation known as *alpha conversion.* This transformation is an easy task for the code walker, as we will describe below. There is one more tweak to let eversion that we make solely for cosmetic reasons: having renamed all the variables, it is clear that the transformation is valid even if `let` becomes `let*` (because none of the initialization forms in the `let` can refer to any of the renamed variables), so we can collapse all the bindings into one `let*`.

Applying let eversion to our example yields the code shown in Figure 6.

The next transformation is a fairly standard compiler optimization called *beta reduction*—substituting the value of a bound variable for every reference to that variable. This transformation is permitted if (1) the variable is not side-effected between the binding and its use, (2) the value expression has no side effects, and (3) the value expression references no variables whose bindings appear between the original binding and the reference.

In the case of an `iterate` form, we want to substitute the generator body `#'(lambda ...)` for its occurrence in the `funcall` in the loop body. Condition (1) is obviously met, because the bound variable is an invisible gensym with a single read-only reference (the `funcall`). Condition (2) is met, because the special form `function` has no side effects.

Condition (3) is met if the generator function makes no free reference to any iteration variables that follow it in the main `let*` (those being the only visible variables between the binding and the `funcall` at the top of the loop). We certainly expect this to be the case normally—most generators have no free references at all

```
(block %itloop
   (let* ((%lsttail l)
          e)
      (loop (setq e
              (funcall #'(lambda (finish)
                           (prog1 (if (endp %lsttail)
                                      (funcall finish)
                                     (first %lsttail))
                                  (setq %lsttail (funcall #'cdr %lsttail))))
                       #'(lambda () (return-from %itloop))))
            body)))
```

Figure 7: Example after applying beta reduction.

```
(block %itloop
   (let* ((%lsttail l)
          e)
      (loop (setq e (prog1 (if (endp %lsttail)
                               (return-from %itloop)
                              (first %lsttail))
                           (setq %lsttail (funcall #'cdr %lsttail))))
            body)))
```

Figure 8: Example after funcall and beta reductions.

outside of the variables they're closed over—but it is possible to construct perverse forms that violate this condition, so our `iterate` implementation must check for it. Applying beta reduction to our example yields the code shown in Figure 7.

The final transformation, a trivial one performed by any Lisp compiler worthy of the name, we'll call *funcall reduction*: transform `(funcall #'`*fn* . *args*`)` into the simpler form `(`*fn* . *args*`)`. We need to perform this transformation explicitly, however, in order to apply beta reduction a second time, substituting for the `finish` argument in the generator. Finally, we apply funcall reduction a second time to pretty up the use of this argument. These transformations yield the final expansion shown in Figure 8.

Of course, not all `iterate` forms are optimizable in this way, since we permit a generator to be any Lisp form. It can be a function call, which we can't touch in any way, or it can be an arbitrarily complex macro that doesn't match the pattern `(let (...) #'(lambda ...))`. For these generators, the expansion of `iterate` is just the formal meaning we outlined at the beginning of this article. However, the vast majority of useful generators can be written in the optimizable form.

A similar application of these same transformations allows us to optimize the `gathering` macro. First, we apply alpha conversion to each gatherer's body; this allows us to apply let eversion to lift its state bindings past the `multiple-value-bind` at the top of the `gathering`. If we're lucky, we now have something of the form

```
(multiple-value-bind (gather finish)
      (values ...)
   ...)
```

```
(defun expand-into-let (form env)
   (prog ((expansion form)
          expandedp binding-type let-bindings let-body)
     expand
         (multiple-value-setq (expansion expandedp)
                (macroexpand-1 expansion env))
         (cond
           ((not (consp expansion)) ; Not an expression
            )
           ((symbolp (setq binding-type (first expansion)))
            (case binding-type
                ((let let*)
                   (setq let-bindings (second expansion))
                   (setq let-body (cddr expansion))
                   (go handle-let))))
           ((and (consp binding-type)
                 (eq (first binding-type) 'lambda)
                 (null (intersection lambda-list-keywords
                                     (setq let-bindings (second binding-type))))
                 (eql (length let-bindings)
                      (length (rest expansion))))
            ;; Treat ((lambda (...) ...) . args) as a let
            (setq let-body (cddr binding-type))
            (setq let-bindings (mapcar #'list let-bindings (rest expansion)))
            (setq binding-type 'let)
            (go handle-let)))
         ;; Fall thru if not a let or lambda
         (if expandedp              ; Try expanding again
             (go expand)
            (return :abort))
    handle-let
         (return
           (if (find-if #'variable-globally-special-p
                        (variables-from-let let-bindings))
               ;; It binds special vars
               :abort
              (values (if (and (consp let-body)
                               (null (cdr let-body)))
                          (first let-body)
                         :abort)    ; More than one expression, or malformed
                      binding-type let-bindings)))))
```

Figure 9: The function expand-into-let.

which we can turn into the obvious `let`. We then apply beta reduction to the `flet` binding of `gather`, and then to each gatherer and finisher binding. Satisfying the preconditions of beta reduction is a little more demanding this time, however, as we have no control over where `gather` appears in the body. So we must walk the entire body of the `gathering` form, substituting for `gather` calls that satisfy the preconditions (no conflicting bindings between the top level binding and the `gather` call), and making sure that there is no code that sets a gatherer's site variable. For simplicity, we also abort the optimization if we discover a reference to a gatherer's site outside of a `gather` call, or a use of `#'gather`. And, of course, we can't optimize any gatherer expression that isn't a macro call, or whose expansion is more complex than the expected form

```
(let ...
   (values #'(lambda ...)
           #'(lambda ...)))
```

Let us now look at the implementation of `iterate` and `gather`. We'll begin with a simplified version of `iterate`. Although we described the optimization of `iterate` above as a sequence of program transformations, the implementation proceeds more in parallel. We know what we are aiming for: a form that binds some variables around a loop that updates the variables and performs the iteration body. Each `iterate` clause contributes an iteration variable and some number of bindings for its generator's state, as well as a form to evaluate at the top of the loop to set the iteration variable to its new value.

The first step in processing a clause is to macro-expand the generator to see if it is of the desired form. This task is handled by the function `expand-into-let` (Figure 9). It takes a form and a macro-expansion environment, and attempts to expand the form into an expression that looks like (`let`[`*`] *bindings body*). When it finds the `let`, it verifies that none of the bindings is of a special variable, since that would preclude let eversion. If successful, it returns three values: the body, the symbol `let` or `let*`, and the list of bindings. If unsucessful, it returns `:abort`. Since calls to simple `lambda` forms (those without lambda-list keywords) are equivalent to `let`, and some Lisp implementations macro-expand `let` into `lambda`, `expand-into-let` also treats such a call as the equivalent `let`.

If the expansion into `let` succeeded, we now have a set of bindings (variables and initialization expressions) and a body. We check that the body is of the desired form, viz., `#'(lambda (`*finisharg*`) ...)`. Next, we prepare for alpha conversion by choosing some gensyms for the variables. If the binding form was `let*`, we must also substitute for variable references in the second and subsequent initialization expressions. This task is handled by the function `rename-let-bindings` (Figure 10), and is the first use we make of the code walker. The function returns the renamed bindings, along with an association list mapping old to new names.

The procedure `rename-variables` takes a form, an environment, and an association list mapping old to new variable names, and substitutes the new variables for all references to the old variables that refer to the binding visible in the environment. This is just like the Common Lisp function `sublis`, except that we must be smart and substitute only for variable references, not function calls or literals, and only in contexts where a variable isn't shadowed by another binding form. So we use the code walker to traverse the form. Every time we are offered a symbol that appears in the a-list we return the corresponding new variable, but only if the symbol refers to the binding we're interested in. The function `variable-same-p` tests whether the bindings of a variable visible in each of two environments are the same. It uses `variable-lexical-p`, a function in the code walker that tests whether a variable is lexical in an environment, and if so returns a unique object representing the instance of the binding visible in the environment.

Finally, we perform two transformations on

```
(defun rename-let-bindings (let-bindings binding-type env)
  (let (renamed-vars)
    (values (mapcar #'(lambda (binding)
              (let ((newvar (gensym))
                    (valueform (cond
                                 ((not (consp binding)) ; No initial value
                                  nil)
                                 ((or (eq binding-type 'let)
                                      (null renamed-vars))
                                  (second binding))
                                 (t ; For let*, do all previous renamings
                                    ; in the initialization form.
                                  (rename-variables (second binding)
                                                    renamed-vars env)))))
                (push (cons (if (consp binding) (first binding) binding)
                            newvar)
                      renamed-vars)
                (list newvar valueform)))
            let-bindings)
            renamed-vars)))
(defun rename-variables (form alist env)
   (walk-form form env
              #'(lambda (form context subenv)
                  (let (pair)
                    (if (and (symbolp form)
                             (setq pair (assoc form alist))
                             (variable-same-p form subenv env))
                        (cdr pair)
                        form)))))
(defun variable-same-p (var env1 env2)
   (eq (variable-lexical-p var env1)
       (variable-lexical-p var env2)))
```

Figure 10: Renaming the bindings of a `let` or `let*`.

the generator body in one pass of the code walker (Figure 11): alpha conversion on the body, using the association list returned from `rename-let-bindings`, and beta reduction and funcall reduction for the generator's finish argument.

At the same time, we verify that the requirement for beta reduction is met (no reference to subsequent iteration variables), and that there is no reference to the generator's finish argument other than funcalls to it.

At this point, we have finished the transformation of the generator body into code that we can evaluate inside the loop to update the iteration variable.

If any of the transformation requirements was not met, however, we instead produce an update expression that reflects the formal semantics of `iterate`: a call to the generator, with the generator being computed in the list of bindings

```
(defun iterate-transform-body (let-body iterate-env renamed-vars
                               finish-arg finish-form bound-vars)
  (walk-form let-body iterate-env
    #'(lambda (form context env)
        (cond
          ((symbolp form)
           (let (renaming)
             (cond
               ((and (eq form finish-arg)
                     (variable-same-p form env iterate-env))
                ;; An occurrence of the finish arg outside of funcall
                (return-from iterate-transform-body :abort))
               ((and (setq renaming (assoc form renamed-vars))
                     (variable-same-p form env iterate-env))
                ;; Reference to a variable to rename
                (cdr renaming))
               ((and (member form bound-vars)
                     (variable-same-p form env iterate-env))
                ;; Reference to a var bound later in same iterate
                (return-from iterate-transform-body :abort))
               (t form))))
          ((and (consp form)
                (eq (first form) 'funcall)
                (eq (second form) finish-arg)
                (variable-same-p (second form) env iterate-env))
           ;; (funcall finish-arg) ⇒ finish-form
           finish-form)
          (t form)))))
```

Figure 11: The function `iterate-transform-body`.

produced for this clause.

After we repeat this process for all the iteration clauses, we assemble the accumulated bindings and update forms into the complete expansion, and we're done.

The complete implementation of the `iterate` macro is shown in Figure 12.

The variables `bindings` and `update-forms` are accumulators for the bindings of generator state variables and iteration variables and for the update forms, respectively. In the variable `bound-vars`, we maintain a list of all the iteration variables from the clauses we have not yet processed, so that the beta reduction step (`iterate-transform-body`) can check that no generator uses them conflictingly. The procedure `function-lambda-p` is a simple pattern-matching function that tests that its argument is of the form `#'(lambda ...)` and returns the `lambda` subexpression on success.

The actual implementation of `iterate` is a bit more complicated in several ways omitted from the discussion above for clarity:

- It does a more thorough job of checking the syntax of the `iterate` form, e.g., that each iteration clause is a list of exactly two elements, and that there are no duplications among the iteration variables.

```
(defmacro iterate (clauses &body body &environment env)
 (let* ((block-name (gensym))
        (finish-form '(return-from ,block-name))
        (bound-vars (mapcar #'car clauses))
        update-forms bindings)
   (dolist (clause clauses)
     (multiple-value-bind (let-body binding-type let-bindings)
           (expand-into-let (second clause) env)
       (let ((iv (first clause))
             gen-arg renamed-vars)
         (cond
           ((eq let-body :abort)       ; We've already failed
            )
           ((null (setq let-body (function-lambda-p let-body 1)))
            (setq let-body :abort))    ; Not of the expected form
           (t ;; Generator body is of the form #'(lambda (finisharg) ...).
              ;; let-body is now the lambda form.
              (setq gen-arg (first (second let-body)))
              (setq let-body '(progn ,@(cddr let-body)))
              (when let-bindings       ; Begin renaming for let eversion.
                  (multiple-value-setq (let-bindings renamed-vars)
                      (rename-let-bindings let-bindings binding-type env)))
              ;; Alpha-convert generator body and expand (funcall finisharg)
              (setq let-body (iterate-transform-body let-body env
                                renamed-vars gen-arg finish-form bound-vars))))
         ;; let-body is now the update form for the iteration variable.
         (when (eq let-body :abort)
            ;; Optimization failed, use the formal semantics
            (let ((gvar (gensym))
                  (generator (second clause)))
              (setq let-bindings (list (list gvar generator)))
              (setq let-body
                    '(funcall ,gvar #'(lambda nil ,finish-form)))))
         (pop bound-vars)        ; Needn't watch out for this variable any more.
         (push '(setq ,iv ,let-body) update-forms)
         (setq bindings (nconc bindings let-bindings (list iv))))))
   ;; All clauses now processed, so emit the code.
   '(block ,block-name
      (let* ,bindings
        (loop ,@(nreverse update-forms)
              ,@body)))))
```

Figure 12: Top level implementation of the `iterate` macro.

- It issues optional warnings when it is unable to fully optimize a form, e.g., when a generator fails to expand into the expected form, or doesn't expand at all. A user-settable flag controls whether warnings are issued for all suboptimal forms, for only those that could be influenced by changing "user" code, or never. Issuing helpful warnings requires passing around the original clause to some of the functions that don't otherwise need access to it in the simplified implementation described above.

- It handles declarations. The complete syntax for `iterate` permits declarations to follow the iteration clauses. These declarations must be moved out of the iteration body and placed immediately after the `let*` bindings in the `iterate` expansion, so that they are in the correct place to describe the iteration variable bindings.

  Declarations can also appear in the expansion of a generator, both in its top-level `let` and in the closure body. These are slightly trickier to handle. Those that refer to variables in the `let` have to migrate as part of the let eversion transformation to the outer `let*` in the `iterate` expansion (and have their variables renamed); others, such as `inline` declarations, must remain with the generator body, which then becomes

  ```
  `(locally
      ,@declarations
      ,@let-body)
  ```

  in `optimize-iterate-form`.

- It allows a generator's `let` to have a larger body. In the simple implementation, we insisted that a generator expand into a `let` with a single expression in its body, `#'(lambda (`*finisharg*`) ...)`.

  Let eversion still works if there are more forms in the `let`, because we know that Common Lisp provides no way to exit from the straight-line computation in a `let` body without also exiting some parent form.

  Thus, we can generalize let eversion to transform

  ```
  (let ((var (let (bindings)
               computation
               value)))
    body)
  ```

  into

  ```
  (let (bindings)
    computation
    (let ((var value))
      body))
  ```

  In order to collapse this into a single `let*`, we simply defer the *computation* until the next variable's initialization form, enclosing it in a suitable `progn`.

- It permits a generator to return multiple values, bound to multiple iteration variables. For example,

  ```
  (iterate (((indicator value)
             (plist-elements
               (symbol-plist s))))
    (format t "~S: ~S~%"))
  ```

  prints out a nice display of the property list of a symbol. The most significant change this makes to the implementation is that the iteration variables are set via `multiple-value-setq` to the output of the generator.

  The special case where, after all the transformations, the body of the generator is a call on `values` is explicitly optimized to multiple `setq`'s for the benefit of the few compilers that wouldn't do that automatically.

There is one further transformation one could make to the `iterate` expansion that would make it essentially indistinguishable from the

```
(block %itloop
   (let* ((%lsttail l)
          e)
      (loop (setq e (if (endp %lsttail)
                        (return-from %itloop)
                      (first %lsttail)))
            body
            (setq %lsttail (funcall #'cdr %lsttail)))))
```

Figure 13: A possible further optimization of the `iterate` example.

code you would write by hand: explode the `progl` of the typical generator body into its first subexpression now and the rest at the end of the loop. This would turn our earlier example (Figure 8) into the code shown in Figure 13. For correctness, this transformation requires that the tail of the `progl` have no side effects other than to the generator state (in this case, `%lsttail`), and in turn that it be unaffected by anything in the body of the `iterate`. Although virtually any reasonable generator satisfies both conditions, it is difficult to verify this mechanically. Thus, we have chosen not to attempt this additional optimization, leaving it instead to a "sufficiently clever compiler," which would be in a better position to verify these conditions.

As for the implementation of `gathering`, the processing of the clauses is similar to that of `iterate`, and uses several of the subfunctions described above. However, the beta reduction step is a more complex task, as we must walk the entire body of the `gathering` form in order to substitute for calls on `gather`. This walk has a few more things to worry about:

- Given the possibility that there are nested `gathering` forms, we must be prepared to encounter a `gather` referring to an accumulation site in a parent form, rather than in the one we are currently expanding. We could ignore such occurrences, relying on the macro expansion of the parent to handle it, but then we'd be unable to issue warnings at compile time about an unrecognized second argument to `gather`. So instead, we maintain a special variable that records all the active accumulation sites. The macro expander for `gathering` rebinds this variable with its own sites consed onto the front.

- When analyzing each gatherer during the alpha conversion step, we must additionally note any free variable references. Later, when walking the `gathering` body, we check each occurrence of `gather` to make sure that no variable referenced freely by the corresponding gatherer has been rebound in the meantime. This test is again made by `variable-same-p`.

- We check that there is no reference to an accumulation site variable outside of a call to `gather`.

The interested reader is invited to peruse the PCL sources for the complete implementations of both `iterate` and `gathering`. To get a copy, either send electronic mail to "CommonLoops-Coordinator.pa@Xerox.Com" or send normal mail to

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

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