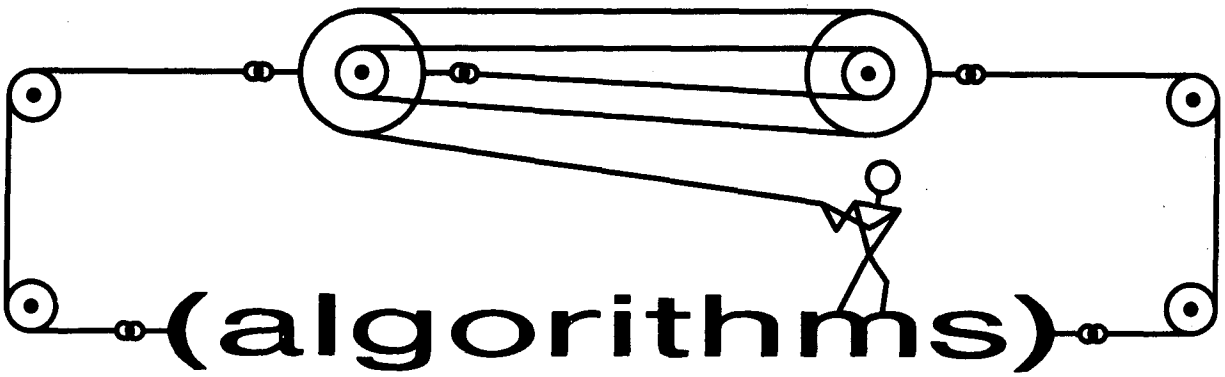

Pavel Curtis, editor

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

Pavel.pa@Xerox.Com

Welcome to the first appearance of this new Lisp Pointers department on Algorithms. It has been noted by some editors that most Lisp hackers would rather write code than articles and I suspect that their reading preferences might run the same way. This department is an attempt to cater to those preferences. The articles you'll see here will tend to fit into one or more of three broad categories:

- Annotated implementations of interesting and relevant algorithms that 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.

I will be continually looking for ideas for appropriate articles for the department. If you've got a nice hack you're proud of, or a particularly elegant piece of code (you know, the kind that you call in one of your fellow hackers to see) and you'd like to see it written up in the Algorithms department, please send it along. What you give me doesn't have to be polished or even contain any prose; if I agree that it's appropriate for the column, I'll work with you to put together an article around it. My electronic and physical mail addresses appear above, so feel free!

Our first topic for the department is *syntactic extension*, a mechanism known in most dialects of Lisp as *macros*. A lot of discussion on this topic has been taking place recently in both the Scheme and Common Lisp communities and one of the more comprehensive written pieces of that discussion is the Indiana University Ph.D. thesis of Eugene Kohlbecker, *Syntactic Extensions in the Programming Language Lisp*, submitted in 1986.

Kohlbecker's thesis covers a lot of territory that I won't bring up here, but one of the most interesting pieces is a facility that he describes for the specification of syntactic extensions. He introduces a special form called `extend-syntax` that would fit into Scheme the way `defmacro` fits into Common Lisp; that is, it's the form you use to define a new macro in the system.

In this column, I'm going to ignore a number of the interesting features of `extend-syntax` to concentrate on the piece that I found most elegant, the little language in which both macro-call patterns and their corresponding expansions are written. A complete description of the facility, along with a formal semantics, appears in Kohlbecker's thesis. There is also a very good description and a large number of sophisticated examples in R. Kent Dybvig's fine book, *The Scheme Programming Language*, available from Prentice-Hall publishers. We'll first look at a few examples of how the facility is used and then move on to the algorithmic aspect, talking about an implementation of `extend-syntax` also written by Dybvig.

The canonical example of an `extend-syntax` form is this definition of the `let` construct from Scheme:

```
(extend-syntax (let)
   ( (let ((var val)
           ...)
      expr1
      expr2
      ...)
     ((lambda (var ...)
        expr1
        expr2
        ...)
      val ...)))
```

The symbol "..." is not, technically speaking, legal in either Scheme or Common Lisp, but we'll pretend that it is for the remainder of this column.

An `extend-syntax` form starts with a list of *keywords*, special symbols that can appear in a use of the new macro and that match only themselves in the patterns found in the rest of the form. The first element of this list is always the name of the new macro.

After the list of keywords comes a series of *transcription clauses*, each of which is a list of two elements, a *pattern* and an *expansion template*. To expand a use of a macro, the macro processor compares it to the pattern of each transcription clause. The expansion template of the first matching pattern is used to construct the transcription of the call. In the definition of `let` above, there is only one clause; we'll see examples with more than one in a moment.

Patterns and expansion templates are mostly made up of list structure and symbols. We refer to those symbols that appear in patterns and are not keywords as *pattern variables*, since they are, in a sense, the parts of the pattern that are allowed to vary. In the example above, the symbols `var`, `val`, `expr1`, and `expr2` are pattern variables. The symbols in expansion templates also fall into two categories; they are either occurences of pattern variables, or they are *expansion constants*. The only expansion constant in the example above is the symbol `lambda`.

The most interesting and elegant thing about the patterns and expansion templates in `extend-syntax` is the use of the *ellipsis* (the "..." symbol). Whenever a piece of a pattern is directly followed by an ellipsis, the pattern matcher will be looking for zero or more occurrences of that sub-pattern. The definition of `let` says "`expr1 expr2 ...`" instead of simply "`expr ...`" so as to require, as Scheme does, that at least one expression always appear. We'll refer to sub-patterns that are directly followed by ellipses as *iterated patterns* and we'll say that the pattern variables appearing in iterated patterns are *iterated pattern variables*.

To avoid extra complexity and the need for any form of backtracking in the implementation, ellipses in `extend-syntax` patterns must appear at the very end of lists, as they do in the definition of `let`. Ellipses in expansion templates, though, are not constrained in this way.

In the expansion template, the ellipsis is again used to denote repetition, in this case of some sub-template. Somewhere within such an iterated sub-template, there must appear one or more iterated pattern variables; it is the appearance of such variables that "drives" the template repetition. The repetitions of the expansion will use successive values of those variables, in the order that the pattern matcher found them.

Let's look at a couple more examples. The following definition of the `cond` macro from Scheme illustrates the use of both more than one keyword and more than one transcription clause. For those who are less familiar with Scheme, `begin` is equivalent to `progn` in Common Lisp.

```
(extend-syntax (cond else)
   ( (cond (else expr1 expr2 ...))      ; first pattern
     (begin                             ; first expansion template
        expr1
        expr2
        ...))

   ( (cond (expr)                       ; second pattern
           more-clauses ...)
     (or expr                           ; second expansion template
         (cond more-clauses ...)))

   ( (cond (predicate body ...)         ; third pattern
           more-clauses ...)
     (if predicate                      ; third expansion template
        (begin
           body ...)
        (cond more-clauses ...))))
```

Finally, for a slightly more complex use of the facility, here's a definition of the `etypecase` macro from Common Lisp:

```
(extend-syntax (etypecase)
   ( (etypecase expr
        (type body1 body2 ...) ...)

     (let ((v expr))
        (cond ((typep v 'type)
                  body1 body2 ...)
              ...
              (t (error "~S is not of type ~S"
                        v
                        '(or type ...)))))))
```

The variable `v` in the expansion template should, of course, really be a `gensym`'d symbol, but the `extend-syntax` mechanism for accomplishing this is outside the scope of this article. See the description in Dybvig's book for more information.

With this intuitive understanding of the facility, let's begin considering Dybvig's very clean and elegant implementation. A complete listing of a simplified version of the code appears at the end of the column, along with some explanation of Scheme for Common Lisp programmers.

In my editing of Dybvig's code, I've dodged the question of how `extend-syntax` is registered with the Scheme system. Since the Scheme specification doesn't yet cover macros, the details vary from implementation to implementation. It is assumed in the code that, one way or another, the function `make-`

`syntax` will be called with the list of keywords and a list of the transcription clauses from an `extend-syntax` form. Its job is to translate that specification into the `lambda`-expression for a function mapping macro-call forms to their expansions. Thus, for the definition of `let` given above, we assume that the following call takes place:

```
(make-syntax '(let) '( ((let ((var val) ...) expr1 expr2 ...)
                        ((lambda (var ...) expr1 expr2 ...) val ...))))
```

The value returned from this call would be the following `lambda`-expression:

```
(lambda (g001)
   (cond ((syntax-match? '(let)
                         '(let ((var val) ...)
                            expr1
                            expr2
                            ...)
                          g001)

          `((lambda (,@(map (lambda (g002) `,(car g002))
                            (car (cdr g001))))
              ,(car (cdr (cdr g001)))
              ,@(map (lambda (g003) `,g003)
                     (cdr (cdr (cdr g001)))))
            ,@(map (lambda (g002) `,(car (cdr g002)))
                   (car (cdr g001)))))
         (else
            (error 'let g001 "invalid syntax"))))
```

Not a pretty sight, eh? It does work, though. The variable `g001` is bound to a given `let` expression and the function `syntax-match?` performs the pattern-matching. The first call to `map` in the expansion is constructing the argument list for the resulting `lambda`; it maps over the `cadr` of `g001` (the list of bindings in the `let`), returning a list of the `cars` (note that `` `,*expression* `` is equivalent to simply *expression*). The expression `(car (cdr (cdr g001)))` is extracting the value of the pattern variable `expr1`, and the second call to `map` is splicing in the expressions covered by "`expr2 ...`". The final call to `map` again maps over the binding list, this time extracting the `cadrs`.

This code is not terribly readable and, in fact, I've done Mr. Dybvig a bit of a disservice by presenting only this simplified version of his implementation. The code generated by the real implementation is the following:

```
(lambda (g001)
   (cond ((syntax-match? '(let)
                         '(let ((var val) ...)
                            expr1
                            expr2
                            ...)
                          g001)
          `((lambda ,(map car (cadr g001))
              ,(caddr g001)
              ,@(cdddr g001))
            ,@(map cadr (cadr g001))))
         (else
            (error 'let g001 "invalid syntax"))))
```

I leave it as an exercise to you readers to find the small number of tweaks to the code that make it generate this much prettier output.

Now that we know where we're going to get, let's look at how the implementation, which is much prettier than its output, gets us there. The expansion function is always of the form

```
(lambda (form-var)
   (cond --one cond-clause per transcription clause--
         (else
            --signal an error--)))
```

and is put together by the function `make-syntax`. That function calls `make-clause` to construct each of the `cond`-clauses, which have the form

```
((syntax-match? 'keywords 'pattern form-var)
   --quasiquote-expression based on the pattern and the expansion template--)
```

Thus, the pattern in each transcription clause is used for two purposes: first, to determine which `cond`-clause to select and second, to label the parts of a macro-call form that will appear in the expansion. The function `syntax-match?` is a reasonably straightforward recursive procedure that walks along the pattern and the macro-call form in parallel, checking that they have the same tree-structure and that the macro-call contains the keywords at the places indicated in the pattern. The only interesting case occurs when the matcher finds an iterated sub-pattern; in such a case, the sub-pattern is recursively matched against each element in the remaining tail of the macro-call form.

A more subtle problem concerns the translation of a pattern and expansion template into a `quasiquote` expression that does the right thing. (See the notes on reading Scheme programs at the end of this column for the meaning of `quasiquote`.) The implementation does this in two phases: parsing the pattern and generating the code.

The idea of the parser (implemented by the function `parse`) is to summarize the the semantic content of all the pattern variables in some form that's convenient for code generation. In essence, the generator needs to know how to extract the appropriate piece of the macro-call form given only the name of one of the pattern variables. If no ellipses were present, the problem is simply one of associating the names with expressions that evaluate to the appropriate piece of the form.

Suppose that the pattern was as follows:

```
(foo a (b c) (d e) ...)
```

and assume that *form-var* is a variable bound to the macro-call form. The parser can associate the names `a`, `b` and `c` with the following forms respectively:

```
(car (cdr form-var))

(car (car (cdr (cdr form-var))))

(car (cdr (car (cdr (cdr form-var)))))
```

(Isn't it fortunate that we, the users of `extend-syntax`, can write in terms of patterns and templates rather than such expressions?) The representation of the iterated pattern variables `d` and `e` is accomplished by a little trick: the implementation makes up a new pseudo-pattern-variable that represents all of the macro-call form that's matched by the iterated sub-pattern. Thus, in this case, it might make up a variable `g004` and associate it with the expression

```
(cdr (cdr (cdr form-var)))
```

Such variables will be used by code generation as iteration variables in loops. The variable g004 will be bound in the generated code to each of the elements of the list that's the result of the above expression. As will be seen later, while such iteration variables are associated by the parser with expressions yielding lists of values, the code generator will end up binding them to the individual values in those lists.

Given this structure to work from, the pattern variables d and e can now be associated with expressions in terms of g004, rather than *form-var*. To be exact, d and e could be associated with these expressions, respectively:

```
(car g004)
(car (cdr g004))
```

Let us now give the names to these concepts that are used in the implementation. Each pattern variable (real or made up by the parser) is represented by an "id" structure with three fields:

- name - For real pattern variables, this is the name given by the user in the pattern itself. For the pseudo-pattern-variables made up by the parser, this is a gensym'd name to be used as an iteration variable in the generated code.
- access - This is the expression to be associated with this variable; the expression tells the code generator how to access the part of the macro-call form corresponding to this pattern variable.
- control - For non-iterated pattern variables, this is the empty list. For variables inside ellipses, such as d and e above, this is the id structure for the corresponding iteration variable, made up by the parser.

Thus, the id structures for the variables a, b, c and g004 would have null control fields, while those for d and e would point to the id structure for g004.

Note that the chain of id structures through the control field could be arbitrarily long, depending only on how many times iterated sub-patterns are nested in the pattern. For example, consider the pattern variables type and body2 in the definition of etypecase earlier. The variable type is inside one ellipsis, but body2 is inside two. Thus, the parser would make up an id structure to represent the outer ellipsis (name it g005) and give it the access expression

```
(cdr (cdr form-var))
```

and a null control field. The id structure for type would have the access expression

```
(car g005)
```

and its control field would point to the id structure for g005. To represent body2, though, a second, nested iteration variable is required. The parser would create another id structure (name this one g006) with the access expression

```
(cdr (cdr g005))
```

The control field for this id is the id structure for g005. Finally, body2 would get an id whose access expression was simply "g006" and whose control field would point to the id structure for g006. Thus, the parser would have recorded the fact that body2 appears two levels deep in ellipses by giving its id structure a control chain that's two steps long (from body2 to g006 and from g006 to g005).

Given this understanding of the data structure in use, the function parse is easy to read. (Recall that a complete listing of the implementation code appears at the end of the article.) It takes four arguments: the list of keywords (so that it won't generate id structures for them), a piece of the pattern, an access expression to get this piece of the pattern, and the value to be placed in the control field of any new id structures. The parser returns a list of the id structures it creates.

The code-generation phase is carried out by the functions `gen` and `gen-loop`. Once the list of `id` structures has been constructed by the parser, the job of the code generator is a fairly simple recursion over the structure of the expansion template. The generator returns a `quasiquote`-expression (like a backquote-expression in other Lisp dialects) that evaluates to the proper expansion text.

When symbols are found in the expansion template, they are looked up in the list of `id`'s; if a match is found, then this is one of the pattern variables. The generated code is an `unquote` (like a use of comma in other Lisp dialects) wrapped around the access expression associated with the pattern variable. Symbols that are not the names of pattern variables and all other atoms are treated as constants in the expansion template and are simply returned as-is.

For list-structured expansion templates, there are two cases, depending upon whether or not the first element of the list is immediately followed by an ellipsis. If it isn't, then the code generator can simply recursively descend both the `car` and the `cdr` of the template and `cons` the resulting code together. For the iterated case, however, some more mechanism is required.

Ellipses in the expansion template must be translated into loops that map over some pieces of the macro-call form, each time executing the code generated for the iterated sub-template. For example, recall the `foo` pattern given as an example in the discussion of parsing above. If the corresponding expansion template were the following:

```
(bar (a b c) (e d) ...)
```

then the code generated for the "`(e d) ...`" piece would involve a loop looking something like this:

```
(map (lambda (g004) `(,(car (cdr g004)) ,(car g004)))
     (cdr (cdr (cdr form-var))))
```

But how is the code-generator to know what to loop over? The answer is that it is determined entirely by which pattern variables are used in the iterated sub-template. Each one's chain of `control` variables indicates the names of the variables that need to be bound in the loop and also the expressions over whose values the loop is supposed to map. In the case above, because the variables `e` and `d` appeared in the iterated sub-template, their common `control` variable `g004` was bound in the loop and `g004`'s access expression provided the value to map over.

The code generator therefore passes around an extra argument aside from the piece of the template and the list of `id` structures; each call to `gen` also includes a list of structures representing the loops inside of which the current piece of the template is nested. These `loop` structures are very simple, having a single "`ids`" field that will hold a list of the `id` structures for the iteration variables to be bound in that loop. Each time the generator comes across a pattern variable, its chain of `control` variables is traversed one-by-one in parallel with the list of surrounding loops; the current `control` variable is added to the list of `id`'s in the current loop. (This traversal is carried out by the `add-control!` procedure.) Thus, in a sense, the pattern variables are responsible for recording their iteration needs in the list of `loop` structures. The loop-generator then uses that information to construct the looping form.

So finally we can see how the iterated sub-template case of the code generator should work. It recursively generates code for the sub-template, passing along a new list of `loop` structures that includes a freshly-minted one for this loop. The information stored into that new `loop` structure then guides the construction of a `map` expression that will contruct a list of expansions. This `map` expression is wrapped in an `unquote-splicing` expression (like a use of comma-at-sign in other Lisp dialects) and the lot is consed onto the generated code for the the part of the template following the ellipsis.

The interested reader of the code should now be able to understand precisely how the expansion function for `let` expressions, shown earlier, was generated. Because every step of the parsing and code generation is simple, only a few simple idioms appear in the final code. With a little more complexity in the

implementation, the nicer-looking (though not very much faster-executing) code produced by Dybvig's unsimplified algorithm can be produced.

As I mentioned at the beginning of the column, the version of `extend-syntax` described here is somewhat simpler than the true version implemented in the *Chez* Scheme environment. That version allows for selectively escaping from the pure world of pattern-matching into arbitrary Scheme code and for adding an extra condition, written in Scheme, to each transcription clause in order to more finely control when a given clause will be selected. I encourage those of you with an interest to take a look at the full description in Dybvig's book and then to try adding support for the missing features to this implementation.

One final puzzle for the reader: about the tenth time that I used `extend-syntax` for a real macro in my work, I discovered that there was a facility missing from the ellipsis mechanism that could greatly simplify certain macros. It concerns the desire occasionally to apply an ellipsis to *more than one* sub-pattern or sub-template. For example, suppose that I wanted to implement a "flattened" version of `let` in which the parentheses around each variable-binding are not given. That is, the form

```
(flat-let (a 1 b 2) body)
```

should expand into

```
(let ((a 1) (b 2)) body)
```

Working in the other direction, I might want to implement another version of `let` that takes two variables and two values in each binding. Thus, the form

```
(doublet ((a b 1 2) (c d 3 4)) body)
```

should expand into

```
(let ((a 1) (b 2) (c 3) (d 4)) body)
```

It is somewhat problematical to implement these with the simple `extend-syntax` given here. But suppose that I could "group together" sub-patterns and sub-templates with the symbols { and } for the purposes of ellipses. Then, I could write these two macros as follows:

```
(extend-syntax (flat-let)
   ( (flat-let ({ var val } ...) body ...)
     (let ((var val) ...) body ...)))

(extend-syntax (doublet)
   ( (doublet ((var1 var2 val1 val2) ...) body ...)
     (let ({(var1 val1) (var2 val2)} ...) body ...)))
```

Can you see how to elegantly add this functionality to the implementation we've seen?

Well, that ties it up for this issue. On the next few pages you'll find the notes on reading Scheme for Common Lisp programmers followed by a complete listing of the code for the implementation. If you have comments on the algorithm or implementation, please send them to me; I'll pass them along to Mr. Dybvig as appropriate.

Finally, remember this: whether you've got a cute hack or an elegant example of Lisp code as Art, send it along. I'm also open to suggestions, comments and/or criticisms of the column, so feel free!

Some notes on Scheme for Common Lisp programmers

The implementation discussed in this issue's column is written in Scheme, but should be readable by most Common Lisp programmers. There are a few features of the Scheme language that deserve explanation, though.

Scheme uses the notations `#t` and `#f` to represent the boolean values true and false.

Several functions exist in both Scheme and Common Lisp, but with different names. Of particular interest for this issue are the Scheme functions `pair?` and `map`, which correspond closely to the Common Lisp functions `consp` and `mapcar`, respectively.

Scheme does not treat the names of functions differently from normal variables. It thus does not need a facility akin to the `function` special form in Common Lisp. Where a Common Lisp program might say

```
(mapcar #'(lambda (f) (funcall f 2 3))
        (list #'+ #'* #'-))
```

the equivalent Scheme program is

```
(map (lambda (f) (f 2 3))
     (list + * -))
```

Both programs yield the list `(5 6 -1)`.

The Scheme form

```
(define (name arg ...)
   body ...)
```

is analogous to the Common Lisp form

```
(defun name (arg ...)
   body ...)
```

While Scheme supports roughly the same backquote/comma notation as Common Lisp, it treats them somewhat differently at read time. In the same way that both languages specify that `'foo` reads as the list `(quote foo)`, Scheme specifies that `` `foo `` reads as `(quasiquote foo)`, that `,foo` reads as `(unquote foo)`, and that `,@foo` reads as `(unquote-splicing foo)`. Thus, programs can write code using the backquote/comma facility without having to go through the reader.

Finally, Scheme provides an iteration facility known as "named `let`". The form

```
(let name ((var1 expr1)
           ...
           (vark exprk))
   body ...)
```

is much like the Common Lisp form

```
(labels ((name (var1 ... vark)
               body ...))
   (name expr1 ... exprk))
```

except that, in Scheme, the *expr*s are not in the scope of *name*. The named `let` form is a very convenient notation for simple loops and recursions.

```
;;; extend.ss
;;; Copyright (C) 1987 R. Kent Dybvig

;;; The basic design of extend-syntax is due to Eugene Kohlbecker.  See
;;; "E. Kohlbecker: Syntactic Extensions in the Programming Language Lisp",
;;; Ph.D.  Dissertation, Indiana University, 1986."  The structure of "with"
;;; pattern/value clauses, the method for compiling extend-syntax into
;;; Scheme code, and the actual implementation are due to Kent Dybvig.


;;; Pattern matching

   (define (syntax-match? keywords pattern s-expr)
      (cond
         ((symbol? pattern)
            (if (memq pattern keywords)
               (eq? s-expr pattern)
               #t))
         ((pair? pattern)
            (if (equal? (cdr pattern) '(...))
               (let loop ((tail s-expr))
                  (or (null? tail)
                      (and (pair? tail)
                           (syntax-match? keywords (car pattern) (car tail))
                           (loop (cdr tail)))))
               (and (pair? s-expr)
                    (syntax-match? keywords (car pattern) (car s-expr))
                    (syntax-match? keywords (cdr pattern) (cdr s-expr)))))
         (else
            (equal? s-expr pattern))))
```

```
;;; Parsing

   (define-structure (id
      name
      access
      control))

   (define (lookup name id-list)
      (let loop ((tail id-list))
         (cond ((null? tail)
                  #f)
               ((eq? name (id-name (car tail)))
                  (car tail))
               (else
                  (loop (cdr tail))))))

   (define (add-car access)
      `(car ,access))

   (define (add-cdr access)
      `(cdr ,access))

   (define (parse keywords pattern access control)
      (cond ((symbol? pattern)
               (if (memq pattern keywords)
                  '()
                  (list (make-id pattern access control))))
            ((pair? pattern)
               (if (equal? (cdr pattern) '(...))
                  (let ((iter-var (gensym)))
                     (parse keywords
                            (car pattern)
                            iter-var
                            (make-id iter-var access control)))
                  (append (parse keywords
                                 (car pattern)
                                 (add-car access)
                                 control)
                          (parse keywords
                                 (cdr pattern)
                                 (add-cdr access)
                                 control))))
            (else
               '())))
```

```
;;; Code-generation

(define-structure (loop
   ids))

(define (add-control! id loops)
   (if (not (null? id))
      (cond ((null? loops)
             (error 'extend-syntax "missing ellipsis in expansion"))
            (else
               (let ((id-list (loop-ids (car loops))))
                  (if (not (memq id id-list))
                     (set-loop-ids! (car loops) (cons id id-list))))
               (add-control! (id-control id) (cdr loops))))))

(define (gen expansion ids loops)
   (cond ((symbol? expansion)
          (let ((id (lookup expansion ids)))
             (if (not id)
                expansion
                (begin
                   (add-control! (id-control id) loops)
                   (list 'unquote (id-access id))))))
         ((pair? expansion)
            (cond ((and (pair? (cdr expansion))
                        (eq? (cadr expansion) '...))
                   (let ((new-loop (make-loop '())))
                      (cons (list 'unquote-splicing
                                  (gen-loop new-loop
                                            (gen (car expansion)
                                                 ids
                                                 (cons new-loop loops))))
                            (gen (cddr expansion) ids loops))))
                  (else
                    (cons (gen (car expansion) ids loops)
                          (gen (cdr expansion) ids loops)))))
         (else
            expansion)))

(define (gen-loop loop body)
   (let ((ids (loop-ids loop)))
      (cond ((null? ids)
             (error 'extend-syntax "extra ellipsis in expansion"))
            (else
               `(map (lambda ,(map id-name ids)
                             ,(add-quasi body))
                     ,@(map id-access ids))))))
```

```
;;; The top-level framework

(define (add-quasi form)
   (list 'quasiquote form))

(define (make-clause keywords clause form-var)
   (cond ((= 2 (length clause))
          (let* ((pattern (car clause))
                 (expansion (cadr clause))
                 (id-list (parse keywords pattern form-var '())))

            `((syntax-match? ',keywords ',pattern ,form-var)
               ,(add-quasi (gen expansion id-list '())))))
         (else
            (error 'extend-syntax clause "invalid clause"))))

(define (make-syntax keys clauses)
   (let ((form-var (gensym)))
      `(lambda (,form-var)
         (cond ,@(map (lambda (clause)
                        (make-clause keys clause form-var))
                      clauses)
               (else
                  (error ',(car keys) ,form-var "invalid syntax"))))
```