JRM's Syntax-rules Primer for the Merely Eccentric

In learning to write Scheme macros, I have noticed that it is easy to find both trivial examples and extraordinarily complex examples, but there seem to be no intermediate ones. I have discovered a few tricks in writing macros and perhaps some people will find them helpful.

The basic purpose of a macro is *syntactic* abstraction. As functions allow you to extend the functionality of the underlying Scheme language, macros allow you to extend the syntax. A well designed macro can greatly increase the readability of a program, but a poorly designed one can make a program completely unreadable.

Macros are also often used as a substitute for functions to improve performance. In an ideal world this would be unnecessary, but compilers have limitations and a macro can often provide a workaround. In these cases, the macro should be a drop-in replacement for the equivalent function, and the design goal is not to extend the syntax of the language but to mimic the existing syntax as much as possible.

Very simple macros

SYNTAX-RULES provides very powerful pattern-matching and destructuring facilities. With very simple macros, however, most of this power is unused. Here is an example:

```
(define-syntax nth-value
  (syntax-rules ()
    ((nth-value n values-producing-form)
     (call-with-values
       (lambda () values-producing-form)
       (lambda all-values
         (list-ref all-values n))))))
```

When using functions that return multiple values, it is occasionally the case that you are interested in only one of the return values. The NTH-VALUE macro evaluates the VALUES-PRODUCING-FORM and extracts the Nth return value.

Before the macro has been evaluated, Scheme would treat a form that begins with NTH-VALUE as it would any other form: it would look up the value of the variable NTH-VALUE in the current environment and apply it to the values produced by evaluating the arguments.

DEFINE-SYNTAX introduces a new special form to Scheme. Forms that begin with NTH-VALUE are no longer simple procedure applications. When Scheme processes such a form, it uses the SYNTAX-RULES we provide to rewrite the form. The resulting rewrite is then processed in place of the original form.

Forms are only rewritten if the operator position is an IDENTIFIER that has been DEFINE-SYNTAX'd. Other uses of the identifier are not rewritten.

You cannot write an infix macro, nor can you write a macro in a nested position, i.e., an expression like ((foo x) y z) is always considered a procedure application. (The subexpression (foo x) will be rewritten of course, but it will not be able to affect the processing of subexpressions y and z.) This will be important later on.

SYNTAX-RULES is based on token-replacement. SYNTAX-RULES defines a series of patterns and templates. The form is matched against the pattern and the various pieces are transcribed into the template. This seems simple enough, but there is one important thing to always keep in mind.

THE SYNTAX-RULES SUBLANGUAGE IS NOT SCHEME!

This is a crucial point that is easy to forget. In the example,

```
(define-syntax nth-value
  (syntax-rules ()
    ((nth-value n values-producing-form)
     (call-with-values
       (lambda () values-producing-form)
       (lambda all-values
         (list-ref all-values n))))))
```

the pattern (nth-value n values-producing-form) looks like Scheme code and the template

```
(call-with-values
    (lambda () values-producing-form)
    (lambda all-values
      (list-ref all-values n)))
```

really seems to be Scheme code, but when Scheme is applying the syntax-rules rewrite there is NO SEMANTIC MEANING attached to the tokens. The meaning will be attached at a later point in the process, but not here.

One reason this is easy to forget is that in a large number of cases it doesn't make a difference, but when you write more complicated rules you may find unexpected expansions. Keeping in mind that syntax-rules only manipulates patterns and templates will help avoid confusion.

This example makes good use of patterns and templates. Consider the form (nth-value 1 (let ((q (get-number))) (quotient/remainder q d))) During the expansion of NTH-VALUE, the *entire* subform (let ((q (get-number))) (quotient/remainder q d)) is bound to VALUES-PRODUCING-FORM and transcribed into the template at (lambda () values-producing-form) The example would not work if VALUES-PRODUCING-FORM could only be bound to a symbol or number.

A pattern consists of a symbol, a constant, a list (proper or improper) or vector of more patterns, or the special token "..." (A series of three consecutive dots in this paper will *always* mean the literal token "..." and *never* be used for any other reason.) It is not allowed to use the same symbol twice in a pattern. Since the pattern is matched against the form, and since the form *always* starts with the defined keyword, it does not participate in the match.

You may reserve some symbols as literals by placing them in the list that is the first argument to syntax-rules. Essentially, they will be treated as if they were constants but there is some trickiness that ensures that users of the macro can still use those names as variables. The trickiness does the right thing so I won't go into details.

You may find macros written using the token "_" rather than repeating the name of the macro:

```
(define-syntax nth-value
  (syntax-rules ()
    ((_ n values-producing-form)
     (call-with-values
       (lambda () values-producing-form)
       (lambda all-values
         (list-ref all-values n))))))
```

I personally find this to be confusing and would rather duplicate the macro name.

Here are the rules for pattern matching:

- A constant pattern will only match against an EQUAL? constant. We'll exploit this later on.
- A symbol that is one of the literals can only match against the exact same symbol in the form, and then only if the macro user hasn't shadowed it.
- A symbol that is *not* one of the literals can match against *any* complete form. (Forgetting this can lead to surprising bugs.)
- A proper list of patterns can only match against a list form of the same length and only if the subpatterns match.
- An improper list of patterns can only match against a list form of the same or greater length and only if the subpatterns match. The dotted tail of the pattern will be matched against the remaining elements of the form. It rarely makes sense to use anything but a symbol in the dotted tail of the pattern.
- The ... token is special and will be discussed a bit later.

Debugging macros

As macros get more complicated, they become trickier to debug. Most Scheme systems have a mechanism by which you can invoke the macro expansion system on a piece of list structure and get back the expanded form. In MzScheme you could do this:

```
(syntax-object->datum
   (expand '(nth-value 1 (let ((q (get-number)))
                           (quotient/remainder q d)))))
```

In MIT Scheme,

```
(unsyntax (syntax '(nth-value 1 (let ((q (get-number)))
                                  (quotient/remainder q d)))
                  (nearest-repl/environment)))
```

Be prepared for some interesting output — you may not realize how many forms are really macros and how much code is produced. The macro system may recognize and optimize certain patterns of function usage as well. It would not be unusual to see (caddr x) expand into (car (cdr (cdr x))) or into (%general-car-cdr 6 x).

Be prepared, too, for some inexplicable constructs. Some syntax objects may refer to bindings that are only lexically visible from within the expander. Syntax objects may contain information that is lost when they are converted back into list structure. You may encounter apparently illegal expansions like this:

```
(lambda (temp temp temp) (set! a temp) (set! b temp) (set! c temp))
```

There are three internal syntax objects that represent the three different parameters to the lambda expression, and each assignment referred to a unique one, but each individual syntax object had the same symbolic name, so their unique identity was lost when they were turned back into list structure.

Debugging trick

One very easy debugging trick is to wrap the template with a quote:

```
(define-syntax nth-value
  (syntax-rules ()
    ((_ n values-producing-form)
     '(call-with-values                    ;; Note the quote!
        (lambda () values-producing-form)
        (lambda all-values
          (list-ref all-values n))))))
```

Now the macro returns the filled-in template as a quoted list:

```
(nth-value (compute-n) (compute-values))

=> (call-with-values (lambda () (compute-values))
     (lambda all-values (list-ref all-values (compute-n))))
```

Sometimes it is difficult to understand why a pattern didn't match something you thought it should or why it did match something it shouldn't. It is easy to write a pattern testing macro:

```
(define-syntax test-pattern
  (syntax-rules ()
    ((test-pattern one two) "match 1")
    ((test-pattern one two three) "match 2")
    ((test-pattern . default) "fail")))
```

Debugging trick

A second trick is to write a debugging template:

```
(define-syntax nth-value
   (syntax-rules ()
     ((_ n values-producing-form)
      '("Debugging template for nth-value"
        "n is" n
        "values-producing-form is" values-producing-form))))
```

N-ary macros

By using a dotted tail in the pattern we can write macros that take an arbitrary number of arguments.

```
(define-syntax when
  (syntax-rules ()
    ((when condition . body) (if condition (begin . body) #f))))
```

An example usage is

```
(when (negative? x)
   (newline)
   (display "Bad number:  negative."))
```

The pattern matches as follows:

```
condition = (negative? x)

body = ((newline) (display "Bad number:  negative."))
```

Since the pattern variable body is in the dotted tail position, it is matched against the list of remaining elements in the form. This can lead to unusual errors. Suppose I had written the macro this way:

```
(define-syntax when
  (syntax-rules ()
    ((when condition . body) (if condition (begin body) #f))))
```

The pattern is matched against the list of remaining arguments, so in the template it will expand to a list:

```
(when (negative? x)
  (newline)
  (display "Bad number:  negative."))
```

expands to

```
(if (negative? x)
    (begin ((newline) (display "Bad number:  negative.")))
    #f)
```

But this *almost* works. The consequence of the condition is evaluated as if it were a function call. The function is the return value of the call to newline and the argument the return value from display. Since the rules for evaluation are to evaluate the subforms and then apply the resulting procedure to the resulting arguments, this may actually print a newline and display the string “Bad number: negative.” before raising an error. One could easily be fooled into thinking that the WHEN form succesfully ran to completion and it was the code *subsequent* to the WHEN that raised the error.

The original code had this in the template: (begin . body)

When the template is filled in, the body is placed in the dotted tail. Since the body is a list of forms, the effect is as if you had used CONS rather than LIST to create the resultant form. Unfortunately, this trick does not generalize; you can only prefix things in this manner.

> Macro rest args get bound to a list of forms, so remember to unlist them at some point.

There is another bug in the original form:

```
(define-syntax when
  (syntax-rules ()
    ((when condition . body) (if condition (begin . body) #f))))

(when (< x 5))
```

expands into

```
(if (< x 5) (begin) #f)
```

Recall that pattern variables match anything, including the empty list. The pattern variable BODY is bound to the empty list. When the template form (begin . body) is filled in, the token BEGIN is consed to the empty list resulting in an illegal (BEGIN) form.

Since (when (< x 5)) is unusual itself, one solution is to modify the macro like this:

```
(define-syntax when
  (syntax-rules ()
    ((when condition form . forms)
     (if condition (begin form . forms) #f))))
```

Now the pattern will match only if the WHEN expression has at least one consequent form and the resulting BEGIN subform is guaranteed to contain at least one form.

This sort of macro — one that takes an arbitrary number of subforms and evaluates them in an implicit begin — is extremely common. It is valuable to know this macro idiom:

> Implicit Begin idiom
>
> Use this idiom for a macro that allows an arbitrary number of subforms and processes them sequentially (possibly returning the value of the last subform).

> The pattern should end in “FORM . FORMS)” to ensure a minimum of one subform.
>
> The template has either (begin form . forms) or uses the implicit begin of another special form, e.g. (lambda () form . forms)

A strange and subtle pitfall

Suppose you are a former INTERCAL hacker and you truly miss the language. You wish to write a macro PLEASE that simply removes itself from the expansion:

```
(please display "foo")  =>  (display "foo")
```

Here is an attempt:

```
(define-syntax please
  (syntax-rules ()
    ((please . forms) forms)))
```

This works on some Scheme systems, but not on others and the reason is quite subtle. In Scheme, function application is indicated syntactically by a list consisting of a function and zero or more arguments. Above macro, although it returns such a list, creates that list as part of the pattern matching process. When a macro is expanded careful attention is paid to ensure that subforms from the point of use continue to refer to the lexical environment at that point while subforms that are introduced by the template continue to refer to the lexical environment of the macro definition. The list that is returned from the PLEASE macro, however, is a subform that was not created at either the macro use point *or* at the macro definition point, but rather in the environment of the pattern matcher. In MzScheme, that environment does not have a syntactic mapping to interpret a list as a function call, so the following error results:

```
"compile: bad syntax; function application is not allowed, because
 no #%app syntax transformer is bound in: (display 33)"
```

The fix is trivial:

```
(define-syntax please
  (syntax-rules ()
    ((please function . arguments) (function . arguments))))
```

The resulting expansion is now a list constructed within the template of the macro rather than one constructed by the pattern matcher. The template environment is used and the resulting list is therefore interpreted as a function call.

Again, this is an extremely subtle point, but it is easy to remember this rule of thumb:

> Don’t use macro rest arguments as an implicit function call. Use a template with an explicit (function . arguments) element.

Multiple patterns

Syntax rules allows for an arbitrary number of pattern/template pairs. When a form is to be rewritten, a match is attempted against the first pattern. If the pattern cannot be matched, the next pattern is examined. The template associated with the first matching pattern is the one used for the rewrite. If no pattern matches, an error is raised. We will exploit this heavily.

Syntax errors

The macro system will raise an error if no pattern matches the form, but it will become useful to us to write patterns that explicitly reject a form if the pattern *does* match. This is easily accomplished by making the template for a pattern expand into poorly formed code, but the resulting error message is rather unhelpful:

```
(define-syntax prohibit-one-arg
  (syntax-rules ()
    ((prohibit-one-arg function argument) (if)) ;; bogus expansion
    ((prohibit-one-arg function . arguments)
     (function . arguments))))

(prohibit-one-arg + 2 3)
  => 5

(prohibit-one-arg display 'foo)
 if: bad syntax (has 0 parts after keyword) in: (if)
```

To make a more helpful error message, and to indicate in the macro definition that the bogus expansion is intentional, we'll define a macro designed to raise a syntax error. (There is, no doubt a procedural way of doing this, but we wish to raise this error during macro expansion and the pattern language provides no way to do this directly. A more complicated macro system could be used, but this is nice, easy, and portable.)

```
(define-syntax syntax-error
  (syntax-rules ()
    ((syntax-error) (syntax-error "Bad use of syntax error!"))))
```

We can now write macros that expand into calls to syntax error. If the call contains any arguments, the pattern will not match and an error will be raised. Most scheme macro systems display the form that failed to match, so we can put our debugging messages there.

```
(define-syntax prohibit-one-arg
  (syntax-rules ()
    ((prohibit-one-arg function argument)
     (syntax-error
      "Prohibit-one-arg cannot be used with one argument."
      function argument))
    ((prohibit-one-arg function . arguments)
     (function . arguments))))

(prohibit-one-arg display 3)
=> syntax-error: bad syntax in: (syntax-error "Prohibit-one-arg cannot
be used with one argument." display 3)

   Write a syntax-error macro.
   Write `rejection` patterns by expanding into a call to
   syntax-error.
```

Accidental matching

The pattern roughly resembles what it matches, but this can be a source of confusion. A pattern like this:

```
(my-named-let name (binding . more-bindings) . body)
```

will match this form:

```
(my-named-let ((x 22)
               (y "computing square root"))
   (display y)
   (display (sqrt x)))
```

as follows:

```
name  =   ((x 22) (y "computing square root"))

binding = display
more-bindings = (y)

body = ((display (sqrt x)))

 Nested list structure in the pattern will match similar nested
 list structure in the form, but symbols in the pattern will match
 *anything*.
```

In this example we want to prohibit matching NAME with a list. We do this by guarding the intended pattern with patterns that should not be allowed.

```
(define-syntax my-named-let
  (syntax-rules ()

    ((my-named-let () . ignore)
     (syntax-error "NAME must not be the empty list."))
    ((my-named-let (car . cdr) . ignore)
     (syntax-error "NAME must be a symbol." (car . cdr)))

    ((my-named-let name bindings form . forms) ;; implicit begin
     (let name bindings form . forms))))
```

```
Protect against accidental pattern matching by writing guard
patterns that match the bad syntax.
```

Recursive expansion

Our syntax-error macro can expand into a syntax-error form. If the result of a macro expansion is itself a macro form, that resulting macro form will be expanded. This process continues until either the resulting form is not a macro call or the resulting form fails to match a pattern. The syntax-error macro is designed to fail to match anything but a no-argument call. A no-argument call to syntax-error expands into a one-argument call to syntax-error which fails to match.

The template for a macro can expand to a form that embeds a call to the same macro within it. The embedded code will be expanded normally if the surrounding code treats it as a normal form. The embedded call will normally contain fewer forms than the original call so that the expansion eventually terminates with a trivial expansion.

Note, however, that the recursive macro will not be expanded unless the intermediate code uses it as a macro call. This is why the debugging trick of quoting the template works: the macro form is expanded to a quote form. The quote form just treats its argument as data so no further expansion is done.

Recursive expansion always produces a nested result. This is used to good effect in this example:

```
(define-syntax bind-variables
  (syntax-rules ()
    ((bind-variables () form . forms)
     (begin form . forms))

    ((bind-variables ((variable value0 value1 . more) . more-bindings) form . forms)
     (syntax-error "bind-variables illegal binding" (variable value0 value1 . more)))

    ((bind-variables ((variable value) . more-bindings) form . forms)
     (let ((variable value)) (bind-variables more-bindings form . forms)))

    ((bind-variables ((variable) . more-bindings) form . forms)
     (let ((variable #f)) (bind-variables more-bindings form . forms)))

    ((bind-variables (variable . more-bindings) form . forms)
     (let ((variable #f)) (bind-variables more-bindings form . forms)))
    ((bind-variables bindings form . forms)
     (syntax-error "Bindings must be a list." bindings))))
```

BIND-VARIABLES is much like LET*, but you are allowed to omit the value in the binding list. If you omit the value, the variable will be bound to #F. You can also omit the parenthesis around the variable name.

```
(bind-variables ((a 1)        ;; a will be 1
                 (b)          ;; b will be #F
                 c            ;; so will c
                 (d (+ a 3))) ;; a is visible in this scope.
    (list a b c d))
```

When bind-variables is processed, its immediate expansion is this:

```
(let ((a 1))
  (bind-variables ((b)
                   c
                   (d (+ a 3)))
    (list a b c d)))
```

The LET form is now processed and as part of that processing the body of the LET will be expanded.

```
(bind-variables ((b)
                   c
                   (d (+ a 3)))
    (list a b c d))
```

This expands into another LET form:

```
(let ((b #f))
  (bind-variables (c
                   (d (+ a 3)))
    (list a b c d)))
```

The process terminates when the binding list is the empty list. At this point each level of expansion is placed back in its surrounding context to yield this nested structure:

```
(let ((a 1))
  (let ((b #f))
    (let ((c #f))
      (let ((d (+ a 3)))
        (list a b c d)))))
```

There are several things to note here:

1. The macro uses the implicit begin idiom.
2. There is a guard patterns to match bindings with more than one value form.

3. In each expansion, the first binding will be removed. The remaining bindings appear in the binding position at the recursive call, thus the macro is inductive over the list of bindings.
4. There is a base case of the induction that matches the empty binding list.
5. On each iteration a match is attempted on first binding against these patterns in order:

   ```
   (variable value0 value1 . more) a list of more than two elements

   (variable value) a list of exactly two elements

   (variable) a list of one element

   variable  not a list
   ```

   This order is used because the last pattern will match anything and would prevent the previous matches from being matched.

This macro idiom is also extremely common.

> List Recursion idiom
>
> This idiom is used to recursively process a list of macro subforms.
>
> The macro has a parenthesised list of items in a fixed position. This list may be empty (), but it may not be omitted.
>
> A pattern that matches the empty list at that position preceeds the other matches. The template for this pattern does not include another use of this macro.
>
> One or more patterns that have dotted tails in the list position are present. The patterns are ordered from most-specific to most-general to ensure that the later matches do not shadow the earlier ones. The associated templates are either syntax-errors or contain another use of this macro. The list position in the recursive call will contain the dotted tail component.

A minimal list recursion looks like this:

```
(define-syntax mli
  (syntax-rules ()

    ((mli ()) (base-case))

    ((mli (item . remaining)) (f item (mli remaining)))

    ((mli non-list) (syntax-error "not a list")))))
```

Note that the recursive call in the second clause uses the pattern variable REMAINING and that it is NOT dotted. Each recursive call therefore contains a shorter list of forms at the same point.

At this point, things are starting to get complicated. We can no longer look upon macros as simple rewrites. We are starting to write macros whose purpose is to control the actions of the macro processing engine. We will be writing macros whose purpose is not to produce code but rather to perform computation.

A macro is a compiler. It takes source code written in one language, i.e. Scheme with some syntactic extension, and it generates object code written in another language, i.e. Scheme *without* that syntactic extension. The language in which we will be writing these compilers is NOT Scheme. It is the pattern and template language of syntax-rules.

A compiler consists of three basic phases: a parser that reads the source language, a transformation and translation process that maps the source language semantics into constructs in the target language, and a generator that turns the resulting target constructs into code. Our macros will have these three identifiable parts. The parser phase will use pattern language to extract code fragments from the source code. The translator will operate on these code fragments and use them to construct and manipulate Scheme code fragments. The generator phase will assemble the resulting Scheme fragments of Scheme code with a template that will be the final result.

The pattern and template language of syntax-rules is an unusual implementation language. The pattern-driven rule model makes it easy to write powerful parsers. The template-driven output model makes code generation a snap. The automatic hygiene tracks the context of the code fragments and essentially allows us to manipulate the intermediate code fragments as elements in an abstract syntax tree. There is just one problem: the model of computation is non-procedural. Simple standard programming abstractions such as subroutines, named variables, structured data, and conditionals are not only different from Scheme, they don't exist in a recognizable form!

But if we look carefully, we will find our familiar programming abstractions haven't disappeared at all — they have been destructured, taken apart, and re-formed into strange shapes. We will be writing some strange-looking code. This code will be written in the form of a macro transformer, but it will not be a macro in the traditional sense.

When a macro is expanded the original form is rewritten into a new form. If the result of macro expansion is a new macro form, the expander then expands that result. So if the macro expands into another call to itself, we have written a tail-recursive loop. We made a minimal use of this above with the syntax-error macro, but now we will be doing this as a standard practice.

We encountered the list recursion idiom above. We can create an analagous list iteration:

```
(define-syntax bind-variables1
  (syntax-rules ()
    ((bind-variables1 () form . forms)
     (begin form . forms))

    ((bind-variables1 ((variable value0 value1 . more) . more-bindings) form . forms)
     (syntax-error "bind-variables illegal binding" (variable value0 value1 . more)))

    ((bind-variables1 ((variable value) . more-bindings) form . forms)
     (bind-variables1 more-bindings (let ((variable value)) form . forms)))

    ((bind-variables1 ((variable) . more-bindings) form . forms)
     (bind-variables1 more-bindings (let ((variable value)) form . forms)))

    ((bind-variables1 (variable . more-bindings) form . forms)
     (bind-variables1 more-bindings (let ((variable #f)) form . forms)))
    ((bind-variables1 bindings form . forms)
     (syntax-error "Bindings must be a list." bindings))))
```

Because we process the leftmost variable first, the resulting form will be nested in the reverse order from the recursive version. We will deal with this issue later and just write the bindings list backwards for now.

```
(bind-variables1 ((d (+ a 3)) ;; a is visible in this scope.
                  c           ;; c will be bound to #f
                  (b)         ;; so will b
                  (a 1))      ;; a will be 1
    (list a b c d))
```

This macro first expands into this:

```
(bind-variables1 (c
                  (b)
                  (a 1))
  (let ((d (+ a 3)))
    (list a b c d)))
```

But since this form is a macro, the expander is run again. This second expansion results in this:

```
(bind-variables1 ((b)
                 (a 1))
  (let ((c #f))
    (let ((d (+ a 3)))
      (list a b c d))))
```

The expander is run once again to produce this:

```
(bind-variables1 ((a 1))
  (let ((b #f))
    (let ((c #f))
      (let ((d (+ a 3)))
        (list a b c d)))))
```

Another iteration produces this:

```
(bind-variables1 ()
  (let ((a 1))
    (let ((b #f))
      (let ((c #f))
        (let ((d (+ a 3)))
          (list a b c d))))))
```

The next expansion does not contain a call to the macro:

```
(begin
  (let ((a 1))
    (let ((b #f))
      (let ((c #f))
        (let ((d (+ a 3)))
          (list a b c d))))))
```

We could call this the List Iteration Idiom, but let's take this in a completely different direction.

Notice that the template for most of the rules begins with the macro name itself in order to cause the macro expander to immediately re-expand the result. But let's ignore the expander and pretend that

> *the template form is a tail-recursive function call*

By removing the error checking and the multiple formats for argument bindings, we can see the essence of what is going on:

```
(define-syntax bind-variables1
  (syntax-rules ()
    ((bind-variables1 () . forms)
     (begin . forms))

    ((bind-variables1 ((variable value) . more-bindings) . forms)
     (bind-variables1 more-bindings (let ((variable value)) . forms)))))
```

BIND-VARIABLES1 is tail-recursive function. SYNTAX-RULES functions as a COND expression. The arguments to bind-variables1 are unnamed, but by placing patterns in the appropriate positions, we can destructure the values passed into named variables.

Let us demonstrate this insight through a simple example.

We want to mimic the Common Lisp macro MULTIPLE-VALUE-SETQ. This form takes a list of variables and form that returns multiple values. The form is invoked and the variables are SET! to the respective return values. A putative expansion might be this:

```
(multiple-value-set! (a b c) (generate-values))
 =>  (call-with-values (lambda () (generate-values))
        (lambda (temp-a temp-b temp-c)
          (set! a temp-a)
          (set! b temp-b)
          (set! c temp-c)))
```

For the sake of clarity, we'll start out with no error checking. Since our macros are tail-recursive, we can write separate subroutines for each part of the expansion.

First, we write the entry-point macro. This macro is the one that would be exported to the user. This macro will call the one that creates the temporaries and the necessary assignment expressions. It passes in empty lists as the initial values for these expressions.

```
(define-syntax multiple-value-set!
  (syntax-rules ()
    ((multiple-value-set! variables values-form)

     (gen-temps-and-sets
         variables
         ()  ;; initial value of temps
         ()  ;; initial value of assignments
         values-form))))
```

Assuming that gen-temps-and-sets does the right thing, we will want to emit the resulting code. The code is obvious:

```
(define-syntax emit-cwv-form
  (syntax-rules ()

    ((emit-cwv-form temps assignments values-form)
     (call-with-values (lambda () values-form)
       (lambda temps . assignments)))))
```

The temps and assignments are just pasted into the right spots.

Now we need to write the routine that generates a temporary and an assignment for each variable. We'll again use induction on the list of variables. When that list is empty, we'll call EMIT-CMV-FORM with the collected results. On each iteration we'll remove one variable, generate the temporary and assignment for that variable, and collect them with the other temporaries and assignments.

```
(define-syntax gen-temps-and-sets
  (syntax-rules ()

    ((gen-temps-and-sets () temps assignments values-form)
     (emit-cwv-form temps assignments values-form))

    ((gen-temps-and-sets (variable . more) temps assignments values-form)
     (gen-temps-and-sets
        more
       (temp . temps)
       ((set! variable temp) . assignments)
      values-form))))
```

Before we develop this further, though, there are some important points about this macro that should not be overlooked.

```
Did I ever tell you that Mrs. McCave
Had twenty-three sons, and she named them all Dave?
     -- Dr. Seuss
```

Our MULTIPLE-VALUE-SET! macro generates a temporary variable for each of the variables that will be assigned (this is in the second clause of GEN-TEMPS-AND-SETS). Unfortunately, all the temporary variables are named TEMP. We can see this if we print the expanded code:

```
(call-with-values (lambda () (generate-values))
  (lambda (temp temp temp)
    (set! c temp)
    (set! b temp)
    (set! a temp)))
```

```
Well, she did. And that wasn't a smart thing to do.
You see, when she wants one, and calls out "Yoo-Hoo!
Come into the house, Dave!" she doesn't get one.
All twenty-three Daves of hers come on the run!
      (Ibid.)
```

The importance of unique identifiers to avoid name collisions is taught at a very young age.

The funny thing is, though, the code works. There are actually three different syntactic objects (all named temp) that will be bound by the LAMBDA form, and each SET! refers to the appropriate one. But there are six identifiers here with the name TEMP. Why did the macro expander decide to create three pairs of associated syntax objects rather than six individual ones or two triplets? The answer lies in this template:

```
(gen-temps-and-sets
     more
    (temp . temps)
    ((set! variable temp) . assignments)
    values-form)
```

The variable TEMP that is being prepended to the list of temps and the variable TEMP in the newly created (SET! VARIABLE TEMP) form will refer to the *same* syntactic object because they are created during the same expansion step. It will be a *different* syntactic object than any created during any other expansion step.

Since we run the expansion step three times, one for each variable to be assigned, we get three variables named temp. They are paired up properly because we generated all references to them at the same time.

> Introduce associated code fragments in a single expansion step.
>
> Introduce duplicated, but unassociated fragments in different expansion steps.

Let's explore two variants of this program. We will separate the genaration of the temps and the assignments into two different functions, GEN-TEMPS and GEN-SETS.

Because both GEN-TEMPS and GEN-SETS iterate over the list of variables as they operate, we modify MULTIPLE-VALUE-SET! to pass the list twice. GEN-TEMPS will do induction over one copy, GEN-SETS will work with the other.

```
(define-syntax multiple-value-set!
  (syntax-rules ()
    ((multiple-value-set! variables values-form)

     (gen-temps
         variables ;; provided for GEN-TEMPS
         ()  ;; initial value of temps
         variables ;; provided for GEN-SETS
         values-form))))
```

GEN-TEMPS does induction over the first list of variables and creates a list of temp variables.

```
(define-syntax gen-temps
  (syntax-rules ()

    ((gen-temps () temps vars-for-gen-set values-form)
     (gen-sets temps
               vars-for-gen-set
               () ;; initial set of assignments
               values-form))

    ((gen-temps (variable . more) temps vars-for-gen-set values-form)
     (gen-temps
       more
       (temp . temps)
       vars-for-gen-set
       values-form))))
```

GEN-SETS also does induction over the list of variables and creates a list of assignment forms.

```
(define-syntax gen-sets
  (syntax-rules ()

    ((gen-sets temps () assignments values-form)
     (emit-cwv-form temps assignments values-form))

    ((gen-sets temps (variable . more) assignments values-form)
      (gen-sets
       temps
       more
       ((set! variable temp) . assignments)
       values-form))))
```

Although the result of expansion looks similar, this version of the macro does not work. The name TEMP that is generated in the GEN-TEMPS expansion is different from the name TEMP generated in the GEN-SETS expansion. The expansion will contain six unrelated identifiers (all named TEMP).

But we can fix this. Notice that GEN-SETS carries around the list of temps generated by GEN-TEMPS. Instead of generating a new variable named TEMP, we can extract the previously generated TEMP from the list and use that.

First, we arrange for GEN-TEMPS to pass the temps twice. We need to destructure one of the lists to do the iteration, but we need the whole list for the final expansion.

```
(define-syntax gen-temps
  (syntax-rules ()

    ((gen-temps () temps vars-for-gen-set values-form)
     (gen-sets temps
               temps ;; another copy for gen-sets
```

```
                vars-for-gen-set
                () ;; initial set of assignments
                values-form))

    ((gen-temps (variable . more) temps vars-for-gen-set values-form)
     (gen-temps
       more
       (temp . temps)
       vars-for-gen-set
       values-form))))
```

GEN-SETS again uses list induction, but on two parallel lists rather than a single list (the lists are the same length). Each time around the loop we bind TEMP to a previously generated temporary.

```
(define-syntax gen-sets
  (syntax-rules ()

    ((gen-sets temps () () assignments values-form)
     (emit-cwv-form temps assignments values-form))

    ((gen-sets temps (temp . more-temps) (variable . more-vars) assignments values-fo
     (gen-sets
       temps
       more-temps
       more-vars
       ((set! variable temp) . assignments)
       values-form))))
```

Now we aren't generating new temps in GEN-SETS, we are re-using the old ones generated by GEN-SETS.

Ellipses

Ellipses are not really that difficult to understand, but they have three serious problems that make them seem mysterious when you first encounter them.

- The ... operator is very strange looking. It isn't a normal token and it has a prior established meaning in the English language that is somewhat at odds with what it does. Macros that use ellipses look like sound bites from movie reviews.
- The ... token is a reserved word in the macro language. Unlike all other tokens, it cannot be re-bound or quoted. In the macro language, ... acts like a syntactic mark of the same nature as a double-quote, dotted tail, or parenthesis. (It is nonsensical to try to use an open-parenthesis as a variable name! The same is true for ... in the macro language.) But in standard scheme, ... is an indetifier token and could conceivably be used as a variable.

- Out of every form in Scheme, the ... operator is the *only* one that is POSTFIX! The ... operator modifies how the PREVIOUS form is interpreted by the macro language. This one exception is probably responsible for most of the confusion. (Of course, if you were to use ... as a function in normal scheme, it is a prefix operator.)

In a macro pattern, ellipses can only appear as the last token in a LIST or VECTOR pattern. There must be a pattern immediately before it (because it is a postfix operator), so ellipses cannot appear right after an unmatched open-paren. In a pattern, the ellipses cause the pattern immediately before it to be able to match repeated occurrances of itself in the tail of the enclosing list or vector pattern. Here is an example.

Suppose our pattern were this:

```
(foo var #t ((a . b) c) ...)
```

This pattern would match:

```
(foo (* tax x) #t ((moe larry curly) stooges))
```

because var matches anything, #t matches #t, and one occurrance of ((a . b) c) would match. a would match moe, b would match the tail (larry curly) and c would match stooges.

This pattern would match:

```
(foo 11 #t ((moe larry curly) stooges)
           ((carthago delendum est) cato)
           ((egad) (mild oath)))
```

because var matches anything, #t matches #t, and three occurrances of the ((a . b) c) pattern would each match the three remaining elements. Note that the third element matches with a being egad, b being empty and the pattern c matching the entire list (mild oath).

This pattern would *not* match:

```
(foo 11 #t ((moe larry curly) stooges)
           ((carthago delendum est) cato)
           ((egad) mild oath))
```

The reason is that the final element ((egad) mild oath) cannot match ((a . b) c) because it the pattern is a list of two elements and we supplied a list of three elements.

This pattern *would* match:

```
(foo #f #t)
```

because var matches anything, #t matches #t, and zero occurrances of the ((a . b) c) pattern would match the empty tail.

> Ellipses must be last in a list or vector pattern. The list or vector must have at least one initial pattern.
>
> Ellipses is a postfix operator that operates on the pattern before it.

> Ellipses allows the containing list or vector pattern to match provided that the head elements of the pattern match the head elements of the list or vector up to (but not including) the pattern before the ellipses, *and* zero or more copies of that pattern match *each and all* of the remaining elements.
>
> Remember that zero occurrance of the pattern can match.

When a pattern has an ellipses, the pattern variables within the clause prior to the ellipses work differently from normal. When you use one of these pattern variables in the template, it must be suffixed with an ellipses, or it must be contained in a template subform that is suffixed with an ellipses. In the template, anything suffixed with an ellipses will be repeated as many times as the enclosed pattern variables matched.

Suppose our pattern is (foo var #t ((a . b) c) ...) and it is matched against

```
(foo 11 #t ((moe larry curly) stooges)
           ((carthago delendum est) cato)
           ((egad) (mild oath)))
```

These are example templates and the forms produced:

```
((a ...) (b ...) var (c ...))

=> ((moe carthago egad)
    ((larry curly) (delendum est) ())
    11
    (stooges cato (mild-oath)))

  ((a b c) ... var)

=>  ((moe (larry curly) stooges)
     (carthago (delendum est) cato)
     (egad () (mild oath)) 11)

((c . b) ... a ...)

=>  ((stooges larry curly)
    (cato delendum est)
    ((mild oath))
     moe carthago egad)

(let ((c 'b) ...) (a 'x var c) ...)

=>  (let ((stooges '(larry curly))
          (cato '(delendum est))
           ((mild oath) '()))

        (moe 'x 11 stooges)
        (carthago 'x 11 cato)
        (egad 'x 11 (mild-oath)))
```

There can be several unrelated ellipses forms in the pattern: The different subpatterns patterns need not have the same length in order for the entire pattern to match.

```
(foo (pattern1 ...) ((pattern2) ...) ((pattern3) ...))
```

will match against `(foo (a b) ((b) (c) (d) (e)) ())`

When a template is replicated, it is replicated as many times as the embedded pattern variable matched, so if you use variables from different subpatterns, the subpatterns must be the same length. (The pattern matching will work if the subpatterns are different lengths, but the template transcription will break if a subpattern uses two different lengths.)

You may have come to the conclusion that ellipses are too complicated to think about, and in the general case they can be very complex, but there are a couple of very common usage patterns that are easy to understand. Let's demonstrate an example.

We wish to write a macro TRACE-SUBFORMS that given an expression rewrites it so that some information is printed when each subform is evaluated. Let's start by using our list induction pattern.

```
(define-syntax trace-subforms
  (syntax-rules ()
    ((trace-subforms traced-form)
     (trace-expand traced-form
                   ()))))) ;; initialize the traced element list

(define-syntax trace-expand
  (syntax-rules ()

   ;; when no more subforms, emit the traced code.
   ((trace-expand () traced)
    (trace-emit . traced))

   ;; Otherwise, wrap some code around the form and CONS it to
   ;; the traced forms list.
   ((trace-expand (form . more-forms) traced)
    (trace-expand more-forms
      ((begin (newline)
              (display "Evaluating ")
              (display 'form)
              (flush-output)
              (let ((result form))
                (display " => ")
                (display result)
                (flush-output)
                result))
        . traced)))))

(define-syntax trace-emit
  (syntax-rules ()
    ((trace-emit function . arguments)
     (begin (newline)
            (let ((f function)
                  (a (list . arguments)))
            (newline)
            (display "Now applying function.")
            (flush-output)
            (apply f a))))))
```

Now let's try it:

```
(trace-subforms (+ 2 3))

Evaluating 3 => 3
Evaluating 2 => 2
Evaluating + => #<primitive:+>
Now applying function.
apply: expects type <procedure> as 1st argument,
    given: 3; other arguments were: (2 #<primitive:+>)
```

It evaluated the subforms backwards and tried to use the final argument as the function. The error, of course, is that as we worked from left to right on the list of subforms, the results were prepended to the list so they ended up in reverse order. We *could* interpose a list-reversing macro between trace-expand and trace-emit, but we can use ellipses here to good effect:

```
(define-syntax trace-subforms
  (syntax-rules ()

    ((trace-subforms (form ...))
     (trace-emit (begin
                  (newline)
                  (display "Evaluating ")
                  (display 'form)
                  (flush-output)
                  (let ((result form))
                    (display " => ")
                    (display result)
                    (flush-output)
                    result)) ...))))
```

The tracing code will be replicated as many times FORM is matched so the subforms will be handed to trace-emit in the correct order. The output is now

```
Evaluating + => #<primitive:+>
Evaluating 2 => 2
Evaluating 3 => 3
Now applying function.5
```

You may have noticed in our MULTIPLE-VALUE-SET! form that the assignments were happening in reverse order of the variable list. This didn't really matter because performing one assignment didn't affect the others. But let's print something out before each assignment and let's make the assignments happen in the correct order. We'll rewrite our MULTIPLE-VALUE-SET! macro using ellipses.

```
(define-syntax multiple-value-set!
  (syntax-rules ()
    ((multiple-value-set! variables values-form)

     (gen-temps-and-sets
         variables
         values-form))))

(define-syntax gen-temps-and-sets
  (syntax-rules ()

    ((gen-temps-and-sets (variable ...) values-form)
     (emit-cwv-form
        (temp)                          ;;     *
        ((begin
           (newline)
           (display "Now assigning ")
           (display 'variable)
           (display " to value ")
           (display temp)
           (force-output)
           (set! variable temp)) ...)
        values-form))))
```

We have a problem. We need as many temps as variables, but the form that generates the temporaries (marked with the asterisks) isn't replicated because it doesn't contain a pattern. We'll fix this by putting the variable in with the temps and stripping it back out in EMIT-CWV-FORM.

```
(define-syntax gen-temps-and-sets
  (syntax-rules ()

    ((gen-temps-and-sets (variable ...) values-form)
     (emit-cwv-form
         ((temp variable) ...)
         ((set! variable temp) ...)
         values-form))))

(define-syntax emit-cwv-form
  (syntax-rules ()

    ((emit-cwv-form ((temp variable) ...) assignments values-form)
     (call-with-values (lambda () values-form)
       (lambda (temp ...)
           . assignments)))))
```

We have another problem. This doesn't work. Recall that when we introduce new variables that we need to introduce unrelated copies in different expansions. Even though we create a list of temps, they are all created in a single expansion, so they will all be the same identifier in the resulting code. You cannot use the same identifier twice in a lambda argument list.

We will need to return to the list induction form, but the problem is that it generates the assignments in reverse order. We therefore use ellipses not to iterate over the variables, but to simulate APPEND:

```
(define-syntax multiple-value-set!
  (syntax-rules ()
    ((multiple-value-set! variables values-form)

     (gen-temps-and-sets
          variables
         ()  ;; initial value of temps
         ()  ;; initial value of assignments
         values-form))))

(define-syntax gen-temps-and-sets
  (syntax-rules ()

    ((gen-temps-and-sets () temps assignments values-form)
     (emit-cwv-form temps assignments values-form))

   ((gen-temps-and-sets (variable . more) (temps ...) (assignments ...) values-form)
    (gen-temps-and-sets
       more
       (temps ... temp)
       (assignments ... (begin
                          (newline)
                          (display "Now assigning value ")
                          (display temp)
                          (display " to variable ")
                          (display 'variable)
                          (flush-output)
                          (set! variable temp)))
       values-form))))

(define-syntax emit-cwv-form
  (syntax-rules ()

    ((emit-cwv-form temps assignments values-form)
     (call-with-values (lambda () values-form)
       (lambda temps . assignments)))))

(multiple-value-set! (a b c) (values 1 2 3))

Now assigning value 1 to variable a
Now assigning value 2 to variable b
Now assigning value 3 to variable c
```

When we use the ellipsis subpattern (temps ...) in the pattern, and the ellipses subtemplate (temps ... temp) in the template, this is analagous to writing (append temps (list temp)). This is a common idiom for extending a list `from the right hand end`. While appending to the right-hand end is a terrible idiom to use in normal Scheme, it is commonly used in macro expansion because expansion time is less important than runtime.

We can now see how to splice sublists when generating code. Suppose the pattern were this:

```
(foo (f1 ...) (f2 ...) . body-forms)
```

and we matched against this:

```
(foo (a b c d) (1 2 3 4) moe larry curly)
```

Then we can flatten the output by using this template:

```
 (f1 ... f2 ... . body-forms)

=>
 (a b c d 1 2 3 4 moe larry curly)
```

```
   Use ellipses to extend lists while retaining the order.

   Use ellipses to `flatten' output.
```

It isn't always convenient to write a separate auxiliary macro for each step of a complex macro. There is a trick that can be used to put labels on patterns. Since a string will only match another string if they are EQUAL, we can place a string in our patterns. A template that wishes to transfer to a particular pattern simply mentions the string. We can combine the gen-sets, gen-temps, and emit-cwv-form with this trick:

```
(define-syntax multiple-value-set!
  (syntax-rules ()
    ((multiple-value-set! (variable . variables) values-form)
     (mvs-aux "entry" variables values-form))

(define-syntax mvs-aux
  (syntax-rules ()
    ((mvs-aux "entry" variables values-form)
     (mvs-aux "gen-code" variables () () values-form))

    ((mvs-aux "gen-code" () temps sets values-form)
     (mvs-aux "emit" temps sets values-form))

    ((mvs-aux "gen-code" (var . vars) (temps ...) (sets ...) values-form)
     (mvs-aux "gen-code" vars
                        (temps ... temp)
                         (sets ... (set! var temp))
                         values-form))

    ((mvs-aux "emit" temps sets values-form)
     (call-with-values (lambda () values-form)
       (lambda temps . sets)))))
```

Let's compare this form of the macro GEN-TEMPS-AND-SETS

```
(define-syntax gen-temps-and-sets
  (syntax-rules ()

    ((gen-temps-and-sets () temps assignments values-form)
     (emit-cwv-form temps assignments values-form))

    ((gen-temps-and-sets (variable . more) temps assignments values-form)
     (gen-temps-and-sets
        more
       (temp . temps)
       ((set! variable temp) . assignments)
       values-form))))
```

with some Scheme code that iterates over a list and constructs a collection of s-expressions. This code is not a macro but it performs a superficially similar calculation:

```
(define (gen-temps-and-sets variables temps assignments values-form)
  (cond

     ((null? variables)
      (emit-cwv-form temps assignments values-form))

     ((pair? variables) (let ((variable (car variables))
                              (more     (cdr variables)))

                          (gen-temps-and-sets
                             more
                             `(temp ,@ temps)
                             `((set! ,variable temp) ,@ assignments)
                              values-form)))))
```

The similarity is striking:

- SYNTAX-RULES is a COND
- The pattern language is a hybrid of predicate (null? and pair?) and list destructuring code. A dotted pair in the pattern takes the CAR and CDR of the list provided it is a PAIR? (.)
- The list structure in the template is QUASIQUOTE (sans punctuation) and is equivalent to the appropriate CONS and LIST forms. The quoting is implicit: everything is quoted except those identifiers that were matched in the pattern.
- Ellipses does a mapcar/append sort of thing.

In our macro language we've identified conditionals, variables, tail-recursive calls, and list and pair data structures, and primitives such as CAR, CDR, CONS, APPEND, and even a weird mapping mechanism. What about non-tail-recursive function calls, data structure abstractions, and first-class functions? Unfortunately, these constructs are a bit more complex.

The macro language appears to be a variant of Scheme, but as seen through the looking glass, where things are not always as they appear. How can we characterize the language?

The most important difference between a form in Scheme and a form in the macro language is in the meaning of parenthesis. In Scheme, a parenthesized form is either a special form or a function call. In the macro language it is either a destructuring predicate or a constructor. But if we are using parenthesis for that purpose, we can't use them for function calls anymore.

- a template in the macro language with a macro name in the function position is a macro function call
- like Scheme, the value of variable is passed to the called procedure, not the name of the variable.
- like Scheme, arguments are positional
- unlike Scheme, the arguments are unnamed. Patterns must be positionally tested.
- unlike Scheme, subexpressions in the pattern can only indirectly invoke list manipulation functions like CAR, CDR, PAIR? and EQUAL? through a pattern matching mechanism.
- unlike Scheme, subexpressions in the template can only indirectly invoke list manipulation functions like CONS or APPEND through a quasiquote-like mechanism.

If we are missing the ability to call arbitrary functions from a subexpression, we are restricted to a linear calling sequence. For the macros written so far this is sufficient, but we will need something better for more complicated macros.

How do we write a macro subroutine? Our macro calls so far have been tail recursive — each routine transfered control to another until the emit code was finally called. A subroutine, however, needs a continuation to determine what macro to return to. We will explicitly pass the macro continuation to the macro subroutine. The subroutine will return by invoking the macro continuation on the values it has produced.

So let's use this technique to deeply reverse a list. The continuation is represented by a list of a string tag and whatever values the continuation may need to use (we don't yet have closures, so we carry the data along). To return a value, we destructure the continuation to obtain the tag and re-invoke sreverse.

```
(define-syntax sreverse
   (syntax-rules ()
     ((sreverse thing) (sreverse "top" thing ("done")))

     ((sreverse "top" () (tag . more))
      (sreverse tag () . more))

     ((sreverse "top" ((headcar . headcdr) . tail) kont)
      (sreverse "top" tail ("after-tail" (headcar . headcdr) kont)))

     ((sreverse "after-tail" new-tail head kont)
     (sreverse "top" head ("after-head" new-tail kont)))

     ((sreverse "after-head" () new-tail (tag . more))
     (sreverse tag new-tail . more))
```

```
    ((sreverse "after-head" new-head (new-tail ...) (tag . more))
     (sreverse tag (new-tail ... new-head) . more))

    ((sreverse "top" (head . tail) kont)
     (sreverse "top" tail ("after-tail2" head kont)))

    ((sreverse "after-tail2" () head (tag . more))
    (sreverse tag (head) . more))

    ((sreverse "after-tail2" (new-tail ...) head (tag . more))
    (sreverse tag (new-tail ... head) . more))

    ((sreverse "done" value)
     'value)))
```

Let's closely examine a simple call and return.

This clause recursively invokes sreverse on the head of the expression. It creates a new continuation by creating the list ("after-head" new-tail kont).

```
((sreverse "after-tail" new-tail head kont)
 (sreverse "top" head ("after-head" new-tail kont)))
```

This is where we will end up when that continuation is invoked. We expect the return value as the first argument after the tag. The remaining arguments after the tag are the remaining list elements of the continuation we made. As you can see here, "after-head" is itself going to invoke a continuation — the continuation saved within the continuation constructed above.

```
((sreverse "after-head" () new-tail (tag . more))
 (sreverse tag new-tail . more))

((sreverse "after-head" new-head (new-tail ...) (tag . more))
 (sreverse tag (new-tail ... new-head) . more))
```

Something very interesting is going on here. Let's add a rule to our macro that causes the macro to halt when the list we wish to reverse is the element 'HALT.

```
(define-syntax sreverse
  (syntax-rules (halt)
    ((sreverse thing) (sreverse "top" thing ("done")))

    ((sreverse "top" () (tag . more))
     (sreverse tag () . more))

    ((sreverse "top" ((headcar . headcdr) . tail) kont)
     (sreverse "top" tail ("after-tail" (headcar . headcdr) kont)))

    ((sreverse "after-tail" new-tail head kont)
     (sreverse "top" head ("after-head" new-tail kont)))
    ((sreverse "after-head" () new-tail (tag . more))
     (sreverse tag new-tail . more))

    ((sreverse "after-head" new-head (new-tail ...) (tag . more))
     (sreverse tag (new-tail ... new-head) . more))

    ((sreverse "top" (halt . tail) kont)
     '(sreverse "top" (halt . tail) kont))

    ((sreverse "top" (head . tail) kont)
     (sreverse "top" tail ("after-tail2" head kont)))

    ((sreverse "after-tail2" () head (tag . more))
     (sreverse tag (head) . more))

    ((sreverse "after-tail2" (new-tail ...) head (tag . more))
     (sreverse tag (new-tail ... head) . more))

    ((sreverse "done" value)
     'value)))

(sreverse (1 (2 3) (4 (halt)) 6 7))
=>
  (sreverse "top" (halt)
    ("after-head" ()
      ("after-tail2" 4
        ("after-head" (7 6)
           ("after-tail" (2 3)
            ("after-tail2" 1
              ("done")))))))
```

Notice how each continuation to sreverse contains the saved state for that continuation in addition to its parent continuation.

Let's take a huge leap of imagination:

The intermediate form is a last-in, first-out stack. The topmost element on the stack is the current function, the next few elements are the arguments, and this is followed by the return address and the remaining dynamic context. We have a stack of call frames.

I like to refer to this as stack-machine style.

We can exploit the analogy between the macro processor and a stack machine more fully by re-arranging the patterns and templates to place the continuation in a fixed spot. The logical spot in the first position rather than the last. In addition, if we change the way the continuation is constructed, i.e., change the format of the stack frame, we can arrange for the return value to be delivered to any desired location. Rather than place the return value immediately after the tag, we'll place the tag and the first few values of the frame in a list and upon returning we'll spread the list and put the value after the last element. This can all be done with an auxiliary macro:

```
(define-syntax return
  (syntax-rules ()

    ;; Continuation goes first.  Location of return value is indicated
    ;; by end of first list.
     ((return ((kbefore ...) . kafter) value)
     (kbefore ... value . kafter))

    ;; Special case to just return value from the null continuation.
    ((return () value) value)))
```

Constructing a continuation of the correct form is tedious, if we are computing something of the form (f a b (g x) c d), we have to write it something like this: (g ((f a b) c d) x) in order to compute G and get the result placed into (f a b <result> c d). We can write a helper macro for that. It will take 3 arguments, the current continuation, the saved elements to come before the return value, the call to the subroutine, and the saved elements after the return value. So if we want to express a nested function call like this, (f a b (g x) c d) we can write

```
(macro-subproblem (f a b) (g x) c d)
```

We are still limited to exactly one subproblem, though.

To facilitate calls with tags, we allow G to be a list that is spread at the head, so

```
(macro-subproblem (f a b) ((g "tag") x) c d)
```

becomes

```
  (g "tag" ((f a b) (c d)) x)
```

```
(define-syntax macro-subproblem
  (syntax-rules ()
    ((macro-subproblem before ((macro-function ...) . args) . after)
     (macro-function ... (before . after) . args))

    ((macro-subproblem before (macro-function . args) . after)
     (macro-function (before . after) . args))))
```

SREVERSE now becomes this:

```
(define-syntax sreverse
   (syntax-rules (halt)
     ((sreverse thing)
       (macro-subproblem
         (sreverse "done" ()) ((sreverse "top") thing)))

     ((sreverse "top" k ())
      (return k ()))
     ((sreverse "top" k ((headcar . headcdr) . tail))
      (macro-subproblem
        (sreverse "after-tail" k) ((sreverse "top") tail) (headcar . headcdr)))

     ((sreverse "after-tail" k new-tail head)
      (macro-subproblem
        (sreverse "after-head" k) ((sreverse "top") head) new-tail))

     ((sreverse "after-head" k () new-tail)
      (return k new-tail))

     ((sreverse "after-head" k new-head (new-tail ...))
      (return k (new-tail ... new-head)))

      ((sreverse "top" k (halt . tail))
      '(sreverse "top" k (halt . tail)))

     ((sreverse "top" k (head . tail))
      (macro-subproblem
        (sreverse "after-tail2" k) ((sreverse "top") tail) head))

     ((sreverse "after-tail2" k () head)
      (return k (head)))

     ((sreverse "after-tail2" k (new-tail ...) head)
      (return k (new-tail ... head)))

     ((sreverse "done" k value)
      (return k 'value))))
```

Although the calling syntax is still rather cumbersome, we have made it easy to write macro subroutines. Here is NULL?, CAR, PAIR? and LIST? written as macro subroutines:

```
(define-syntax macro-null?
  (syntax-rules ()
    ((macro-null? k ())         (return k #t))
     ((macro-null? k otherwise) (return k #f))))

(define-syntax macro-car
   (syntax-rules ()
    ((macro-car k (car . cdr)) (return k car))
    ((macro-car k otherwise)   (syntax-error "Not a list"))))

(define-syntax macro-pair?
   (syntax-rules ()
    ((macro-car k (a . b))   (return k #t))
    ((macro-car k otherwise) (return k #f))))
```

```
(define-syntax macro-list?
  (syntax-rules ()
     ((macro-car k (elements ...)) (return k #t))
    ((macro-car k otherwise)      (return k #f))))
```

The macro analog of IF suggests itself. We just tail call the pred after inserting IF-DECIDE into the continuation chain.

```
(define-syntax macro-if
   (syntax-rules ()
    ((macro-if (pred . args) if-true if-false)
     (pred ((if-decide) if-true if-false) . args))))

(define-syntax if-decide
  (syntax-rules ()
     ((if-decide #t if-true if-false) if-true)
    ((if-decide #f if-true if-false) if-false)))
```

An example use would be:

```
 (define-syntax whatis
  (syntax-rules ()
    ((whatis object) (whatis1 () object))))

(define-syntax whatis1
  (syntax-rules ()
    ((whatis1 k object)
      (macro-if (macro-null? object)
       (return k 'null)
       (macro-if (macro-list? object)
         (macro-subproblem
           (whatis1 "after car" k proper) (macro-car object))
         (macro-if (macro-pair? object)
           (macro-subproblem
             (whatis1 "after car" k improper) (macro-car object))
           (return k 'something-else)))))

    ((whatis1 "after car" k type c)
     (return k '(a type list whose car is c)))))
```

The first obvious drawback to these stack-machine style macros is the clumsy calling sequence. We can only call compute one subproblem at a time and we need to have a label to return to for each subproblem. As you might imagine, this, too, can be solved by a macro function.

However, there is a bit of a problem. If take the obvious approach and scan the macro call for parenthesized subexpressions, we have destroyed our ability to use templates as templates. We will need to be able to determine which parenthesis are not meant as function calls but are meant as list templates. Although we could re-introduce a quoting syntax, we would be losing most of the power of templates and we would be burying our code in a pile of quote and unquote forms.

Instead, let us place marks within the template to indicate what subexpressions are meant as subproblems.

```
(define-syntax macro-call
  (syntax-rules (!)
    ((macro-call k (! ((function ...) . arguments)))
     (function ... k . arguments))

     ((macro-call k (! (function . arguments)))
     (macro-call ((macro-apply k function)) arguments))

    ((macro-call k (a . b))
     (macro-call ((macro-descend-right k) b) a))

    ((macro-call k whatever) (return k whatever))))

(define-syntax macro-apply
  (syntax-rules ()
    ((macro-apply k function arguments)
     (function k . arguments))))

(define-syntax macro-descend-right
  (syntax-rules ()
    ((macro-descend-right k evaled b)
     (macro-call ((macro-cons k evaled)) b))))

(define-syntax macro-cons
   (syntax-rules ()
     ((macro-cons k ca cd) (return k (ca . cd)))))
```

So if we expand this form:

```
(macro-call ()
  (this is a (! (macro-cdr (a b (! (macro-null? ())) c d))) test))
```

The expansion is

```
(this is a (b #t c d) test)
```

Let's write a macro similar to LET. Since let-like binding lists are a common macro feature, we'd like to have a subroutine to parse them. We'd also like a subroutine to tell us if the first argument to our LET is a name or a binding list.

```
(define-syntax my-let
  (syntax-rules ()
     ((my-let first-subform . other-subforms)
     (macro-if (is-symbol? first-subform)
       (expand-named-let () first-subform other-subforms)
       (expand-standard-let () first-subform other-subforms)))))
```

To test if something is a symbol, we first test to see if it is a list or vector. If it matches, we simply return if-no. If it doesn't match, then we know it is atomic, but we don't know if it is a symbol. Recall that symbols in macros match anything, but other atoms only match things equal? to themselves. Oleg Kiselyov suggests this nasty trick: we build a syntax-rule using it and see if the rule matches a list.

```
(define-syntax is-symbol?
  (syntax-rules ()

    ((is-symbol? k (form ...))  (return k #f))
     ((is-symbol? k #(form ...)) (return k #t))

    ((is-symbol? k atom)
     (letrec-syntax ((test-yes (syntax-rules () ((test-yes) (return k #t))))
                     (test-no  (syntax-rules () ((test-no)  (return k #f))))
                     (test-rule
                       (syntax-rules ()
                         ((test-rule atom) (test-yes))
                         ((test-rule . whatever) (test-no)))))
            (test-rule (#f))))))
```

We need to use LETREC-SYNTAX here because we do not want our continuation forms to be in the template of the test-rule; if the test rule matches, the atom we just tested would be rewritten!

Parsing a binding-list list is easy to do with an ellipses:

```
(define-syntax parse-bindings
  (syntax-rules ()
     ((parse-bindings k ((name value) ...))
     (return k ((name ...) (value ...))))))

(define-syntax expand-named-let
  (syntax-rules ()
    ((expand-named-let k name (bindings . body))
      (macro-call k
         (! (emit-named-let name (! (parse-bindings bindings)) body))))))

(define-syntax expand-standard-let
  (syntax-rules ()
    ((expand-standard-let k bindings body)
     (macro-call k
       (! (emit-standard-let (! (parse-bindings bindings)) body))))))

(define-syntax emit-named-let
  (syntax-rules ()
    ((emit-named-let k name (args values) body)
     (return k (((lambda (name)
                  (set! name (lambda args . body))
                   name) #f) . values)))))
```

The obligatory hack.

The following is a small scheme interpreter written as a syntax-rules macro. It is incredibly slow.

```
(define-syntax macro-cdr
  (syntax-rules ()
    ((macro-cdr k (car . cdr)) (return k cdr))
    ((macro-cdr k otherwise)   (syntax-error "Not a list"))))

(define-syntax macro-append
   (syntax-rules ()
     ((macro-append k (e1 ...) (e2 ...)) (return k (e1 ... e2 ...)))))

(define-syntax initial-environment
  (syntax-rules ()
    ((initial-environment k)
     (return k ((cdr . macro-cdr)
                (null? . macro-null?)
                (append . macro-append)
                )))))

(define-syntax scheme-eval
  (syntax-rules ()
    ((scheme-eval expression)
     (macro-call ((quote))
       (! (meval expression (! (initial-environment))))))))

(define-syntax meval
  (syntax-rules (if lambda quote)
    ((meval k () env)  (return k ()))

    ((meval k (if pred cons alt) env)
     (macro-if (meval pred env)
       (meval k cons env)
       (meval k alt env)))

    ((meval k (lambda names body) env)
     (return k (closure names body env)))

    ((meval k (quote object) env) (return k object))

    ((meval k (operator . operands) env)
     (meval-list ((mapply k)) () (operator . operands) env))

    ((meval k whatever env)
     (macro-if (is-symbol? whatever)
       (mlookup k whatever env)
       (return k whatever)))))

(define-syntax mlookup
  (syntax-rules ()
    ((mlookup k symbol ())
     '(variable not found))
    ((mlookup k symbol ((var . val) . env))
     (macro-if (is-eqv? symbol var)
       (return k val)
       (mlookup k symbol env)))))

(define-syntax meval-list
  (syntax-rules ()
    ((meval-list k evaled () env)
     (return k evaled))

    ((meval-list k (evaled ...) (to-eval . more) env)
     (macro-call k
       (! (meval-list (evaled ... (! (meval to-eval env))) more env))))))

(define-syntax mapply
  (syntax-rules (closure)
    ((mapply k ((closure names body env) . operands))
     (macro-call k
       (! (meval body (! (extend-environment env names operands))))))


    ((mapply k (operator . operands))
     (macro-if (is-symbol? operator)
       (operator k . operands)
       '(non-symbol application)))))

(define-syntax extend-environment
  (syntax-rules ()
    ((extend-environment k env () ())
     (return k env))

   ((extend-environment k env names ())
    '(too-few-arguments))

   ((extend-environment k env () args)
    '(too-many-arguments))

   ((extend-environment k env (name . names) (arg . args))
    (extend-environment k ((name . arg) . env) names args))))
```