Building Little Languages With Macros

Picking up where language designers leave off

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi

A t some point, every language design has to end. The design of Java, for example, includes constructs from for loops to inner classes, but does not include a for each construct, a lex/yacc-style parser-generator form, call-by-reference methods, or many other constructs.

Macro systems let programmers continue where language designers left off and extend languages with new constructs. Furthermore, macro-extended languages continue to work with the basic tool chain—in contrast to lex- and yacc-style language extensions, for example, which require modified code generators, makefiles, and the like. Finally, if the macro system uses patterns to specify language extensions, macros are easily defined and understood in terms of a simple rewriting process.

One apparent drawback of pattern-based macros is that they cannot express interesting language extensions, such as new forms of variable binding or new forms of control. Macros in C are certainly limited in this regard, but pattern-based macros can be considerably more expressive in principle. In this article, we show how pattern-based macros in Scheme can express interesting language extensions, such as a call-by-reference extension of a call-by-value language.

A second drawback of macros in general is that they interact poorly with programming tools, such as type checkers and debuggers, which have difficulty correlating a program's original source and its macro-expanded version. Again, C macros certainly exhibit this problem, but Scheme macros play well with source-correlating tools, and they play well for much the same reason that they can implement interesting language extensions. Scheme macros thus provide a foundation for building little languages that fit neatly into existing programming environments.

Pattern-Based Macros

Most programmers are familiar with C-style pattern-based macros. In general, pattern-based macro definitions include pattern variables that match syntax fragments in a macro use. To expand a macro use, each pattern variable in the macro's template is replaced by the matching syntax fragment. For instance, in the ever-popular swap in Example 1(a), the swap(x,y) part is the pattern, x and y are pattern variables, and the (int tmp=y; y=x; x=tmp;) part is the template. Example 1(b) matches the pattern swap(x,y), where x is matched to c.red and y is matched to d->blue. Expansion proceeds by substituting c.red for x and d->blue for y in the template, producing Example 1(c). Although C macros are easy to understand, they have a deservedly poor reputation in that expansion is the only way to understand the macro. In particular, Example 1(d) expands to Example 1(e), which does not actually swap the values in tmp and other. The underlying problem is that the macro system manipulates plaintext instead of structured syntax fragments. As a result, the macro cannot implement your true intent, which is to have a private temporary variable, and it cannot help programming tools accurately relate the original source and its macro-expanded version. These problems with C macros are absent from Scheme's macro system.

Scheme Macro Basics

For simple tasks, Scheme macros look similar to C macros. The C macro header in Example 2(a) translates to Scheme as in Example 2(b). The syntax-rules part of the definition indicates that the macro takes "arguments," just as the parentheses after swap in C indicate a macro with arguments.

<table>
<thead>
<tr>
<th>Example 1: swap</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) #define swap(x,y) (int tmp=y; y=x; x=tmp;)</td>
</tr>
<tr>
<td>(b) swap(c.red, d-&gt;blue)</td>
</tr>
<tr>
<td>(c) (int tmp=d-&gt;blue; d-&gt;blue=c.red; c.red=tmp; )</td>
</tr>
<tr>
<td>(d) swap(tmp, other)</td>
</tr>
<tr>
<td>(e) (int tmp=other; other=tmp; tmp=tmp; )</td>
</tr>
</tbody>
</table>

Matthias is a professor at the College of Computer Science at Northeastern University; Robert is an assistant professor of computer science at the University of Chicago; Matthew is an assistant professor in the School of Computing at the University of Utah; and Shriram is an assistant professor of computer science at Brown University. They can be contacted at matthias@ccs.neu.edu, robbie@cs.uchicago.edu, mjflatt@cs.utah.edu, and sk@cs.brown.edu, respectively.
A C macro with arguments matches only one pattern—a certain number of comma-separated elements within parentheses. In contrast, a Scheme macro can match multiple patterns, and each pattern can have a specific structure, such as auxiliary keywords or nested parentheses. Therefore, syntax rules in a macro definition is followed by a sequence of auxiliary keywords (to distinguish them from pattern variables), then a sequence of pattern-template pairs. For swap, you have no keywords and you have only one pattern, as in Example 2(c).

A use of swap such as in Example 3(a) roughly expands to Example 3(b), which is Scheme notation for introducing a local variable temp1, then assigning y=x and x=temp1. In other words, let acts like a new block for variable bindings, and set! acts like "=" for assignment.

The macro system adds the subscript in temp1 to indicate that temp was introduced by the macro, and it has no relationship to other existing syntax. Thus, even if swap is used with a variable named temp, as in Example 3(c), this use of swap behaves as expected because the variables temp and temp1 are different in the expansion; see Example 3(d). Introducing subscripts, as in temp1, ensures that language extensions play well with the lexically scoped base language. Avoiding unintentional capture by introduced variable declarations, as in Example 3(d), is only half of the problem, however. The macro system must also prevent unintended capture of introduced variables, as in Example 3(e) where (define (f x) ... ) is Scheme notation for defining a function f that takes a single argument, x. The body of the function contains a local macro definition, swap-with-arg, followed by a block with local bindings of x and y and a body that uses swap-with-arg.

The local macro swap-with-arg swaps a variable's value with the value of f's x argument. Even if x has a different binding in the context where swap-with-arg is used, the x introduced by the macro expansion always refers to the function argument. In other words, in expanding the aforementioned definition, the second x must be renamed to preserve lexical scope; see Example 3(f).

In general, the macro system adds subscripts and superscripts as necessary to preserve lexical scope for the original identifiers.

**Matching Multiple Patterns**

A natural generalization of swap is the rotate macro, which rotates the values of a set of variables to the left. For example, (rotate bucket box bin) moves the value of box to bucket, the value of bin to box, and the value of bucket to bin. The rotate form should also work on four variables, as in (rotate n e s w), or any number of variables.

Rotating a set of variables is the same as swapping the first two, then rotating all but the first variable. Rotating a single variable has no effect. Based on these two observations, you can implement the rotate macro as in Example 4(a). The first pattern, (rotate a), matches when a single variable is supplied, and the expansion is the nonoperational (void) expression. The second pattern, (rotate a b c ...), contains ellipses that allow any number of matches for c—including zero matches. Thus, the macro use (rotate n e s w) matches the second case, and expands to Example 4(b), which in turn expands to Example 4(c), and so on, ending with Example 4(d). The Scheme begin form combines and orders statements, so the expansion in Example 4 rotates n through w by bubbling n's value through the other variables.
Instead of bubbling one value through all of the variables, you can define a `rotate` macro that directly moves each value to its target. The new `rotate` uses a helper macro, and it shows how ellipses can be placed after any subpattern or subtemplate to generate multiple instantiations of the subpattern or subtemplate; see Example 5(a). Using these definitions, `(rotate n e s w)` expands to a kind of parallel assignment in Example 5(b), which in turn expands to Example 5(c). The use of ellipses after the subtemplate (set! to from) causes to and from matches to be paired up and instantiated together. The `shift-to` macro fails if the number of from matches is not the same as the number of to matches, but the use in `rotate` always creates the same number of each.

**Identifier Macros**

The C macro system supports two kinds of macros — function-shaped macros like `swap`, and identifier-shaped macros that match only the macro name. For example:

```lisp
#define MAX_INT 0x7fffffff
```
defines a macro `MAX_INT` so that it is replaced in any context with `0x7fffffff`. A `syntax-rules` macro is always a `swap`-style macro. A `syntax-id-rules` macro is more like `MAX_INT`, but the identifier's expansion can be made sensitive to the way that it's used — as the target of a `set!` assignment, immediately after an open parenthesis, or as an immediate expression. (In Scheme, the position after `set!` or after an open parenthesis is generally special.)

**Example 4: rotate macro.**

```lisp
(a)
define-syntax rotate
  (syntax-rules ()
    ((rotate a) (void)) ; i.e., do nothing
    ((rotate a b c ...) (begin
      (swap a b)
      (rotate b c ...))))
(b)
  (begin
    (swap n e)
    (rotate e s w))
(c)
  (begin
    (swap n e)
    (begin
      (swap e c)
      (rotate s w)))
(d)
  (begin
    (swap n e)
    (begin
      (swap e s)
      (begin
        (swap s w)
        (void))))

Example 5: New rotate macro.

```lisp
(a)
define-syntax rotate
  (syntax-rules ()
    ((rotate a c ...) (shift-to (c ... a) (a c ...)))
    (define-syntax shift-to
      (syntax-rules ()
        ((shift-to (from0 from ...) (to0 to ...))
          (let ((tmp from0)
                  (set! to from0)
                  (set! to0 tmp))
            ...) )))
(b)
  (shift-to (n e s w) (v n e s))
(c)
  (let ((tmp n))
    (set! n e)
    (set! e s)
    (set! s w)
    (set! w n))
```
(define-syntax clock
  (syntax-id-rules (set!)
    ((set! clock e) (set-time! e))
    ((clock a ...) (error "clock is not a function"))
    (clock (get-time()))))

Example 6: Identifier macros.

(a)
(define-get/set-var clock get-time set-time!)
(define-get/set-var pwd getcwd setcwd)
(define-get/set-var user getuid setuid)
...

(b)
(define-syntax define-get/set-var
  (syntax-rules ()
    ((define-get/set-var id get set)
      (define-syntax id
        (syntax-id-rules (set!))
        ((set! id e) (set e))
        ((id a (...) ...) ((get) a (...) ...))
        (id (get))))))

Example 7: Macro-generating macros.

(a)
(define-cbr (f a b)
  (swap a b))

; Produces 2:
(let ((x 1) (y 2))
  (f x y))

(b)
(define (do-f get1 get2 set1 set2)
  (define-get/set-var a get1 set1)
  (define-get/set-var b get2 set2)
  (swap a b))

(c)
(let ((x 1) (y 2))
  (f x y))

(b)
(define (do-f get set)
  (define-get/set-var arg get set)
  body)

Example 9: define-cbr macro that expands.

(a)
(define-cbr-as-cbr do-f (arg ...)
  () body)

(b)
(define-cbr (f a b)
  (swap a b))

(c)
(begin
  (define-cbr-as-cbr do-f (a b)
    () (swap a b))
  (define-do-f ...))

(d)
(begin
  (define-cbr-as-cbr do-f ()
    ((a get1 set1) (b get2 set2) (swap a b))
    (define-do-f ...)))

Example 10: Replacement for ???.

For example, the definition in Example 6 causes clock to act like a normal Scheme variable, but assignments and uses of the variable are implemented through a set-time! mutator function and a get-time accessor function. With this macro, clock expands to (get-time), while (set! clock 5) expands to (set-time! 5).

The middle case in the macro causes (clock 8) to expand to an error because applying clock as a function is useless. For other macros, the middle case is useful. In fact, the syntax-rules form could be considered a special case of syntax-id-rules with only the middle case.

Macro-Generating Macros

Macro-generating macros let you create new forms for defining other macros. For example, suppose you want to convert many sets of mutator-accessor functions to pseudovariables like clock. Instead of copying the definition of clock and replacing the names, you would instead prefer to write Example 7(a). The define-get/set-var form can be implemented as a macro-generating macro, as in Example 7(b).
The (...) combination in a template produces a plain ... in the macro expansion. In other words, the first ellipses escape the second ellipses so that the macro-produced macro contains pattern- and template-duplicating ellipses.

**Extended Example: Call-by-Reference**

The *swap* and *rotate* examples are special cases of call-by-reference functions. Using only the pattern-based macro tools described so far, you can implement an extension of Scheme for defining call-by-reference functions. Our new `define-cbr` form allows programs such as Example 8(a). Since `f` is a call-by-reference function, the swap in `f` swaps the values of the `x` and `y` variables passed to `f` in the example call.

The expansion of `define-cbr` consists of two parts. The first part introduces a `do-f` call-by-value function that acts like a call-by-reference version of `f`. Instead of accepting `f's` arguments directly, it accepts accessor and mutator functions. In particular, the `f` definition in Example 8(a) expands to 8(b). Where the original function had two arguments, `a` and `b`, the expanded function has four arguments: `getf`, `setf`, which are functions to get and set `a`; and `getf2` and `setf2`, which get and set `b`. Inside the function, `a` and `b` are mapped to the accessor and mutator functions using our `define-get/set-var` macro. The other half of the transformation affects uses of `f`. The use of `f` in Example 8(c) expands to a call to `do-f`, where the arguments `x` and `y` are replaced with functions that get and set the values of `x` and `y`; see Example 8(d).

The `lambda` keyword in Scheme means "anonymous function," and `(lambda () x)` is an anonymous function that accepts no arguments and returns `x`. Similarly, `(lambda (v) (set! x v))` is an anonymous function that accepts one argument and puts it into `x`.

The key to connecting the definition of `f` to uses of `f` is that `define-cbr` is a macro-defining macro. It defines `f` as a macro that expands to uses of `do-f`. In other words, the `(define-cbr (f a b) (swap a b))` definition actually expands to Example 8(e).

Our task, then, is to implement a `define-cbr` macro that expands as above. Half of this macro is straightforward; see Example 9(a). The `???` part turns out to be more complex. If `define-cbr` allowed only a single `arg` for the defined function, then `???` could be simply Example 9(b). To support multiple arguments, however, you must generate many distinct `get` and `put` names to go with many `args`. To generate the names, you can exploit the way the macro expander adds subscripts to introduced variables. The idea is to take one `arg` at a time and introduce a corresponding `get` and `set`, leaving the rest of

```
(define-syntactic define-cbr-as-cbv
 (syntax-rules ()
  ([define-cbr-as-cbv do-f (arg0 arg ...)]
   (gens ...)
   (body))
  ([define-cbr-as-cbv do-f (arg ...)]
   (gens ... (arg0 get set))
   (body))
  ([define-cbr-as-cbv do-f ()]
   (get set)
   (body))
  ([define-cbr-as-cbv do-f []]
   (get set)
   (body))
  ))
```

**Example 11: define-cbr-as-cbv macro.**

The next level of macro sophistication in Scheme is the *syntax-case* macro system. A *syntax-case* macro can perform arbitrary computation (using `Scheme`) on the results of a pattern match. For example, a *syntax-case* macro can distinguish call-by-reference calls that have variable arguments from those that do not, and it can produce a different expansion in each case.

A *syntax-case* macro does not manipulate plain text, or even plain symbols and lists (as does a Lisp macro). Instead, the macro manipulates syntax objects, which encode the lexical context and source locations of program fragments. Consequently, just like a `syntax-rules` macro, a *syntax-case* macro respects the lexical structure of the source program (by default) and it plays well with source-correlating tools.

Syntax objects enable the implementation of most any little language or language extension. Using *syntax-case*, we have implemented a Java-like class system for Scheme, lex- and yacc-like forms for building parsers, and constructs for defining and linking program components. Programmers using these constructs do not reason about them in terms of their expansion. Instead, syntax objects allow the expansion to be hidden behind abstract definitions of the constructs, just as the inner workings of any compiler are hidden behind a language definition.

A language's concrete syntax need not be parenthesized to make use of syntax objects. In particular, we are currently developing implementations of Java and ML for our programming environment, DrScheme (see "Fostering Little Languages," *DDJ*, March 2004). Since our parsing tools produce syntax objects, we can treat Java and ML like macro extensions of Scheme. These macro implementations resemble typical Java-to-Scheme and ML-to-Scheme compilers, but little additional work is needed to adapt our entire programming environment to new languages.

**Beyond Patterns**

With the definitions of the `define-get/set-var`, `define-cbr`, and `define-cbr-as-cbv` macros, we have added call-by-reference functions to Scheme, using only 33 lines of pattern-based macro definitions. For many purposes, the implementation is complete, though it suffers some obvious limitations:

- Call-by-value functions in Scheme are higher order, which means that they can be passed to other functions or returned from other functions. Our call-by-reference functions are first-order, which means they can only be used in a direct call.
- When arguments are not variables, as in `(f 1 2)`, the result is a `set!` syntax error. In most languages with call-by-reference functions, nonvariable arguments are effectively placed into a fresh dummy variable at the call site.

These problems cannot be solved using only pattern-matching macros in Scheme, and many other kinds of language extensions fall outside the capabilities of pattern-matching macros. Nevertheless, the ideas behind Scheme's pattern-matching macros provide a solid foundation for more elaborate macro systems.

**http://www.ddj.com Dr. Dobb's Journal, April 2004**