The Scheme of Things: Implementing Lexically Scoped Macros

Jonathan Rees
Massachusetts Institute of Technology
jar@ai.mit.edu

I have been hearing some complaints that Scheme's new lexically scoped macro facility is difficult to implement. There are two components to the proposal: the pattern language and lexical scoping. The two pose independent problems. I agree that the ellipsis-enriched pattern language can be tricky to implement; implementations that I have seen take anywhere from 250 to 1400 lines of Scheme code. However, I believe that it is conceptually straightforward, and several implementations have been around for many years (see [9]). On the other hand, many people are unnecessarily getting tripped up on lexical scoping, which, unlike the pattern matcher, is very simple to implement.

(The Scheme report authors have not agreed on any low-level macro defining facility. The appendix to the Revised^4 Report [4] describes a sample low-level facility, but even that facility's own authors have repudiated it. There are three other proposals for low-level facilities on the table [7, 3, 6], and debate and experimentation continue.)

When the interpreter or compiler comes across a macro application, it invokes a *transcription function* that computes a form that replaces the macro application. For macros defined using **syntax-rules**, the transcription function is responsible for matching the macro application against the available patterns, and building a replacement according to the appropriate template. For example:

```
(define-syntax test
  (syntax-rules ()
    ((test ?thing ?proc ?else)
     (let ((temp ?thing))
       (if temp (?proc temp) ?else)))))
```

(The use of question marks is just a naming convention for meta-variables, not part of their syntax.) Lexical scoping requires this macro to work even when `if` or `let` is lambda-bound or when `temp` occurs in the `?proc` or `?else` expressions. (With the Revised^4 Report's macro facility, there are no reserved words, so one may now bind any identifier, including those such as `if` that are initially syntactic keywords.) Consider the following somewhat contrived expression:

```
(test (read if)
      (lambda (form)
        (write (list form temp) of))
      (newline of))
```

The variable `if` presumably holds an input file port, and `temp` might be the current temperature. A naive transcription function would rewrite this expression as

```
(let ((temp (read if)))
  (if temp
      ((lambda (form) (write (list form temp) of))
       temp)
      (newline of))).
```

The naive transcription suffers from both kinds of capture problems: the macro's **temp** collides with the macro user's **temp**, and the macro's **if** collides with the macro user's **if**. In order to implement lexical scoping, constant pieces of the output that come from the macro template must be distinguished from parts that come from the input form. We have at least two choices here:

1. Represent the input form using ordinary Scheme symbols and pairs, and have the transcription function use some different data types in order to distinguish the text that it adds. Then, arrange for the system that processes the result to interpret the special types appropriately.

2. Represent the input form in some non-standard way, with symbols or pairs replaced by some different data types, and allow the transcription function to embed the unusual structure in new text that is represented using ordinary symbols and pairs. Then, have the system that delivers the input to the transcription function make sure that the input has the proper non-standard representation.

Nothing says that we even have to use symbols and pairs in the interpreter at all, so there is also the third option of combining the first two options.

Option 1, using ordinary pairs but "unusual" identifiers for added text, corresponds to the algorithm described in [5]. Option 2, with ordinary pairs and unusual identifiers, corresponds to Kohlbecker's algorithm [8]. Option 2 also more or less encompasses syntactic closures [1], which use ordinary symbols but wraps them inside unusual surrounding structure.

The implementation I will describe is a version of option 1. It is basically a transcription into Scheme of the algorithm presented in [5].

A transcription function accepts and returns S-expressions in the usual way. However, instead of inserting symbols into the output, as a naive transcription function (one that does not respect lexical scoping) would have done, it inserts *generated names*, which I will write using brackets, e.g. `[temp 13]`. The number is a unique tag that distinguishes the generated names introduced by this macro expansion from those introduced by other macro expansions. There is no reason that the tag has to be a number, as long as it is unique to one particular expansion. Also, we could use a distinct tag for each name, but it suffices to use the same tag for all names generated for a single macro expansion.

For the above example, the transcription function might return

```
([let 13] (([temp 13] (read if)))
  ([if 13] [temp 13]
           ((lambda (form) (write (list form temp) of))
            [temp 13])
           (newline of)))
```

Now the crucial step: We interpret (or compile) the output in a special kind of lexical environment. Suppose that *tag* is the unique tag generated for an expansion. The new lexical environment has the property that a generated name of the form [*name tag*] has the same binding that the name *name* does in the environment in which the macro was defined. All other names are interpreted just as they are in the environment where the macro application occurs. In the example, `test` was defined at top level, so `[let 13]` and `[if 13]` get the top-level environment's bindings of `let` and `if`, not the bindings of `let` and `if` in the environment in which the `(test ...)` form occurs. The name `[temp 13]` is initially bound to whatever `temp` is bound to at top level, or perhaps it is unbound. `[temp 13]` is not used free in the expansion, however, but only occurs in the `let` body, where it is bound to the variable that gets the result of `(read if)`.

Here we see both functions of generated names. They can act as reliable references to top-level variables, or they can act as "gensyms" or nonconflicting names for temporary quantities.

Figure 1 is the relevant portion of a simple Scheme compiler that implements this mechanism. The figure includes all the code necessary to support lexically scoped macros, except for details such as data structure definitions. The output of this compiler might be machine code, byte codes, some kind of tree-structured intermediate code ("S-code"), or even Scheme.

Figure 1 assumes that macros are represented as ⟨*transcribe*, *environment*⟩ pairs. Following [3], *transcribe* is a procedure of three arguments: the expression to be expanded, a name generation procedure that maps symbols to generated names, and a comparison procedure for recognizing auxiliary keywords (such as `else` in `cond`). The *environment* is the environment in which the macro was defined; for macros defined with `define-syntax`, this will be the top-level environment. Environments map names (symbols or generated names) to "denotations," where a denotation is either a token designating one of the special operators (`lambda`, `quote`, etc.), a macro, or a compile-time representation of a bound or top-level variable.

A final detail: In order to support macros that introduce fixed quoted symbols into their expansions, the compilation routine for `quote` must replace generated names with their underlying symbols. An example of a macro for which this matters is

```
(define-syntax cell
  (syntax-rules ()
    ((cell ?x)
     (list 'cell
           (lambda () ?x)
           (lambda (new) (set! ?x new))))))
```

The output of the transcription function will be something like

```
([list 17] ([quote 17] [cell 17]) ...)
```

and we need to make sure that the expression `([quote 17] [cell 17])` delivers the symbol `cell`, not a generated name.

* * *

In the particular case where the compiler's output is Scheme, the "compiler" is a Scheme-to-Scheme macro expander of the sort found in several papers on

```
(define (compile exp env)
  (cond ((name? exp) (compile-variable exp env))
        ((pair? exp)
         (if (name? (car exp))
             (let ((den (binding env (car exp))))
               (cond ((special? den)
                      (compile-special-form den exp env))
                     ((macro? den)
                      (compile-macro-application den exp env))
                     (else
                      (compile-application exp env))))
             (compile-application exp env)))
        ((literal? exp) (compile-constant exp))
        (else (syntax-error "invalid expression" exp))))

(define (compile-macro-application mac exp env-of-use)
  (let* ((uid (generate-unique-id))
         (new-exp (transcribe mac exp env-of-use uid)))
    (compile new-exp (bind-aliases uid mac env-of-use))))

(define (transcribe mac exp env-of-use uid)
  (let* ((env-for-expansion (bind-aliases uid mac env-of-use))
         (rename (lambda (name)
                   (generate name uid)))
         (compare (comparison-procedure env-for-expansion)))
    ((macro-transcribeer mac) exp rename compare)))
```

; Create an environment suitable for processing a macro expansion.

```
(define (bind-aliases uid mac env-of-use)
  (let ((env-of-definition (macro-env-of-definition mac)))
    (lambda (name)
      (if (and (generated? name)
               (eqv? (generated-uid name) uid))
          (env-of-definition (generated-name name))
          (env-of-use name)))))
```

; Environments are procedures that map names to denotations.

```
(define (binding env name)
  (env name))

(define (generate-unique-id)
  (let ((uid *unique-id*))
    (set! *unique-id* (+ *unique-id* 1))
    uid))

(define *unique-id* 0)

(define (name? x)
  (or (symbol? x) (generated? x)))
```

Figure 1: Expanding macros

the subject. That macro expansion algorithms are often presented as recursive source-to-source preprocessors has obscured the fact that it is not necessary to fully expand the entire program applications before compilation or interpretation can proceed. Macro expansion can be easily performed concurrently with compilation or interpretation, even when macros respect lexical scoping.

Three implementations of lexically scoped macros and the **`syntax-rules`** pattern language are available on the Internet in directory **`pub/scheme/doc/`** on **`nexus.yorku.ca`**:

- **`simple-macros.tar.Z`** — the implementation from which Figure 1 was extracted.
- **`syntax-case.tar.Z`** — Kent Dybvig's system, as described in [6].
- **`synclo.tar.Z`** — Chris Hanson's implementation based on syntactic closures, as described in [7].

* * *

Thanks to Brian Reistad for comments and corrections.

References

[1] Alan Bawden and Jonathan Rees. Syntactic closures. *1988 ACM Conference on Lisp and Functional Programming,* pages 86–95.

[2] William Clinger. Macros in Scheme. *Lisp Pointers* IV(4): 17–23, October–December 1991.

[3] William Clinger. Hygienic macros through explicit renaming. *Lisp Pointers* IV(4): 25–28, October–December 1991.

[4] William Clinger and Jonathan Rees (editors). The revised^4 report on the algorithmic language Scheme. *Lisp Pointers* IV(3): 1–55, July–September 1991.

[5] William Clinger and Jonathan Rees. Macros that work. *1991 ACM Symposium on Principles of Programming Languages,* pages 155–162.

[6] R. Kent Dybvig. Writing macros in Scheme with syntax-case. Indiana University Computer Science Department technical report #356, June 1992.

[7] Chris Hanson. A syntactic closures macro facility. *Lisp Pointers* IV(4): 9–16, October–December 1991.

[8] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. *1986 ACM Conference on Lisp and Functional Programming,* pages 151-159.

[9] Eugene E. Kohlbecker and Mitchell Wand. Macro-by-example: Deriving syntactic transformations and their specifications. *1987 ACM Symposium on Principles of Programming Languages,* pages 77–84.