A Few Principles of Macro Design

David Herman
Northeastern University
dherman@ccs.neu.edu

David Van Horn
Brandeis University
dvanhorn@cs.brandeis.edu

Abstract

Hygiene facilitates the implementation of reliable macros but does not guarantee it. In this note we review the introspective capabilities of macros, discuss the problems caused by abusing this power, and suggest a few principles for designing well-behaved macros.

1. Introduction

Hygiene makes it easier to write reliable syntactic abstractions, but does not guarantee it. The power of macros lies in the ability to define syntactic abstractions, i.e., derived special forms that extend Scheme as if they were a part of the language itself. In form and function, hygienic macros achieve this spectacularly well. But for reasoning about programs, they still fall short of this goal.

Hygiene is often motivated by reference to variable conventions and α-equivalence [12, 1, 4], but these concepts are really only understood as properties of fully-expanded terms, with only the pre-defined forms of Scheme and no remaining macros. Indeed, the roles of identifiers as bindings, references, or quoted symbols are only discovered incrementally during the process of expansion as macro uses are eliminated. Effectively, α-equivalence has no meaning in Scheme until after expansion.

Meanwhile, lexical scope is a practical language design principle because it allows programmers to understand the local definitions of a program by simple inspection. Indeed, as the R^{6}RS describes, "Scheme is a statically scoped programming language. Each use of a variable is associated with a lexically apparent binding of that variable" [16].

Since macros may manipulate programs arbitrarily, it is not necessarily possible to understand the binding structure of a program without first fully expanding its macros. Often, automated tools such as the `expand` procedure or graphical IDE tools such as DrScheme's "Check Syntax" button [5] fulfill this need. But such tools either require inspecting fully-expanded code, revealing the complete implementation details of their macros, or at least preclude simple syntactic inspection of the source program. To achieve true syntactic abstraction, programmers should be able to use and understand macros as if they were built-in forms in Scheme. Specifically, users should be able to understand a macro's scoping discipline without reference to its implementation or expansion.

A Programming Puzzle

Albert Meyer posed thirteen provocative programming puzzles to show that, when it comes to reasoning about programs, "intuition has its limits" [13]. Let us reconsider one of Meyer's puzzles, tailored for Scheme:

PUZZLE *Exhibit a simple context into which any two Scheme programs can be substituted, such that if the Scheme programs are not identical S-expressions, the resulting programs yield different results.*

The answer is simple: `(quote □)` distinguishes any two Scheme programs syntactically by reifying their source as runtime values.

Of course, it is also possible for a macro to quote its argument, so another solution to the puzzle is:

```
(let-syntax ((q (syntax-rules ()
                  ((q x) (quote x)))))
  (q □))
```

In fact, any Scheme context $C[$`(m □)`$]$ involving the application of a macro `m` might quote its argument. Given that macros are Turing-complete, the question of whether `m` quotes its argument is undecidable.

But `quote` is not the only way a Scheme program can inspect its own source code. Whereas `quote` turns source text into data that a program can consume at runtime, macros written with `syntax-rules` can actually make compile-time decisions based on program source text. In Section 2, we review several known techniques exploiting this introspective power to demonstrate pathological hygienic macros: one that distinguishes all syntactic differences between arbitrary Scheme programs, and a classic `syntax-rules` macro that behaves like an unhygienic macro.

Such dramatic exploitation of syntactic introspection is probably rarely needed. And in Section 3, we show that the power of Scheme macros comes at a price: the application of any macro may eliminate all Scheme program equivalences in its arguments. In practice, though, well-behaved macros seem to be usable as true abstractions, i.e., without exposing their implementations. In Section 4, we describe a few simple design principles for writing macros that act as consistent extensions of Scheme, so that clients can treat them as if they were built-in language forms.

2. The Introspective Power of Macros

In this section, we review the introspective power of `syntax-rules` and use several known tricks to demonstrate a solution to Meyer's puzzle that distinguishes two programs syntactically at macro-expansion time.

2.1 A Compile-Time Predicate

Figure 1 gives a definition of a macro `sexp=?`, which serves as a compile-time predicate. Because of the non-strict nature of macro expansion, predicates are written in continuation-passing style,

with two additional arguments for the respective success and failure continuations. Thus `sexp=?` takes four arguments: two terms to compare as S-expressions and two continuation terms.

The definition of `sexp=?` decomposes its arguments structurally, comparing the contents of pairs and vectors componentwise. In the final clause, `sexp=?` uses an auxiliary predicate `ident?` to test whether the arguments are identifiers or constants, as well as comparison predicates `ident=?` and `const=?` to compare them.

The predicate `ident?` is due to Kiselyov [11]. The implementation defines a derived macro `test`, which cleverly expands the term `a` into the pattern of `test`. If `a` is bound to an identifier, then applying `test` to an identifier will match its first rule and expand to the success continuation. Otherwise, applying `test` to an identifier will not match the first rule, since a compound pattern never matches an identifier; in this case the second rule will apply and `test` will expand to the failure continuation. Thus by applying `test` to the identifier `*anident*`, `ident?` serves as a predicate testing whether its argument `a` is bound to an identifier.

The predicates `ident=?` and `const=?`, due to Petrofsky and Kiselyov [14, 10], compare two terms which are assumed to both be identifiers or constants, respectively. Like the macro `ident?` described above, `ident=?` defines a nested macro `test`, but ingeniously also expands its argument `a` into the literals list of `test`. Thus the first rule of the `test` macro compares the identifier bound to `a` as a syntactic keyword rather than an ordinary pattern variable. By applying `test` to the argument `b`, the two identifiers are effectively compared for syntactic equality.[1] The implementation of `const=?` is similar but does not require the use of the literals list to match constants.

2.2 Seemingly Unhygienic Macros with `syntax-rules`

The *hygiene conditions* [12, 1] for macro expansion are described informally as properties of variable occurrences:

1. *It is impossible to write a macro that introduces a binding that can capture references other than those introduced by the macro.*
2. *It is impossible to write a macro that introduces a reference that can be captured by bindings other than those introduced by the macro.*

A macro that violates the first hygiene condition, for example, must "introduce" a binding—i.e., an identifier that does not occur in the macro's input—that captures a variable reference from its input.

Petrofsky [14] and Kiselyov [10] described a clever technique for implementing macros with `syntax-rules` that appear to violate the hygiene conditions. For example, a `loop` macro that implicitly binds the variable `break` to a captured continuation for escaping the loop might appear to be unhygienic:

```
(loop (set! x (+ x 1))
      (display x)
      (if (>= x 100)
          (break #f)))
```

However, even a hygienic macro can *search for a reference* in its input and *copy* that identifier into a binding position. The implementation of `loop` appears in Figure 2. The auxiliary macro `find` searches for an identifier in a term, inserting the found identifier into a success continuation term or else expanding into a failure continuation term. The `loop` macro uses `find` to extract the identifier `break` from the loop body and bind it as the argument to `call/cc`, or else introduces a variable `dummy` if `break` does not occur in the body.

The intuition that `loop` is unhygienic is based on the assumption that the identifier `break` does not occur in the input to `loop`, so its binding must therefore have been introduced by the macro. But this assumption is false since `loop` is able to find references to `break` in the body expression. Thus `loop` does not introduce the binding for `break` at all; it merely copies an occurrence into both binding and reference positions.

[1] More precisely, the identifiers are compared as if with the procedure `free-identifier=?` [4, 16].

```
(define-syntax sexp=?
  (syntax-rules ()
    ((sexp=? (a1 . b1) (a2 . b2) yes no)
     (sexp=? a1 a2 (sexp=? b1 b2 yes no) no))
    ((sexp=? (a1 . b1) e2 yes no)
     no)
    ((sexp=? e1 (a2 . b2) yes no)
     no)
    ((sexp=? #(e1 ...) #(e2 ...) yes no)
     (sexp=? (e1 ...) (e2 ...) yes no))
    ((sexp=? #(e1 ...) e2 yes no)
     no)
    ((sexp=? e1 #(e2 ...) yes no)
     no)
    ((sexp=? e1 e2 yes no)
     (ident? e1
       (ident? e2 (ident=? e1 e2 yes no) no)
       (ident? e2 no (const=? e1 e2 yes no))))))

(define-syntax ident?
  (syntax-rules ()
    ((ident? a yes no)
     (let-syntax ((test (syntax-rules ()
                          ((test a y n) y)
                          ((test _ y n) n))))
       (test *anident* yes no)))))

(define-syntax ident=?
  (syntax-rules ()
    ((ident=? a b yes no)
     (let-syntax ((test (syntax-rules (a)
                          ((test a y n) y)
                          ((test x t n) n))))
       (test b yes no)))))

(define-syntax const=?
  (syntax-rules ()
    ((const=? a b yes no)
     (let-syntax ((test (syntax-rules ()
                          ((test a y n) y)
                          ((test _ y n) n))))
       (test a b yes no)))))
```

Figure 1. A compile-time solution to Meyer's puzzle.

3. Hygiene is Not a Cure-All

As the above examples demonstrate, even with the rather simple hygienic macro system of `syntax-rules`, it is still possible to write rather ill-conceived macros. The question then is how to use hygienic macros effectively, and how better to understand programs that use them.

```
(define-syntax find
  (syntax-rules ()
    ((find ident (a . b) sk fk)
     (find ident a sk (find ident b sk fk)))
    ((find ident #(a ...) sk fk)
     (find ident (a ...) sk fk))
    ((find ident a (sk-op . sk-args) fk)
     (ident? a
       (ident=? ident a (sk-op a . sk-args) fk)
       fk))))

(define-syntax loop
  (syntax-rules ()
    ((loop e)
     (let-syntax ((k (syntax-rules ()
                       ((_ ident e*)
                        (call/cc (lambda (ident)
                                   (let f ()
                                     e*
                                     (f))))))))
       (find break e (k e) (k dummy e))))
    ((loop es ...)
     (loop (begin es ...)))))
```

Figure 2. Petrofsky's `loop` macro.

3.1 Is the Theory of Macros Trivial?

The old Lisp programming construct of *fexprs*, i.e., functions that dynamically receive their arguments as source code, is widely deprecated because it allows programs to distinguish essentially all syntactic differences between programs [15]. In effect, the operator in any procedure application might turn out to be a fexpr, which could dynamically distinguish any syntactic transformation within its arguments. This causes problems for compilers, which cannot then exploit standard program equivalences to perform optimizations, as well as programmers, because their programs become sensitive to even the slightest changes.

Wand [17] formalized this idea in a lambda calculus with fexprs by proving that for any program p, we can construct a context that distinguishes any $p' \neq_\alpha p$.[2] With the definitions in Figure 1, we can similarly construct a context for a Scheme program p that distinguishes all other programs at macro-expansion time:

`(sexp=?` p $\square$ `#t #f)`

In the extreme, this implies that the use of an untrusted macro in Scheme can defeat any program equivalences we might expect to hold for the arguments passed to the macro. Does this mean that hygienic macros are as problematic and difficult to reason about as fexprs?

Certainly not! In practice, hygienic macros do not prove to be as troublesome as fexprs, whether for compilers or for programmers. For the former, the reason is simple: whereas fexprs can inspect source code dynamically, macros are restricted to performing this introspection at compile-time. The compiler only needs to operate on a program (or module [6]) after expansion, at which point it can safely perform optimizations. For programmers, however, working with a fully-expanded program would be impractical. But well-written macros serve as *syntactic abstractions*, allowing clients to use them without inspecting their expansion.

[2] In Wand's simple model, fexprs cannot distinguish α-conversions, although in real implementations even variable names are observable.

3.2 Alpha, the Ultimate Refactoring

A well-written syntactic abstraction should behave like a consistent extension of Scheme. Unlike `sexp=?`, which interferes with almost all program equivalences, a macro should not subvert the usual semantic properties of Scheme. Program equivalences are especially important because they facilitate program *refactoring* [7], i.e., improving internal characteristics of a program without changing its external behavior.

A particularly important program equivalence for Scheme is α-equivalence. Indeed, hygiene is often motivated by reference to α-conversion and variable conventions [12, 1, 4]. Now, the only known notion of α-equivalence for Scheme involves fully expanding programs and comparing the results. Automated tools such as the α-renaming feature of the DrScheme IDE [5] must internally expand a program to determine the scoping relationships. But human readers should certainly not have to inspect the output of expansion to understand a program.

Consider the standard `let` form: the Scheme standard allows for the possibility of `let` to be implemented as a macro, but programmers can think of it as a built-in form. Moreover, we can think of Scheme's core notion of α-equivalence over the built-in forms as if it were extended with a rule:

$$\frac{e_1 =_\alpha e_1' \qquad z \text{ fresh} \qquad e_2[z/x] =_\alpha e_2'[z/x']}{\texttt{(let ((}x\ e_1\texttt{))}\ e_2\texttt{)} =_\alpha \texttt{(let ((}x'\ e_1'\texttt{))}\ e_2'\texttt{)}}$$

Because `let` is well-behaved, programmers understand this rule intuitively and act as though it is a primitive form, regardless of whether their Scheme implementation implements `let` as a macro.

4. Principles of Macro Design

By foregoing some of the introspective power of macros, macro writers can allow their clients to use macros as if they were built-in forms in Scheme. In this section, we catalog several design principles for writing well-behaved macros and demonstrate how macros like `sexp=?` and `loop` violate these principles.

Just as many hygienic macro systems are designed to allow for the occasional use of intentional variable capture, we acknowledge that our design rules are not always appropriate and can sometimes be disregarded to good effect. For this reason we also discuss some reasonable exceptions to our rules.

4.1 Design with α-equivalence in mind.

For clients of the `let` macro to intuit the α-equivalence rule described in the previous section, `let` needs to obey a documented scoping discipline. Section 11.4.6 of the R^{6}RS explains this discipline in prose, which we could summarize in an informal *binding specification*:

```
;; (let ((x:ident expr) ...) expr[x ...]) :: expr
```

This specification indicates the general *shape* of the macro [2], such as the parenthesization of the individual clauses, as well as the binding structure. The name `x` is used as a placeholder to refer to the identifiers bound in the clauses, to show that these variables are added to the lexical environment of the body expression.

Documenting the scoping discipline of a macro helps impose structure on its design and conveys the information clients need in order to know how to α-convert uses of the macro. Despite the lack of a general α-equivalence relation for Scheme programs with macros, macros that obey regular scoping disciplines can still be α-converted in a natural way.

The remaining design principles in this section can be seen as ways of designing macros with α-equivalence in mind.

4.2 Use identifiers consistently.

The hygiene conditions refer to "bindings" and "references," which are only discovered incrementally during the process of macro expansion; until macros are expanded into primitive Scheme forms such as `lambda`, the binding structure of a term is unknown. In general, an identifier is only known to be a binding or a reference (or in fact a quoted symbol) by completely expanding the program and relating the identifier in the unexpanded program to its corresponding identifier in the expanded program.

But such a mapping may not be one-to-one; macros are not required to use every argument exactly once. It is not hard to construct macros that copy an input identifier into several different contexts:

```
;; (dup x:ident expr[x]) :: expr
(define-syntax dup
  (syntax-rules ()
    ((dup a e)
     (begin
       (set! a 42)
       (lambda (a) e)))))
```

The same variable `a` is used both as a free variable to be updated to 42 and a bound name within the subexpression `e`. So in an application such as `(dup x x)`, different choices of names for `x` change the meaning of the expression. This does not follow the usual structure of Scheme binding constructs: since the first occurrence of `x` is both free and bound, its inner binding cannot be α-converted independently from its surrounding binding.

Sometimes it may be a useful shorthand to use a single identifier both as a bound variable and as a free reference. For example, an object-oriented library might provide a syntax for inheriting a field in a subclass with a single identifier simultaneously specifying the name of the inherited variable and binding the variable locally:

```
;; (inherit x:ident) :: defn[x]
(inherit foo)
```

Here `foo` serves both as a reference and a binding occurrence. But our design principle also suggests an alternative, more flexible syntax that allows the two uses of `foo` to operate independently:

```
;; (inherit ident as x:ident) :: defn[x]
(inherit foo as bar)
```

In this revised syntax, the first variable serves as a reference to the field being inherited, whereas the second variable is the local binding. This allows for the possibility of α-renaming the local binding independently from the name of the inherited field. The former syntax is useful for common cases, but the latter syntax is more amenable to α-conversion.

4.3 Avoid observing variable names.

In addition to copying an identifier into both free and binding positions, quoting a bound identifier can be problematic. For example, we could devise an alternative to `lambda` that disallows the specific variable name `quux`:

```
;; (lambda* (x:ident ... . rest:ident)
;;    expr[x ... rest]) :: expr
(define-syntax lambda*
  (syntax-rules ()
    ((lambda* (x ... . rest) b ...)
     (if (memq 'quux '(x ... rest))
         (error 'lambda* "bad name: quux")
         (lambda (x ... . rest) b ...)))))
```

By inspecting the bound variables and disallowing a particular name, `lambda*` restricts the usual freedom of the programmer to α-convert a bound variable to any name.

More subtly, `syntax-rules` provides another means for observing names through the literals list [16], which allows a macro to match against particular identifiers in a pattern. The `ident=?` macro of Section 2 uses the literals list for just this purpose. While the intention of the literals list is to identify certain identifiers as syntactic keywords, there is nothing preventing its use as a general mechanism for comparing identifiers as with `ident=?`. In other words, `syntax-rules` provides enough introspective power to compare identifiers at compile-time just as `quote` and `eq?` can do at runtime.

Observing variable names can sometimes be useful for introspective facilities like IDE's and automated documentation tools. A Scheme with a "docstring" facility like Emacs Lisp's might provide a form `define/doc` for associating a documentation string with a variable definition:

```
;; (define/doc x:ident expr expr) :: defn[x]
(define-syntax define/doc
  (syntax-rules ()
    ((define/doc x doc e)
     (begin
       (define x e)
       (set-doc! x (format "~a: ~a" 'x doc))))))
```

If the library exposes this documentation string, then the program could discover the variable name and become sensitive to α-conversions. In practice, though, programmers adhere to conventions that docstrings are strictly for informative purposes, such as displaying to a user, and should not be relied on for the internal logic of programs.

4.4 Treat subterms as atomic.

Ganz et al [8] coined the term *analytic macros* to describe macros that inspect the internal structure of their subterms. This often occurs when macro-writers are tempted to optimize their macros for special cases of their input. For example, a logical `or` operator usually needs to bind the result of its first expression to a temporary variable in order to prevent evaluating the expression twice, but if the expression is itself already a variable, the additional temporary variable is superfluous. A macro-writer might be tempted to optimize the `or` macro by optimizing this special case:

```
;; (or expr expr) :: expr
(define-syntax or
  (syntax-rules ()
    ((or e1 e2)
     (ident? e1
       (if e1 e1 e2)
       (let ((tmp e1))
         (if tmp tmp e2))))))
```

In the "macro-writer's bill of rights" [3], Dybvig explains that these are the kinds of optimizations that a compiler can make perfectly well all by itself. Useless variable elimination is an easy and common compiler optimization. Dybvig argues that compilers should alleviate the incentive for hand-writing such optimizations by performing them automatically. The simpler and clearer implementation of `or` that the programmer ought to write is:

```
;; (or expr expr) :: expr
(define-syntax or
  (syntax-rules ()
    ((or e1 e2)
     (let ((tmp e1))
       (if tmp tmp e2)))))
```

This simpler version of `or` is easier to read and maintain since it treats only the general case and leaves the delicate business of optimization to the compiler.

Analytic macros can violate information hiding by observing very fine-grained details about the implementation of an expression. The `sexp=?` macro of Section 2 uses this introspective power to compare terms for syntactic equality. And the `loop` macro combines the analytic `find` with duplicating the identifier `break` into both binding and reference positions to behave much like an unhygienic macro.

5. Discussion

It is an established fact, though perhaps not widely-enough understood, that even hygienic macros allow for extremely fine-grained syntactic introspection. Based on this observation, we have proposed some principles for designing macros that at least informally allow programmers to use α-conversion on unexpanded Scheme programs as a semantics-preserving transformation.

Herman and Wand [9] have studied hygienic macros in a restricted setting that enforces design principles through a static type system. The informal binding specifications we use here are described formally as types, which allows for a formal definition of α-equivalence of programs with macros independent of the expansion algorithm. Hygienic expansion is then shown to preserve the meaning of α-equivalent source programs.

In general, the flexibility of full Scheme seems to preclude the semantic properties (primarily, the definition and preservation of α-equivalence) provided in that setting. But more work is needed to study whether those properties extend to macros written according to such a discipline, when implemented in the context of full Scheme. Until then, we hope the informal principles described in this note shed some light on the design of reliable macros.

Acknowledgments

We are indebted to Will Clinger, Jacob Matthews, Sam Tobin-Hochstadt, Mitchell Wand, and the anonymous reviewers for their attentive readings and insightful comments.

References

[1] William Clinger and Jonathan Rees. Macros that work. In *Principles of Programming Languages (POPL)*, 1991.

[2] Ryan Culpepper and Matthias Felleisen. Taming macros. In *Generative Programming and Component Engineering (GPCE)*, October 2004.

[3] R. Kent Dybvig. The guaranteed optimization clause of the macro-writer's bill of rights, 2004. `http://video.google.com/videoplay?docid=-6899972066795135270`.

[4] R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326, December 1993.

[5] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: a programming environment for Scheme. *Journal of Functional Programming*, 12(2):159–182, 2002.

[6] Matthew Flatt. Composable and compilable macros: You want it when? In *International Conference on Functional Programming (ICFP'2002)*, 2002.

[7] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[8] Steven E. Ganz, Amr Sabry, and Walid Taha. Macros as multi-stage computations: type-safe, generative, binding macros in MacroML. In *ICFP 2001*, pages 74–85, 2001.

[9] David Herman and Mitchell Wand. A theory of hygienic macros. In *ESOP '08: Proceedings of the Seventeenth European Symposium On Programming*, March 2008.

[10] Oleg Kiselyov. How to write seemingly unhygienic and referentially opaque macros with syntax-rules. In *Workshop on Scheme and Functional Programming*, October 2002.

[11] Oleg Kiselyov. How to write symbol? with syntax-rules, September 2003. `http://okmij.org/ftp/Scheme/macro-symbol-p.txt`.

[12] Eugene E. Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce F. Duba. Hygienic macro expansion. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pages 151–161, 1986.

[13] Albert R. Meyer. Thirteen puzzles in programming logic. In D. Bjørner, editor, *Proceedings of the Workshop on Formal Software Development: Combining Specification Methods*. Lecture Notes in Computer Science, May 1984.

[14] Al* Petrofsky. How to write seemingly unhygienic macros using syntax-rules, November 2001. `http://groups.google.com/group/comp.lang.scheme/msg/5438d13dae4b9f71`.

[15] Kent M. Pitman. Special forms in Lisp. In *LFP '80: Proceedings of the 1980 ACM conference on LISP and functional programming*, pages 179–187, New York, NY, USA, 1980. ACM.

[16] Michael Sperber, R. Kent Dybvig, Matthew Flatt, and Anton van Straaten (Editors). Revised^6 report on the algorithmic language Scheme, 2007.

[17] Mitchell Wand. The theory of fexprs is trivial. *Lisp and Symbolic Computing*, 10(3):189–199, 1998.