Macros in Scheme

William Clinger

Although macros are even older than Lisp, Scheme has just become the first block-structured programming language to support a reliable macro system. This macro system is described in the appendix to the *Revised^4 Report on the Algorithmic Language Scheme* [1]. This technical report contains three more articles on the subject of macros in Scheme, counting the article you are reading. I wrote this as an introduction to the other two articles, because it is very hard to understand their purpose without some further background on macros in Scheme.

Macros are hardly new in the Lisp world, so I can explain what is new about the Scheme macro system by comparing it to Common Lisp's [5]. Consider a simple `push` macro that is like the `push` macro in Common Lisp except that the *place* being pushed is required to be a variable. In Scheme we can write this macro as

```
(define-syntax push
  (syntax-rules ()
    ((push item place)
     (set! place (cons item place)))))
```

and we can use `push` in an expression such as

```
(let* ((cons (lambda (name)
               (case name
                 ((phil)  '("three-card monte"))
                 ((dick)  '("secret plan to end the war"
                            "agnew"
                            "not a crook"))
                 ((jimmy) '("why not the best"))
                 ((ron)   '("abolish the draft"
                            "balance the budget"))
                 (else    '()))))
       (scams (cons 'phil)))
  (push (car (cons 'jimmy)) scams)
  (push (cadr (cons 'ron)) scams)
  scams)
```

This expression will evaluate to

```
("balance the budget" "why not the best" "three-card monte")
```

That, at least, is what this expression will evaluate to in Scheme.

We're not supposed to write expressions like this in Common Lisp, according to `LISP-SYMBOL-REDEFINITION:MAR89-X3J13`. (The "consequences are undefined"). One reason for this decision by X3J13 is that the Common Lisp macro system is unreliable, so a lexical binding of `cons` as in the above example can break macros such as `push`. The decision by X3J13 doesn't really solve the problem, because it doesn't address macros that refer to global functions written by the user of a Common Lisp system, but it does make Common Lisp significantly less unreliable in practice.

Recognizing that `LISP-SYMBOL-REDEFINITION:MAR89-X3J13` doesn't completely solve the problem, let's push forward with our example as though X3J13 didn't exist. (Otherwise I would have to construct a different example!)

If we translate everything to pre-X3J13 Common Lisp we will get an error, because in Common Lisp the first use of `push` would expand into

```
(setq scams (cons (cons 'jimmy) scams))
```

within a context where `cons` is bound to a one-argument procedure. There are various ways one might try to work around this problem in Common Lisp, but none are fully general. If a macro needs to refer to a global variable or function (other than those predefined in the `common-lisp` package, which X3J13 has effectively made into reserved words), then it is quite impossible to write that macro reliably using the Common Lisp macro system.

I'm as willing as anyone to pick on Common Lisp, but that's not what I'm doing here. When Common Lisp was being designed Scheme had no macro system at all, for this very reason: Macros were inherently unreliable, or so we thought then. Even today, except for Scheme and a couple of research languages that employ a Scheme-style macro system, Common Lisp probably has the most sophisticated macros of any programming language. The fact is that macros have been unreliable in block-structured programming languages for thirty years, and that unreliability has just been accepted as the state of the art.

How does Scheme's new macro system manage to avoid this problem? In effect, the macro expander systematically renames local variables to avoid all inadvertent captures of bound variables, so the macro-expanded form of the expression above will be something like

```
(let* ((cons.1 (lambda (name)
                 (case name
                   ((phil)  '("three-card monte"))
                   ((dick)  '("secret plan to end the war"
                              "agnew"
                              "not a crook"))
                   ((jimmy) '("why not the best"))
                   ((ron)   '("abolish the draft"
```

```
                               "balance the budget"))
                  (else    '()))))
      (scams (cons.1 'phil)))
  (set! scams (cons (car (cons.1 'jimmy)) scams))
  (set! scams (cons (cadr (cons.1 'ron)) scams))
  scams)
```

A macro system that avoids inadvertent captures through systematic renaming is said to be *hygienic*. Eugene Kohlbecker developed and implemented the first hygienic macro system for his 1986 PhD dissertation [4]. In 1988 the Scheme community asked a four-person committee to design a hygienic macro system for inclusion in the Revised^4 Report. In 1990 Jonathan Rees and I developed a more general and efficient form of Kohlbecker's algorithm, which Jonathan used to implement a prototype of the Scheme macro system [2].

The Revised^4 Report specifies the syntax and semantics of a hygienic macro system, but does not prescribe any particular algorithm. The Revised^4 Report *does* describe a non-hygienic, low-level macro facility that *could* be used to implement the hygienic macro system I have been describing, but goes on to note that this particular low-level facility "is but one of several low-level facilities that have been designed and implemented to complement" Scheme's hygienic macro system. The following articles describe two more of them.

Before moving on to the low-level facilities, I want to show you more of the power of Scheme's hygienic macro system. First let us consider a simplified form of Common Lisp's `setf` macro. With the Scheme macro system there are no reserved words, so we can redefine `set!` locally as in

```
(let-syntax
  ((set! (syntax-rules (car cdr vector-ref)
           ((set! (car x) y)          (set-car! x y))
           ((set! (cdr x) y)          (set-cdr! x y))
           ((set! (vector-ref x e) y) (vector-set! x e y))
           ((set! x y)                (set! x y)))))
  (let* ((days (list 'monday 'wednesday 'friday))
         (day1 'sunday))
    (set! (car days) 'tuesday)
    (set! day1 (car days))
    day1))
```

The `(car cdr vector-ref)` following `syntax-rules` means that `car`, `cdr`, and `vector-ref` are not pattern variables. They can match only themselves. The use of `let-syntax` instead of `letrec-syntax` means that the `let-syntax` macro is not recursive, so it can refer to the outer definition of `set!` in the last rule without circularity.

In Common Lisp, `(setf (car days) 'tuesday)` will not be expanded if it occurs within a local binding for `car`. (At least that is so according to

FUNCTION-NAME:SMALL. On the other hand, the consequences are supposed to be undefined if car is bound locally. Perhaps some Common Lisp wizard can explain this to me.) The purpose of this rule in Common Lisp is to ensure that the scope of a setf method that is associated with a function does not exceed the scope of the function itself.

This limitation of the scope of a set! method will be enforced automatically by the Scheme macro system, because the car in the pattern will match a car in a use only if the two occurrences of car are within the scope of the same binding of car. Since this scope is lexical (whether local or global), the Scheme macro system generalizes Common Lisp's rule for setf in the way that is appropriate for block-structured languages like Scheme.

Of course, none of this is special to setf. Scheme's hygienic macro system is a general mechanism for defining syntactic transformations that reliably obey the rules of lexical scope.

Another (rather silly) example might help to make the point:

```
(let ((car cdr)
      (set-car! set-cdr!)
      (cdr car)
      (set-cdr! set-car!))
  (let-syntax
    ((set! (syntax-rules (car cdr vector-ref)
             ((set! (car x) y)          (set-car! x y))
             ((set! (cdr x) y)          (set-cdr! x y))
             ((set! (vector-ref x e) y) (vector-set! x e y))
             ((set! x y)                (set! x y)))))
    (let ((days (list 'monday 'wednesday 'friday))
          (set-car! (lambda () 17)))
      (set! (car days) 'tuesday)
      (cons (set-car!) days))))
```

In Scheme this evaluates to (17 monday . tuesday). This is easy to see if you forget everything you thought you knew about macros, and rely only on the fact that all (non-pattern-variable) identifiers in the macro definition, whether they occur on the left or on the right hand side of a syntax rule, are resolved using Scheme's familiar lexical scope rules.

Alas, I cannot easily implement an analogue of Common Lisp's setf in Scheme using the high-level macro system. Although local macros can be either recursive (letrec-syntax) or non-recursive (let-syntax), global macros are always recursive (define-syntax). This should be fixed.

Another problem with macros in Scheme is that some macros are awkward or impossible to describe using the pattern language. Suppose we want to define a set*! macro to perform assignments in parallel, with

```
(set*! i1 e1 i2 e2 ...)
```

expanding into

```
(let ((t1 e1) (t2 e2) ...)
  (set! i1 t1)
  (set! i2 t2)
  ...)
```

This is hard to express for two reasons. The first is that the variables to be assigned need to be paired up with the expressions giving their new values. The second difficulty is that the macro must generate an indefinite number of temporary variables.

The `set*!` macro can be defined using an auxiliary macro to perform the pairing and to generate the temporaries. Although it might appear that the same temporary is being used for each assignment, the hygienic macro system automatically renames each binding occurrence that is inserted by a macro, together with all occurrences within its scope. Since each temporary that is inserted by the auxiliary macro eventually becomes a binding occurrence, each will be renamed. (Amazing but true.)

```
(define-syntax set*!
  (syntax-rules ()
    ((set*! i1 e1 more ...)
     (set*!-aux () i1 e1 more ...))))


(define-syntax set*!-aux
  (syntax-rules ()
    ((set*!-aux ((i1 e1 t1) ...))
     (let ((t1 e1) ...)
       (set! i1 t1) ...))
    ((set*!-aux ((i1 e1 t1) ...) i2 e2 more ...)
     (set*!-aux ((i1 e1 t1) ... (i2 e2 newtemp)) more ...))))
```

This definition of `set*!` may be compared with the `set*!` macro that is defined in the Revised^4 Report using the low-level macro system. Although `set*!` was put forth as an example of a macro that would be easier to write in a low-level macro system, I prefer the high-level definition above.

This definition would be more elegant if the auxiliary macro were local to each use of `set*!`. That would be less efficient because the auxiliary macro would have to be re-compiled for each use, but the real reason I didn't use a local macro is that I can't express the macro I want using `syntax-rules`!

The problem is annoyingly syntactic: If I tried to use `syntax-rules` to say that `set*!` expands into a `letrec-syntax` that defines `set*!-aux`, then the seven ellipses that appear in the definition of `set*!-aux` would appear on the right hand side of the syntax rule for `set*!`. The macro expander would try to expand these seven ellipses when transcribing the use of `set*!`, just as it expands the one ellipsis that appears on the right hand side of the syntax rule for `set*!` above. Somehow the macro expander needs to know to expand that one ellipsis, while waiting until the local macro is used to expand the other seven ellipses. A similar problem is solved in TeX by an escape character, and this solution should work in Scheme as well.

The low-level system described in the Revised[4] Report uses `syntax` as an escape character. Some such mechanism needs to be added to the high-level system.

Despite these problems, `syntax-rules` with recursion is reasonably expressive. It can express all recursive functions on lists, where `cons`, `car`, `cdr`, `null?`, and `equal?` on non-list elements are the base functions. The pattern language cannot do much with non-lists, however. It cannot take the successor of a numeral, for example.

The primary limitation of the hygienic macro system is that it is thoroughly hygienic, and thus cannot express macros that bind identifiers implicitly. Common Lisp's `defstruct` is an example of a non-hygienic macro, since it implicitly defines accessor functions for the structure. The `loop-until-exit` macro that is used as an example of the low-level macro system in the Revised[4] Report is also a non-hygienic macro. It may usefully be compared with similar macros in the following articles.

In conclusion, the hygienic macro system described in the Revised[4] Report is a considerable advance over previous macro systems. It has a few bugs yet, for which solutions can easily be imagined. Even if these bugs are fixed, however, there will still be non-hygienic macros and perhaps even a few hygienic macros that cannot be written using the hygienic macro system. That is why the Revised[4] Report also describes a compatible low-level macro system.

The low-level macro system described in the Revised[4] Report is experimental, and is likely to evolve considerably or be replaced altogether. Two alternative low-level macro systems are described in the following articles, and I am sure that even more will be designed during the next few years.

Scheme's hygienic macro system uses a stable technology, and Scheme programmers can use it to define most macros in a portable way. I am sure that the hygienic macro system will be extended, but I do not expect it to change very much. The hygienic macro system should certainly be considered for inclusion when the IEEE standard for Scheme [3] is next revised.

The low-level macro facilities provide an escape for those few macros that cannot be written using the hygienic macro system, but macros written using the low-level facilities will not be portable until the Scheme community has gained enough experience with these low-level macro systems to design and adopt one low-level system as an informal standard.

References

[1] William Clinger and Jonathan Rees [editors]. Revised^4 Report on the Algorithmic Language Scheme. To appear in Lisp Pointers.

[2] William Clinger and Jonathan Rees. Macros that work. In *Proceedings of the 1991 ACM Conference on Principles of Programming Languages*, pages 155–162.

[3] *IEEE Standard 1178-1990. IEEE Standard for the Scheme Programming Language.* IEEE, New York, 1991.

[4] Eugene Edmund Kohlbecker Jr. *Syntactic Extensions in the Programming Language Lisp.* PhD thesis, Indiana University, August 1986.

[5] Guy Lewis Steele Jr. *Common Lisp: The Language. Second Edition.* Digital Equipment Corporation, 1990.