Advanced Scheme Techniques

Some Naughty Bits

Jeremy Brown

January 7-8, 2003

Acknowledgements

Jonathan Bachrach, Alan Bawden, Chris Hanson, Neel Krishnaswami, and Greg Sullivan offered many helpful suggestions.

These slides draw on works by

Hal Abelson, Alan Bawden, Chris Hanson, Paul Graham, Oleg Kiselyov, Neel Krishnaswami, Al Petrofsky, Jonathan Rees, Dorai Sitaram, Gerry Sussman, Julie Sussman, and the R5RS authors group

Thanks also to Scheme Boston, the Boston-area Scheme User's Group.

And of course to SIPB, for organizing.

Scheme Macros

What is a Macro?

A macro is

- a stylized code transformation...
- performed without evaluating code...
- and using no runtime information.

What is a Macro?

A macro is

- a stylized code transformation...
- performed without evaluating code...
- and using no runtime information.

Suppose cond-set! is a macro:

```
(cond-set! (> test 4) var 15)
```

What is a Macro?

A macro is

- a stylized code transformation...
- performed without evaluating code...
- and using no runtime information.

Suppose cond-set! is a macro:

```
(cond-set! (> test 4) var 15)
```

This expression might expand to:

```
(if (> test 4) (set! var 15))
```

What is a Macro?

A macro is

- a stylized code transformation...
- performed without evaluating code...
- and using no runtime information.

Suppose cond-set! is a macro:

```
(cond-set! (> test 4) var 15)
```

This expression might expand to:

```
(if (> test 4) (set! var 15))
```

Note: the expansion process does not evaluate test or var!

When to Use Macros

Use macros to vary the order of evaluation

(in other words, to create new syntax/special forms).

For example:

- conditional evaluation (cond, case)
- repeated evaluation (do, named-let)
- binding (let, let*)
- un-evaluated syntactic tokens (case's =>)

When NOT to Use Macros

Any time you can avoid it!

- Don't use them for efficiency hacks.

  Let the compiler handle that.

Why not?

- Macros aren't first-class objects
  - You can't use a macro as any sort of runtime value
  - Thus, you reduce your development flexibility.
- You make debugging more difficult

Another Look at cond-set!

Remember our example:

```
(cond-set! (> test 4) var 15)
```

What's wrong with making cond-set! a function? E.g:

```
(define (cond-set! test variable value)
   (if test (set! variable value)))
```

Another Look at cond-set!

Remember our example:

```
(cond-set! (> test 4) var 15)
```

What's wrong with making cond-set! a function? E.g:

```
(define (cond-set! test variable value)
   (if test (set! variable value)))
```

The set! only mutates the parameter in the function.

The original var is unchanged.

Scheme's Derived Special Forms

Scheme actually has very few primitive special forms:

- lambda
- if
- quote
- set!

All the other forms *may* be derived using macros:

- conditionals (cond, case), binding (let, let*), etc.
- sequencing (begin, and, or)
- iteration (do, named let)

Of course, they may be implemented directly by the compiler, too.

Scheme Macro Systems

A number of macro systems have been implemented for various Schemes:

- Common Lisp-style *defmacro*
- *syntax-table*
- syntactic closures
- *syntax-case*
- *syntax-rules*
- ...and more!

syntax-rules is the macro system endorsed by the "Revised^5 Report on the Algorithmic Language Scheme" (R5RS). syntax-rules macros are often called "hygienic" macros.

Other Macro Systems You May Have Met

- m4
- tex/latex
- cpp

R5RS Macro Properties

Scheme ”hygienic macros’ feature three innovations
(as enumerated by Matthias Felleisen)

- Pattern-matching syntax
- hygiene
- referential transparency

And I’ll mention a fourth:

- Macro language is decoupled from base Scheme

We will cover these topics in order.

Macro Expansion Overview

A brief, somewhat inaccurate view of the macro expansion process:

- Pattern-matcher discovers an invocation form with macro keyword in operator position, e.g.
  `(unless (procedure? f) (display f))`
- Keyword is associated with one or more pattern/template pairs E.g.
  ```
  <(when condition consequent),
   (if (not condition) consequent)>
  ```
- If form matches a pattern, the corresponding template is filled in and replaces the form.

Pattern Language

A basic R5RS macro pattern is pretty straightforward:

- it is a list form
- its first element is the keyword
- strings, numbers, booleans, lists, vectors represent themselves
- non-keyword symbols represent *pattern variables*

For a form to match a pattern:

- each number, boolean, etc. must match exactly
- each pattern variable matches a single subform

Unaddressed so far: how do we represent specific symbols?

We'll come back to that in awhile.

Pattern Language Examples I

Pattern

```
(let1 (name value) body)
```

matches form

```
(let1 (x (read))
  (if (not x) (display "you said no")))
```

with the pattern variables matching like this:

```
name = x
value = (read)
body = (if (not x) (display "you said no"))
```

Pattern Language Examples II

Pattern

```
(contrived #((first . rest) #(3 any)))
```

matches form

```
(contrived #((1 2 3 4 5) #(3 '(foo))))
```

with the pattern variables matching like this:

```
first = 1
rest = (2 3 4 5)
any = '(foo)     AKA  (quote (foo))
```

Template Language

A template is an arbitrary Scheme form whose interpretation depends on the pattern it's paired with.

- numbers, booleans, lists, vectors represent themselves
- symbols which don't appear in the pattern represent themselves
- symbols which do appear in the pattern represent pattern variables

Expansion replaces each pattern variable in a template with the subform it matched in the input form.

Template Language Example

Pattern

```
(let1 (name value) body)
```

and template

```
(let ((name value)) body)
```

applied to form

```
(let1 (x (read))
  (if (not x) (display "you said no")))
```

expands to

```
(let ((x (read)))
  (if (not x) (display "you said no")))
```

Matching Multiple Forms at Once

A pattern variable followed by ... (an ellipsis) matches a group of consecutive forms.

For example, if we match the pattern

```
(dotimes count statement ...)
```

against the code form

```
(dotimes 5 (set! x (+ x 1)) (display x))
```

then

```
statement ... = (set! x (+ x 1)) (display x))
```

Template Expansion with Ellipses I

In a template, a pattern variable followed by an ellipsis expands into the group of forms it matched.

E.g. given this template for dotimes

```
(let dotimes-loop ((counter count))
    (if (> counter 0)
      (begin
        statement ...
        (dotimes-loop (- counter 1)))))
```

Template Expansion with Ellipses II

...then the expansion will look like this:

```
(let dotimes-loop ((counter 5))
    (if (> counter 0)
      (begin
        (set! x (+ x 1))
        (display x)
        (dotimes-loop (- counter 1)))))
```

Producing Repeated Forms

Suppose we want

```
(thunkify 5 (* x x))
```

to expand to

```
(list (lambda () 5) (lambda () (* x x)))
```

This does the trick:

Pattern: `(thunkify body ...)`

Template: `(list (lambda () body) ...)`

Matching Repeated Forms

Suppose we want

```
(update-if-true!
  ((> x 5) x-is-big)
  ((zero? y) y-is-zero))
```

to expand to

```
(begin
    (let ((test (> x 5)))
      (if test (set! x-is-big test)))
    (let ((test (zero? y)))
      (if test (set! y-is-zero test))))
```

Matching Repeated Forms II

We can match a group of forms by following a form with ...

pattern variables in the form match the corresponding subforms.

This does the trick:

Pattern:

```
(update-if-true! (condition variable) ...)
```

Template:

```
(begin (let ((test condition))
  (if test (set! variable test))) ...)
```

Nesting Ellipses

Ellipses may be nested in both patterns and templates.

A highly artificial example: we want this

```
(quoted-append (1 2 3) (a b c) (+ x y))
```

to expand into this

```
'(1 2 3 a b c + x y)
```

This does it:

Pattern: `(quoted-append (guts ...) ...)`

Template: `(quote (guts ... ...))`

This can be tricky!

Grouping Pattern/Template Pairs

A keyword may be associated with multiple pattern/template pairs. The complete ruleset for a keyword is given by a *syntax-rules* form, for instance this syntax-rules for *and* from R5RS:

```
(syntax-rules ()
  ((and) #t)                         ; first pair
  ((and test) test)                  ; second pair
  ((and test1 test2 ...)             ; third pattern
   (if test1 (and test2 ...) #f))))  ; third templ.
```

Notes on syntax-rules

- Patterns may contain their keyword, causing recursive expansion!
- Forms are matched against patterns in top-down order
- and *syntax-rules* solves another problem for us...

Representing Specific Symbols in Patterns

Suppose we want

`(implications (a => b) (c =>d) (e =>f))`

to expand to

`(begin (if a b) (if c d) (if e f))`

But ”=>” is a scheme symbol just like ”foo”; if we write

```
(syntax-rules ()
  ((implications (condition => consequent) ...)
   (begin (if condition consequent) ...)))


(implications (test =. (set! testp #t))) ;typo
```

then “=>” would match =., instead of the expander signaling an error.

Representing Specific Symbols in Patterns II

We can specify a list of non-pattern-variable symbols as part of syntax rules, for example

```
(syntax-rules (=>)
  ((implications (condition => consequent) ...)
   (begin (if condition consequent) ...)))
```

Now the “=>” in the pattern will only match the symbol “=>”; it is no longer a pattern variable.

Binding Constructs

We are now ready to use syntax-rules for real. But how?

We have three options:

- `(let-syntax bindings body)`
- `(letrec-syntax bindings body)`
- `(define-syntax symbol (syntax-rules ....`
  (top level only)

All behave in an extraordinarily obvious way.

Representing Specific Symbols in Patterns III

One final subtlety arises. Consider

```
(define-syntax implications
  (syntax-rules (=>) ; body elided
))


(let ((=> 5))
  (implications (foo => bar)))
```

In the define-syntax form, “=>” names an implicit top-level binding.

In the implications form, “=>” names the let binding.

Because of this, they do not match. Thus,

in this lexical context, the expansion of implications will fail.

Representing Specific Symbols in Patterns IV (Final)

In general:

- A literal symbol in a syntax-rules names a binding in the lexical scope of the syntax-rules.
- A symbol in a form names a binding in the lexical scope in which the form appears.
- A symbol in a form will only match a literal symbol in a pattern if both symbols name the same binding.
- (A symbol which doesn't correspond to an explicit binding is assumed to correspond to an implicit binding in the top level.)

The Pattern Keword

The keyword that starts a pattern is ignored. Some people prefer to use _ in its place:

```
(define-syntax when
  (syntax-rules ()
        ((_ cond consequent ...)
          (if cond (begin consequent ...)))))
```

Thoughts on this convention:

- Visually distinguishes the beginning of a pattern
- Makes renaming a macro somewhat easier
- Yet somehow tasteless.

Debugging Tips

- Quote the result of an expansion that's giving you trouble in order to see the intermediate result.
- If MIT scheme says "Hardware trap SIGSEGV" when you define a macro, it means you have ellipses that don't make sense.
- MIT scheme doesn't like ellipses in vectors, i.e. `#(a ...)` buys you a SIGSEGV message.

Yes, I'll report these bugs. RSN.

10 Minute Break

R5RS Macro Semantics

Macro Expansion: Apparent Danger

```
(define-syntax dotimes
  (syntax-rules ()
    ((dotimes count body ...)
     (let loop ((counter count))
       (if (> counter 0)
           (begin
             body ...
             (loop (- counter 1))))))))

(define counter 0)
(dotimes 5 (set! counter (+ counter 1)))
```

Does the dotimes invocation ever terminate?

Dotimes Expansion

```
(let loop ((counter 5))
  (if (> counter 0)
      (begin
        (set! counter (+ counter 1))
        (loop (- counter 1)))))
```

Doesn't look good!

Appears that the let binding of counter will capture the user's reference to counter.

This really happens with some macro systems — but not Scheme's.

Macro Hygiene

Hygienic macros cannot contaminate the lexical scope in which they expand by introducing symbols that shadow other bindings.

Hygienic Dotimes Expansion

A hygienic expansion contains extra information:

```
(let loop ((counter 5))
  (if (> counter 0)
      (begin
        (set! counter (+ counter 1))
        (loop (- counter 1)))))
```

counter is not the same as counter, and therefore is not captured by the let.

Substitutions *Can* Shadow

Using pattern-variables, macros can create bindings that shadow lexically-enclosing bindings.

```
(define-syntax shadow
  (syntax-rules ()  ((shadow used-arg body)
                     (let ((used-arg 5)) body))))
(define test 7)
(shadow test test))
```

yields

```
(let ((test 5)) test)  ==> 5
```

This technique is how binding constructs are implemented.

Another Apparent Danger

```
(define-syntax dotimes
  (syntax-rules ()
    ((dotimes count body ...)
     (let loop ((counter count))
       (if (> counter 0)
           (begin
             body ...
             (loop (- counter 1))))))))

(let ((- 'minus))
  (dotimes 5 (display "hello, world"))
```

Does (- counter 1) evaluate properly?

Textual Expansion

```
(let ((- 'minus))
  (let loop ((counter 5))
    (if (> counter 0)
        (begin
          (display "hello, world")
          (loop (- counter 1))))))
```

Fudging for clarity: the outer let, if it were implemented as a macro, would have already expanded first.

Referential Transparency

Referentially-transparent macros are not contaminated by the lexical scope in which they expand.

Identifiers introduced by the template refer to their bindings at the point of macro definition.

Referentially Transparent Dotimes Expansion

```
(let ((- 'minus))
  (let loop ((counter 5))
    (if (> counter 0)
        (begin
          (display "hello, world")
          (loop (- counter 1))))))
```

- is not the same as -; the macro expansion contains information differentiating identifiers introduced by the macro template from those bound in the expansion context.

Lexically-Safe Macros

hygiene and *referential transparency* combine to make R5RS macros *lexically safe*: lexical scoping is always preserved. In particular:

- macros don't interfere with their expansion environments (hygiene)
- macros aren't interfered with by their expansion environments (referential transparency)
- temporary names introduced by recursive/nested macros never collide
- "Principle of least surprise"

R5RS macros are often (inaccurately) simply called "hygienic".

R5RS Macro Properties In Review

Felleisen's three key properties:

- pattern language
- hygiene
- referential transparency

The implication:

- Hygiene combined with referential transparency guarantees lexical scoping at all times...
- ...Even when you don't want it to. (We'll touch on that later.)

One More Property

The R5RS macro language is decoupled from base Scheme

Most (all?) other macro systems let you use Scheme as part of the macro expansion process, i.e.

- Some Scheme is executed at compile-time, to help produce...
- Scheme which is executed at runtime

Such macros can be exceedingly hard to read.

Design Idioms

Ordering Multiple Patterns

A form is compared against a keyword's patterns in top-down order.

In dealing with things that process lists in order:

- The first form should match zero items
- The second should match one item
- The third should generalize for more than one item

For example, *and*...

Ordering Multiple Patterns: R5RS Example

```
(define-syntax and
  (syntax-rules ()
    ((and) #t)          ; zero items
    ((and test) test)   ; one item
    ((and test1 test2 ...) ; two items
     (if test1 (and test2 ...) #f))))
```

Your macros may not need all three cases.

Staged Expansion

Consider this pair of macros:

```
(define-syntax reverse-and-quote-list
  (syntax-rules ()
    ((reverse-and-quote-list list)
     (rl-helper list ()))))

(define-syntax rl-helper
  (syntax-rules ()
    ((rl-helper () (backw ...))
     '(backw ...))
    ((rl-helper (arg rest ...) (backw ...))
     (rl-helper (rest ...) (arg backw ... )))))

(reverse-and-quote-list (1 2 3 4 5))  ==> (5 4 3 2 1)
```

rl-helper can't be nested: it has ... in it

Staged Expansion Using Keystrings

Use “keystrings” to stage expansion without cluttering the namespace with innumerable helper macros.

```
(define-syntax reverse-and-quote-list
  (syntax-rules ()
    ((reverse-and-quote-list "helper" () (backw ...))
     '(backw ...))
    ((reverse-and-quote-list "helper" (arg rest ...)
                                      (backw ...))
     (reverse-and-quote-list "helper" (rest ...)
                                      (arg backw ... )))
    ((reverse-and-quote-list (list ...))
     (reverse-and-quote-list "helper" (list ...)  ()))))
```

“Tail recursive” strategy.

Macro Subroutines: “Falling-Forward”

Consider:

```
(let ((letrec 5)) letrec) ==> 5
```

The expansion process can reorganize the inner forms, so the outermost layer of a form is always expanded first.

The next expansion doesn’t begin until the first one is completely done.

How can an macro call another macro as a subroutine, and continue expanding afterward?

Macro Subroutines

Passing the Next Macro as an Argument

To use a macro as a subroutine, we pass along the "return" as pair of arguments to be used in a "tail-call" strategy.

- *future-keyword* names the macro to be applied when this one is through.
- *future-args* provides initial arguments to that macro.

Example:

```
(define-syntax cps-quote
  (syntax-rules ()
    ((cps-quote future-keyword (future-args ...) stuff ...)
     (future-keyword future-args ... (quote stuff ...)))))
```

Terminal Macros

Some macros are “terminal” — they don’t call any more macros:

```
(define-syntax apply-to-result
  (syntax-rules ()
    ((apply-to-result func list ...)
     (func list ...))))
```

We can use this to output a quoted value:

```
(cps-quote apply-to-result ((lambda (x) x)) (1 2 3 4 5))
 ==>
  (1 2 3 4 5)
```

Staged Subroutine Macros

Staged macros pass their future-arguments along until they're done:

```
(define-syntax cps-reverse-list
  (syntax-rules ()
    ((cps-reverse-list "helper" future-keyword
                       (future-args ...) () (backw ...))
     (future-keyword future-args ... (backw ...)))
    ((cps-reverse-list "helper" future-keyword
                       future-args (arg forw ...) (backw ...))
     (cps-reverse-list "helper" future-keyword
                       future-args (forw ...) (arg backw ...)))
    ((cps-reverse-list future-keyword future-args (list ...))
     (cps-reverse-list "helper" future-keyword future-args
                       (list ...) ()))))
```

Nesting

The more “subroutines” you want to call, the longer the chain:

```
(cps-reverse-list
  cps-quote
  (apply-to-result ((lambda (x) x))) (1 2 3 4 5))
 ==>
    (5 4 3 2 1)
```

CPS (Continuation Passing Style)

At any point in a macro expansion series, the future-keyword/future-args pair represent all of the expansion to come.

We say they represent the “continuation” of the macro expansion.

The subroutining style is called Continuation Passing Style (CPS).

But we’ve only talked about macro-continuations, and macros aren’t first-class.

Tomorrow:

First Class Scheme Continuations

They ain’t macros...

Tomorrow:

First Class Scheme Continuations

They ain't macros...

but first...

R5RS Macro Limitations...

R5RS Macro Limitations...

...and Workarounds

Hygiene-Imposed Limitation

Suppose we *want* dotimes to expose a variable counter, so that a programmer could write

```
(dotimes 5 (display counter))
```

and get “54321” as the result.

We already know that hygiene prevents this definition from working that way:

```
(define-syntax dotimes
  (syntax-rules ()
    ((dotimes count body ...)
     (let loop ((counter count))
   ....
```

The Workaround

We know that we *can* use symbols from the arguments to the macro to create bindings, e.g. let, letrec, etc.

In fact, if the arguments contain the identifier you're looking for (e.g. counter) anywhere, you can dig it out and use it!

Petrofsky's find-identifier

```
(define-syntax find-identifier
  (syntax-rules ()
    ((_ ident (x . y) sk fk)
     (find-identifier ident x sk
                      (find-identifier ident y sk fk)))
    ((_ ident #(x ...) sk fk)
     (find-identifier ident (x ...) sk fk))
    ((_ ident form sk fk)
     (let-syntax
         ((check
           (syntax-rules (ident)
             ((_ ident ident* (s-f . s-args) fk_)
              (s-f ident* . s-args))
             ((_ x y sk_ fk_) fk_))))
       (check form form sk fk)))))
```

“Hygiene-Violating” dotimes

```
(define-syntax dotimes
  (syntax-rules ()
    ((dotimes count body ...)
     (find-identifier counter (body ...)
                      (dotimes-finish count body ...)
                      (dotimes-finish temp count body ...)))))

(define-syntax dotimes-finish
  (syntax-rules ()
    ((dotimes-finish counter count body ...)
     (let loop ((counter count))
       (if (> counter 0)
           (begin
             body ...
             (loop (- counter 1))))))))
```

And Sure Enough

```
(dotimes 5 (display counter))
   prints 54321
```

Homework Problem: Nesting Scopes

We expect this

```
(dotimes 5 (dotimes 5 (display counter)))
```

to yield

```
5432154321543215432154321
```

But actually we get

```
5555544444333332222211111
```

It can be solved...

...but it is tricky.

The Other Commonly-Complained About R5RS Limitation

You cannot synthesize symbols, and thus you cannot make a macro `(define-structure foo)` which defines make-foo, is-foo?, etc.

The workaround I've seen is ugly or unportable.

The End

But come back tomorrow for the Continuations lecture!