Hygienic Literate Programming: ChezWEB

Aaron W. Hsu <awhsu@indiana.edu>

Scheme Workshop 2011

What this talk is not

I am not proselytizing literate programming

You will not learn to program in a literate style

You will not learn to use ChezWEB (it's not hard enough for a talk)

What this talk is

About programming DSLs

About Scheme macros

About `syntax-case`

About hygiene in DSLs

Literate Programming

Writing programs to be read by humans

An emphasis on appropriate use of prose and code

Implemented many different ways

Available for Scheme: SchemeWEB, Scribble/LP, noweb, &c.

Traditional systems emphasize a special syntax for code lifting

Traditional Example

```
@ This code will be lifted out.

@<Compute factorial@>=
(let fact ([n 1]) (if (= n max) n (* n (fact (1+ n)))))

@ Used here.

@p
(define (factorial max) @<Compute factorial@>)
```

Hygiene and Literate Programming

Traditional approaches copy and paste text verbatim

Completely unhygienic

Sort of “dynamic scope” introduced into the language

Not visible or always obvious in woven output.

Claim: this is bad for coding reliability and for our mental models

Reformulating Code Lifting as a Macro

```
; This code is be lifted out.

(@< (|Compute factorial|)
  (let fact ([n 1])
    (if (= n max) n (* n (fact (1+ n))))))

; Used here.

(define (factorial max) |Compute factorial|)
```

Literate programming code lifting/reordering
is just another Scheme macro

What should this macro be?

PUT ON YOUR MACRO HATS

Our Macro Challenge

```
(@< name (captures ...) (exports ...) body+ ...)
```

`syntax-case` Primer

```
(let ([x 'inner-x] [y 'inner-y])
  (define-syntax (grab-x z)
    (syntax-case z ()
      [(k)
       (with-syntax ([x (datum->syntax #'k 'x)])
         #'(list x y))]))
  (let ([x 'outer-x] [y 'outer-y])
    (grab-x)))
;;; => '(outer-x inner-y)
```

syntax-case Primer

```
(let ([x 'inner-x] [y 'inner-y])
  (define-syntax (grab-x z)
    (syntax-case z ()
      [(k)
       (with-syntax ([x (datum->syntax #'k 'x)])
         #'(list x y))]))
  (let ([x 'outer-x] [y 'outer-y])
    (grab-x)))
;;; => '(outer-x inner-y)
```

`syntax-case` Primer

```
(let ([x 'inner-x] [y 'inner-y])
  (define-syntax (grab-x z)
    (syntax-case z ()
      [( k )
       (with-syntax ([ x (datum->syntax #'k 'x)])
         #'(list x y))]))
  (let ([x 'outer-x] [y 'outer-y])
    (grab-x)))
;;; => '(outer-x inner-y)
```

The traditional version

```
(define-syntax (@< x)
  (syntax-case x ()
    [(_ name (c ...) (e ...) b1 b2 ...)
     #'(define-syntax (name x)
         (datum->syntax x '(begin b1 b2 ...)))]))
```

Fails

```
(define-syntax (@< x)
  (syntax-case x ()
    [(_ name (c ...) (e ...) b1 b2 ...)
     #'(define-syntax (name x)
         (datum->syntax x '(begin b1 b2 ...)))]))

(let ([x 3])
  (@< ex1 (x) (y) (define y (list x)))
  (let ([x 0] [list '(1)]) ex1 (append y list)))
; => '(0 1)
```

Fails

```
(define-syntax (@< x)
  (syntax-case x ()
    [(_ name (c ...) (e ...) b1 b2 ...)
     #'(define-syntax (name x)
         (datum->syntax x '(begin b1 b2 ...)))]))
(let ([x 3])
  (@< ex1 (x) (y) (define y (list x)))
  (let ([x 0] [list '(1)]) ex1 (append y list)))
```

```
(let ([x.0 3])
  (let ([x.1 0] [list.2 '(1)])
    (letrec* ([y.3 (list.2 x.1)])
      (#2%append y.3 list.2))))
```

Our “Principles”

Hygiene. No definition in the body of a code section shall be visible to the surrounding context of a section reference unless that binding is explicitly exported by the code section at the point of definition.

Referential Transparency. Free variables in the body of a code section will always refer to the nearest lexical binding at the definition site of the code section, unless the identifier is explicitly noted as a capture at the definition site, in which case the binding to a captured variable will refer to the nearest lexical binding at the reference site of the code section.

Go full hygienic

```
(define-syntax (@< x)
  (syntax-case x ()
    [(_ n (c ...) (e ...) b1 b2 ...)
     #'(define-syntax (n x)
          (syntax-case x ()
            [(_ c ... e ...) #'(begin b1 b2 ...)]))]))
```

Fails

```
(define-syntax (@< x)
  (syntax-case x ()
    [(_ n (c ...) (e ...) b1 b2 ...)
     #'(define-syntax (n x)
         (syntax-case x ()
           [(_ c ... e ...) #'(begin b1 b2 ...)]))]))

(let ()
  (@< ex2 () (make-x x?) (define-record-type x))
  (ex2 make-x x?)
  (x? (make-x)))
; => #t
```

Fails, Expansion

```
(define-syntax (@< x)
  (syntax-case x ()
    [(_ n (c ...) (e ...) b1 b2 ...)
     #'(define-syntax (n x)
         (syntax-case x ()
           [(_ c ... e ...) #'(begin b1 b2 ...)]))]))
(let ()
  (@< ex2 () (make-x x?) (define-record-type x))
  (ex2 make-x x?)
  (x? (make-x)))
```

```
(letrec* (...
          [ make-x.2 <make constructor>]
          [ x?.3 <make predicate>]
          ...)
  (x? (make-x)))
```

Still fails, though more succinctly

```
(define-syntax (@< x)
  (syntax-case x ()
    [(_ n (c ...) (e ...) b1 b2 ...)
     #'(define-syntax (n x)
         (syntax-case x ()
           [ n (identifier? #'n)
            (with-implicit (n c ... e ...)
              #'(begin b1 b2 ...))]))]))
```

Core problem

We want to be able to use and reference the same values in different contexts.

Link or alias the two contexts together so that they point to the same locations.

```
(define-syntax inner
  (identifier-syntax
    [inner outer]
    [(set! inner val) (set! outer val)]))
```

Chez Scheme nicety

```
(alias inner outer)
```

Linking with alias

```
(define-syntax (@< x)
  (syntax-case x ()
    [(_ n (c ...) (e ...) b1 b2 ...)
     #'(define-syntax (n x)
         (syntax-case x ()
           [ n (identifier? #'n)
            (with-syntax ([(ic (... ...)) #'(c ...)]
                          [(ie (... ...)) #'(e ...)]
                          [(oc (... ...))
                           (datum->syntax x '(c ...))]
                          [(oe (... ...))
                           (datum->syntax x '(e ...))])
              #'(begin
                  (alias ic oc) (... ...)
                  b1 b2 ...
                  (alias oe ie) (... ...)))]))]))
```

Encapsulating bodies with modules

```
(define-syntax (@< x)
  (syntax-case x ()
    [(_ n (c ...) (e ...) b1 b2 ...)
     #'(define-syntax (n x)
         (syntax-case x ()
           [ n (identifier? #'n)
            (with-syntax ([(ic (... ...)) #'(c ...)]
                          [(ie (... ...)) #'(e ...)]
                          [(oc (... ...))
                           (datum->syntax x '(c ...))]
                          [(oe (... ...))
                           (datum->syntax x '(e ...))])
              #'(module (oe (... ...))
                   (alias ic oc) (... ...)
                   (module (ie (... ...))
                     b1 b2 ...)
                   (alias oe ie) (... ...)))]))]))
```

Still not right!

```
(define-syntax (@< x)
  (syntax-case x ()
    [(_ n (c ...) (e ...) b1 b2 ...)
     #'(define-syntax (n x)
         (syntax-case x ()
           [ n (identifier? #'n)
            (with-syntax ([(ic (... ...)) #'(c ...)]
                          [(ie (... ...)) #'(e ...)]
                          [(oc (... ...))
                           (datum->syntax x '(c ...))]
                          [(oe (... ...))
                           (datum->syntax x '(e ...))])
              #'(module (oe (... ...))
                  (alias ic oc) (... ...)
                  (module (ie (... ...))
                    ((... ...) b1) ((... ...) b2) ...)
                  (alias oe ie) (... ...)))]))]))
```

Limitations

We can not do alpha renaming like the first version

Hygienic version can do alpha renaming

You can only use this where expressions are evaluated

Only does definitions, but easily extended to values

Question: why was this not the first thing that everyone thinks of?

Thank you.

```
(let ()
  (define a 1)
  (define-syntax (m x)
    (syntax-case x ()
      [(_ a) #'(y here)]))
  (define-syntax (y x)
    (syntax-case x ()
      [(_ k) (datum->syntax #'k 'a)]))
  (m a))
```