Keeping it Clean with Syntax Parameters

Eli Barzilay, Ryan Culpepper, Matthew Flatt
Macros

Macros are great.
Macros

Hygienic macros are great.
Hygienic macros are great, but...
Macros

Hygienic macros are great, but...

```
(define-struct point (x y))
(point-x (make-point 1 2))
```
Macros

Hygienic macros are great, but...

\[
\begin{align*}
&(\text{define-struct point} \ (x \ y)) \\
&(\text{point-x} \ (\text{make-point} \ 1 \ 2)) \\
\Rightarrow&(\text{datum->syntax} \ \text{name} \ \text{a-symbol})
\end{align*}
\]
Hygienic macros are great, but...

```
(define-syntax forever
  (syntax-rules ()

    [((forever body ...)
      (call/cc (lambda (abort)
            (let loop ()
                  body ...
                  (loop))))))]
```
Macros

Hygienic macros are great, but...

```
(define-syntax aif
  (syntax-rules ()
    [(aif test then else)
      (let ([it test])
        (if it then else))])))
```
Non-Solution#1

```
(define-syntax (forever stx)
  (syntax-case stx ()
    [(forever body ...)
      (with-syntax ([abort (datum->syntax #'forever 'abort)])
        #'(call/cc (lambda (abort)
                      (let loop ()
                        body ... (loop))))))]
```
Non-Solution#1

(define-syntx (forever stx)
  (syntax-case stx ()
    [(forever body ...)
      (with-syntx ([abort (datum->syntax #'forever #'abort)])
        #'(call/cc (lambda (abort)
          (let loop ()
            body ... (loop)))))]))

(define-syntx while
  (syntax-rules ()
    [(while test body ...)
      (forever (unless test (abort)) body ...)]))
Non-Solution#1

```
(define-syntx (forever stx)
 (syntax-case stx ()
  [(forever body ...) (with-syntx ([[abort (datum->syntax #\'forever #\'abort)])
    #\'(call/cc (lambda (abort)
      (let loop ()
        body ... (loop))))))])
)

(define-syntx while
 (syntax-rules ()
  [(while test body ...) ([forever (unless test (abort)) body ...)])])

>(while #t (abort))
```
Non-Solution#1

```
(define-syntax (forever stx)
  (syntax-case stx ()
    [(forever body ...) (with-syntax ([abort (datum->syntax #'forever 'abort)]) #'(call/cc (lambda (abort)
       (let loop ()
         body ... (loop))))))
)

(define-syntax while
  (syntax-rules ()
    [(while test body ...) (forever (unless test (abort)) body ...)])

> (while #t (abort))
reference to undefined identifier: abort
```
(define-syntax (forever stx)
  (syntax-case stx ()
    [(forever body ...)
      (with-syntax ([abort (datum->syntax #'forever 'abort)])
        #'(call/cc (lambda (abort)
                    (let loop ()
                      body ... (loop))))))]

(define-syntax (while stx)
  (syntax-case stx ()
    [(while test body ...)
      (with-syntax ([forever (datum->syntax #'while 'forever)])
        #'(forever (unless test (abort))
               body ...))])))
Non-Solution#1

(define-syntax (forever stx)
  (syntax-case stx ()
    [(forever body ...)
     (with-syntax ([abort (datum->syntax #'forever 'abort)])
      #'(call/cc (lambda (abort)
                    (let loop ()
                      body ... (loop)))))])
)

(define-syntax (while stx)
  (syntax-case stx ()
    [(while test body ...)
     (with-syntax ([forever (datum->syntax #'while 'forever)])
      #'(forever (unless test (abort))
              body ...))))}
Non Solution #2

“Hygiene macros are ok, but for real code, use defmacro”
Fix Solution #1

(define-syntax (while stx)
  (syntax-case stx ()
    [(while test body ...) #'(forever (unless test (abort))
                            body ...)]))
(define-syntax (while stx)
  (syntax-case stx ()
    [(while test body ...)
      (with-syntax (; abort* is user-accessible as `abort'
        [abort* (datum->syntax
                 #'while 'abort)]
      #'(forever (let (; link the two bindings
            [abort* abort])
        (unless test (abort))
          body ...)))]))
(define-syntax (while stx)
  (syntax-case stx ()
    [(while test body ...)]
    ([with-syntax (; abort* is user-accessible as `abort'
                   [abort* (datum->syntax #'while 'abort)])
      #'(forever (let (; link the two bindings
                      [abort* abort])
                   (unless test (abort))
                   body ...)))]))

(define-syntax (until stx)
  (syntax-case stx ()
    [(until test body ...)]
    ([with-syntax ([abort* (datum->syntax #'until 'abort)])
      #'(while (not test)
           (let ([abort* abort]) body ...)))]))
Fix Solution #1

(define-syntax (while stx)
  (syntax-case stx ()
    [(while test body ...)
     (with-syntax (; abort* is user-accessible as `abort'
                   [abort* (datum->syntax
                               #'while 'abort)]
                   #'(forever (let (; link the two bindings
                                [abort* abort])
                                (unless test (abort))
                                body ...))))]))

(define-syntax (until stx)
  (syntax-case stx ()
    [(until test body ...)
     (with-syntax ([abort* (datum->syntax
                              #'until 'abort)]
                   #'(while (not test)
                       (let ([abort* abort] body ...)))))])

• What if abort is a macro binding?

• Not mechanical enough to automate
Fix Solution #1

(define-synatx (while stx)
    (syntax-case stx ()
        [(while test body ...)
            (with-syntax (; abort* is user-accessible as `abort'
                [abort* (datum->syntax
                    #'while 'abort)])
                #'(forever (let (; link the two bindings
                    [abort* abort])
                (unless test (abort))
                    body ...)))]))

(define-synatx (until stx)
    (syntax-case stx ()
        [(until test body ...)
            (with-syntax ([abort* (datum->syntax
                #'until 'abort)])
                #'(while (not test)
                    (let ([[abort* abort]] body ...)))]))

• (make-rename-transformer #'abort)

• Specify “link point”
Automated Solution

Define a `define-syntax-rules/capture` macro to automate linking. “Link points” specified with an L.

```
(define-syntax-rules/capture forever (abort) ()
  [(forever body ...)
   (call/cc (lambda (abort)
     (L (let loop () body ... (loop))))))]

(define-syntax-rules/capture while (abort) ()
  [(while test body ...)
   (forever (L (unless test (abort)) body ...))])

(define-syntax-rules/capture until (abort) ()
  [(until test body ...)
   (while (L (not test)) (L body ...))])
```

→ We can even use the same macro to define the base level `forever` macro.
Automated Solution

Define a `define-syntax-rules/capture` macro to automate linking. “Link points” specified with an \L.

```
(define-syntax-rules/capture forever (abort) ()
  [(forever body ...)
   (call/cc (lambda (abort)
              (L (let loop () body ... (loop))))))]

(define-syntax-rules/capture while (abort) ()
  [(while test body ...)
   (forever (L (unless test (abort)) body ...))])

(define-syntax-rules/capture until (abort) ()
  [(until test body ...)
   (while (L (not test)) (L body ...))])

(define-syntax until
  (syntax-rules ()
    [(until test body ...)
     (while (not test) body ...)]))
```
Automated Solution

Define a **define-syntax-rules/capture** macro to automate linking. “Link points” specified with an \texttt{L}.

\begin{verbatim}
(define-syntax-rules/capture forever (abort) ()
  [(forever body ...)
    (call/cc (lambda (abort)
      (L (let loop () body ... (loop))))))]

(define-syntax-rules/capture while (abort) ()
  [(while test body ...)
    (forever (L (unless test (abort)) body ...))])

(define-syntax-rules/capture until (abort) ()
  [(until test body ...)
    (while (L (not test)) (L body ...))])
\end{verbatim}

\textbf{Note:}

The following example:

\begin{verbatim}
(define-syntax until
  (syntax-rules ()
    [(until test body ...)
      (while (not test) body ...)])
\end{verbatim}

does not propagate the \texttt{abort} binding.
The “Simple” Utility

(define-syntax (define-syntax-rules/capture stx0)
  (syntax-case stx0 ()
    [(def name (capture ...) (keyword ...) [patt templ] ...) (with-syntax ([(L (datum->syntax #'def 'L))]
      #'(define-syntax (name stx)
        (syntax-case stx (keyword ...)
          [patt (with-syntax ([(user-ctx stx)]
            #'(with-links L user-ctx (capture ...) templ))]
            ...)))])]

(define-syntax with-links
  (syntax-rules ()
    [(with-links L user-ctx (capture ...) template)
      (let-syntax
        ([(L (lambda (stx)
          (syntax-case stx ()
            [(L e (... . . .))
              (with-syntax ([(id ( ... . . .)) (list (datum->syntax #'L 'capture) . . .)]
                [(id* ( ... . . .)) (list (syntax-local-introduce
                  (datum->syntax #'user-ctx 'capture))
                ...))]
                #'(let-syntax ([(id* (make-rename-transformer #'id))
                  (... . . .))
                  e (... . . .))]
            template)))]))
• Tedious to propagate unhygienically-bound names around

• Might not be possible with library macros that we didn’t write

⇒ Same kind of problems that lead to `fluid-let`. 
Non Solution #3

“Never break hygiene!” — always specify bindings.
Non Solution #3

“Never break hygiene!” — always specify bindings.

(define-syntax forever
  (syntax-rules ()
    [(forever abort body ...)
      (call/cc (lambda (abort)
                  (let loop () body ... (loop))))])))
Non Solution #3

“Never break hygiene!” — always specify bindings.

```
(define-syntax forever
  (syntax-rules ()
    [(forever abort body ...) (call/cc (lambda (abort)
                                          (let loop () body ... (loop)))])

(define-syntax aif
  (syntax-rules ()
    [(aif it test then else) (let ([it test]) (if it then else))])
```
Non Solution #3

“Never break hygiene!” — always specify bindings.

(define-syntax forever
  (syntax-rules ()
    [(forever abort body ...) (call/cc (lambda (abort)
        (let loop () body ... (loop))))]))

(define-syntax aif
  (syntax-rules ()
    [(aif it test then else) (let ([it test]) (if it then else))])

(define-syntax while
  (syntax-rules ()
    [(while abort it test body ...) (forever abort
      (aif it test (begin body ...) (abort)))]))
Non Solution #3

But this is worse...

(while abort it (memq x l)
 (display (car it))
 (set! l (cdr it)))
Non Solution #3

But this is worse...

```scheme
(while abort it (memq x l)
  (display (car it))
  (set! l (cdr it)))
```

```scheme
(define-syntax until
  (syntax-rules ()
    [(until abort it test body ...)\n     [(while abort it (not test) body ...)]))
```
Non Solution #3

But this is worse...

\[
\begin{align*}
\text{(while abort it (memq x l)} & \\
& \quad \text{(display (car it))} \\
& \quad \text{(set! l (cdr it))})
\end{align*}
\]

\[
\begin{align*}
\text{(define-synt{y}x until} & \\
\text{ (syntax-rules (}} & \\
& \quad \text{[(until abort it test body ...)} \\
& \quad \quad \text{(while abort it (not test) body ...)]})
\end{align*}
\]

(Even worse with core language constructs.)
Solution: Dynamic Bindings

In the runtime world, we avoid threading parameters along call-chains using “dynamic bindings”.
Solution: Dynamic Bindings

In the runtime world, we avoid threading parameters along call-chains using “dynamic bindings”.

```lisp
(define (abort)
  (error "abort must be used in a loop"))
(define (thunk-forever body-thunk)
  (call/cc
   (lambda (k)
     (fluid-let ([abort k])
       (let loop () (body-thunk) (loop))))))
(thunk-forever
  (lambda ()
    (let ([c (read-char)])
      (if (eof-object? c)
        (abort)
        (display (char-upcase c)))))))
```
Solution: Dynamic Bindings

In the runtime world, we avoid threading parameters along call-chains using “dynamic bindings”.

```
(define (abort)
  (error "abort must be used in a loop"))
(define (thunk-forever body-thunk)
  (call/cc
   (lambda (k)
     (fluid-let ([abort k])
       (let loop () (body-thunk) (loop))))))
(thunk-forever
  (lambda ()
    (let ([c (read-char)])
      (if (eof-object? c)
        (abort)
        (display (char-upcase c))))))
```

▶ The binding is lexical, the value is dynamically adjusted
Solution: Parameters

fluid-let is too strong: \( (\text{fluid-let } \text{[cons } +\text{]} \text{ ...}) \)

Parameters: avoid indiscriminate use.
Solution: Parameters

fluid-let is too strong: \( \text{fluid-let ([cons +]) ...} \)

Parameters: avoid indiscriminate use.

\[
(\text{define current-abort} \\
(\text{make-parameter} \\
(\lambda () (\text{error "abort must be used in a loop"}))))
\]

\[
(\text{define (abort) ((current-abort)))}
\]

\[
(\text{define (thunk-forever body-thunk)} \\
(\text{call/cc} \\
(\lambda (k) \\
(\text{parameterize ([[current-abort k]])} \\
(\lambda (loop) (body-thunk) (loop))))))
\]

Solution: Parameters

- fluid-let is too strong: \( (\text{fluid-let } ([\text{cons } +]) \ldots) \)

Parameters: avoid indiscriminate use.

\[
\begin{align*}
(\text{define } \text{current-abort} \\
(\text{make-parameter} \\
(\lambda () (\text{error } \text{"abort must be used in a loop"}))))
\end{align*}
\]

\[
(\text{define } \text{abort} ((\text{current-abort})))
\]

\[
(\text{define } \text{thunk-forever } \text{body-thunk} \\
(\text{call/cc} \\
(\lambda k) \\
(\text{parameterize } ([\text{current-abort } k]) \\
(\lambda () (\text{body-thunk}) (\text{loop}))))))
\]

- abort also separates ‘read’ and ‘write’ access
Syntax Parameters

The same solution of an adjustable binding carries over to the syntax world.

➤ Prefer `syntax-parameterize` over `fluid-let-syntac`x for similar reasons.
Syntax Parameters

The same solution of an adjustable binding carries over to the syntax world.

- Prefer `syntax-parameterize` over `fluid-let-syntax` for similar reasons.

```scheme
(define-syntax-parameter abort
  (syntax-rules ()
    ()))

(define-syntax forever
  (syntax-rules ()
    (forever body ...)
    (call/cc
     (lambda (abort-k)
      (syntax-parameterize
       (let loop ()
        (body ... (loop))))))))
```
Syntax Parameters

The same solution of an adjustable binding carries over to the syntax world.

Prefer **syntax-parameterize** over **fluid-let-syntax** for similar reasons.

```
(define-syntax-parameter abort
  (syntax-rules ()))

(define-syntax forever
  (syntax-rules ()
    [[(forever body ...)
      (call/cc
       (lambda (abort-k)
         (syntax-parameterize
           ([abort
             ; or `make-rename-transformer'
             (syntax-rules () [[(_) (abort-k)]]))
           (let loop () body ... (loop))))))))
```

Everything “just works” now.
Conclusions

- Very convenient
- Modular macros, abstract both macros and on syntax parameters (e.g., a macro that abstracts over `abort`)
- Used extensively in Racket
- Like `syntax-rules` — covers many more cases, but there are still uses for unhygienic macros
Subtleties I

; Two seemingly identical abstractions
(define a (lambda () (abort)))
(define-syntax a (syntax-rules () [(_ (abort))]))

> (forever
   (define a (lambda () (abort)))
   (forever (display "inner\n") (a))
   (display "outer\n")
   (abort))
inner

> (forever
   (define-syntax a (syntax-rules () [(_ (abort)]))
   (forever (display "inner\n") (a))
   (display "outer\n")
   (abort))
inner
outer
(define-syntacten-times
 (syntax-rules ()
  [(_ body ...)
   (let loop ([n 10])
     (when (> n 0) body ... (loop (- n 1)))]))

; Refactor
(define-synacten-times
 (syntax-rules ()
  [(_ body ...)
   (let ([n 10])
     (forever body ...
       (set! n (- n 1))
       (when (= n 0) (abort)))]))

> (forever (ten-times (display "hey\n") (abort)))
; loops forever
(define-syntax (ten-times stx)
  (syntax-case stx ()
    [(_ body ...)
      (with-syntax ([old (syntax-parameter-value #'abort)])
        #'(let ([n 10])
          (forever (syntax-parameterize ([abort old]) body ...)
            (set! n (- n 1))
            (when (= n 0) (abort)))))))))
Conclusions II

• Very convenient

• Modular macros, abstract both macros and on syntax parameters (eg, a macro that abstracts over `abort`)

• Used extensively in Racket

• Like `syntax-rules` — covers many more cases, but there are still uses for unhygienic macros

• Need to be aware of subtleties, but still better for newbies, and easy to get an intuition for experienced macro writers.