First-class continuations

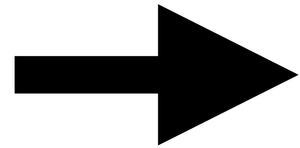
call/cc, stack-passing CEK machines

But first…

Assignment 2

```
e ::= (letrec* ([x e] ...) e)
    | (letrec ([x e] ...) e)
    | (let* ([x e] ...) e)
    | (let ([x e] ...) e)
    | (let x ([x e] ...) e)
    | (lambda (x ...) e)
    | (lambda x e)
    | (lambda (x ...+ . x) e)
    | (dynamic-wind e e e)
    | (guard (x cond-clause …)
        e)
    | (raise e)
    | (delay e)
    | (force e)
    | (and e ...)
    | (or e ...)
    | (cond cond-clause ...)
    | (case e case-clause ...)
```

```
    | (case e case-clause ...)
    | (if e e e)
    | (when e e)
    | (unless e e)
    | (set! x e)
    | (begin e ...+)
    | (call/cc e)
    | (apply e e)
    | (e e ...)
    | x
    | op
    | (quote dat)
```



```
e ::= (let ([x e] ...) e)
    | (lambda (x ...) e)
    | (lambda x e)
    | (apply e e)
    | (e e ...)
    | (prim op e …)
    | (apply-prim op e)
    | (if e e e)
    | (set! x e)
    | (call/cc e)
    | x
    | (quote dat)
```

utils.rkt

```
prim?
reserved?

scheme-exp?
ir-exp?

eval-scheme
eval-ir
```

Write your own tests

./tests/*/mytest.scm

```
(require “utils.rkt”)
(require “desugar.rkt”)

(define scm (read (open-input-file “…”)))
(scheme-exp? scm)
scm

(define ir (desugar scm))
(ir-exp? ir)
ir

(eval-ir ir)
```

start-0

(solved with only prims and `quote`)

```
(+ ‘5 ‘6)
```

```
(prim + ‘5 ‘6)
```

start-1

(solved once you add forms in both langs such as `let`, …)

```
(let ([x ‘1]
  (let ([_ (set! x ‘2)])
    …
    (+ x y) …))
```

```
(unless e₀ e₁)
```

```
(if e₀ (void) e₁)
```

```
(case e_k [(d_0 d_1) e_bdy] clauses …)
```

```
(let ([t e_k])
  (if (memv t ‘(d_0 d_1))
      e_bdy
      (case t clauses …)))
```

promises

(delay e) | (force e) | promise?

- Delay wraps its body in a thunk to be executed later by a force form. A promise is returned.
- Prim promise? should desugar correctly.
- Forcing a promise evaluates and *saves* the value.

call/cc

```
((((λ (u) (u u)) (λ (a) a)) e₁) e₀)
```

```
((((λ (u) (u u)) (λ (a) a)) e₁) e₀)
```

$\mathscr{E}$ = `((□ e₁) e₀)`

r = `((λ (u) (u u)) (λ (a) a))`

```
((((λ (u) (u u)) (λ (a) a)) e₁) e₀)
```

$\mathscr{E}$ = ((□ e₁) e₀)

r = ((λ (u) (u u)) (λ (a) a))

```
(λ (z) (((λ (u) (u u)) (λ (a) a)) z))
```

η-expansion & thunking

allows us to take hold of a suspended first-class computation (a call site) we may apply later.

call/cc

allows us to take hold of a suspended first-class computation (*a return point*) we may apply later.

```
(((call/cc (λ (k) (k (λ …)))) e₁) e₀)
```

$\mathcal{E}$ = ((□ e₁) e₀)

r = (call/cc (λ (k) (k (λ …))))

→ ((λ (k) (k (λ …))) (λ (□) ((□ e₁) e₀)))

→ ((λ (□) ((□ e₁) e₀)) (λ …))

→ (((λ …) e₁) e₀)))

Example 1.
Preemptive function return

```
(define (fun x)
  (let ([y (if (p? x)
               …
               …)])
    (g x y)))
```

```
(define (fun x)
  (call/cc (lambda (return)

    (let ([y (if (p? x)
                  …
                  (return x))])

      (g x y)))))
```

Example 2.

Coroutines / cooperative threading

Suggested exercise.

```
(lambda (yield)
  (let loop ([n 0])
      (yield n)
      (loop (+ n 1))))
```

```
(define (coroutine->gen co)
  (define resume co)
  (lambda ()
    (call/cc (lambda (return)

      (define yield
        (lambda (v)
          (call/cc
            (lambda (r)
              (set! resume r)
              (return v)))))

      (resume yield)))))
```

Example 3.
Backtracking search

```
(let ([a (amb '(1 2 3 4 5 6))]
      [b (amb '(1 2 3 4 5 6))]
      [c (amb '(1 2 3 4 5 6))])

  ;(pretty-print `(trying ,a ,b ,c))

  (assert (= (+ (* a a) (* b b)) (* c c)))

  `(solution ,a ,b ,c))
```

```
(define (amb lst)

  (let ([cc (call/cc (lambda (u) (u u)))])

    (if (null? lst)
        (fail)
        (let ([head (car lst)])
          (set! lst (cdr lst))
          (set! ccstack (cons cc ccstack))
          head))))
```

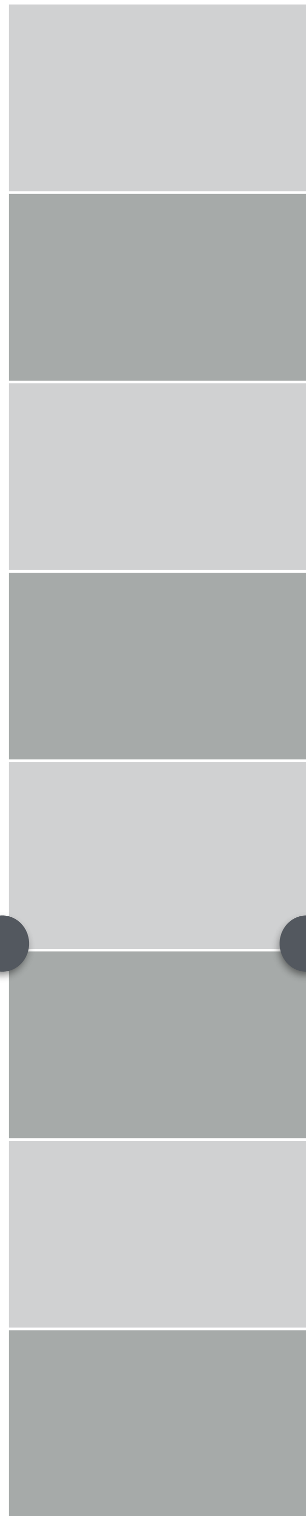
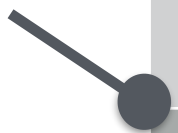
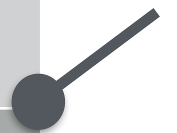
```
(define ccstack ‘())

(define (fail)
  (if (null? ccstack)
      (error 'no-solution)
      (let ([next-cc (car ccstack)])
        (set! ccstack (cdr ccstack))
        (next-cc next-cc))))

(define (assert t)
  (if t (void) (fail)))
```

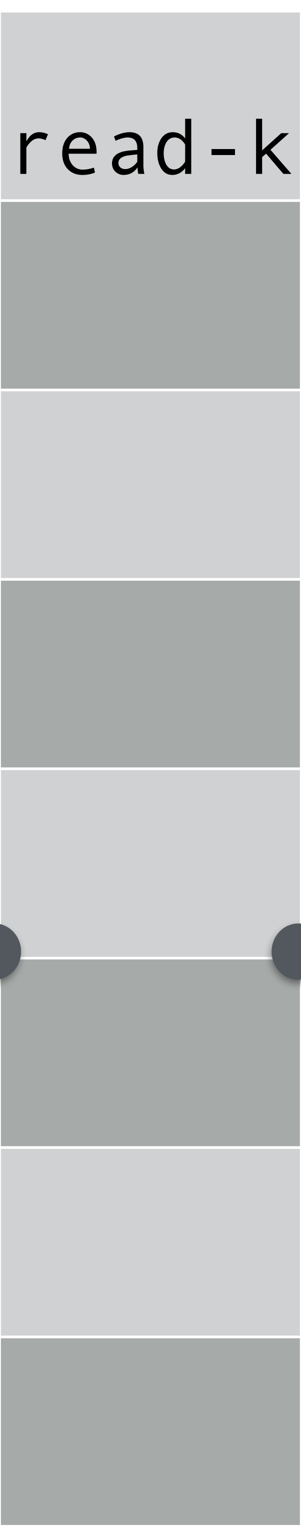
dynamic-wind & call/cc

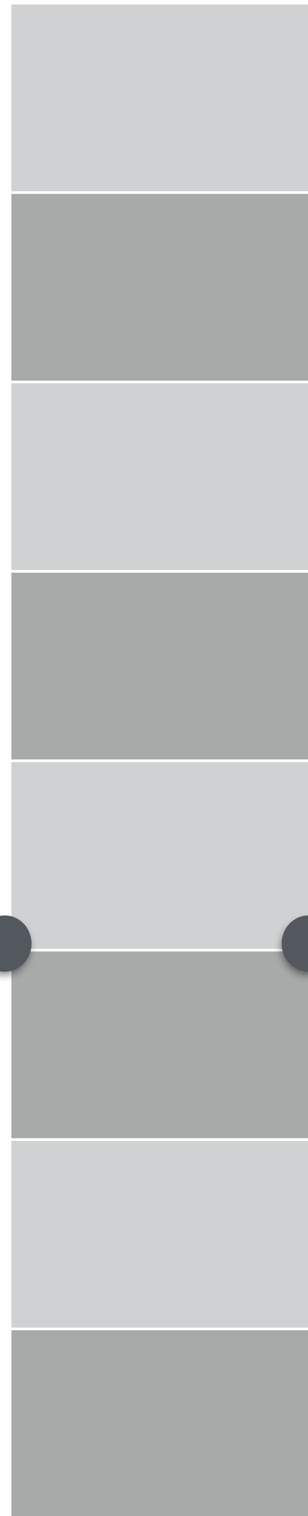
```
(dynamic-wind e0 e1 e2)
```

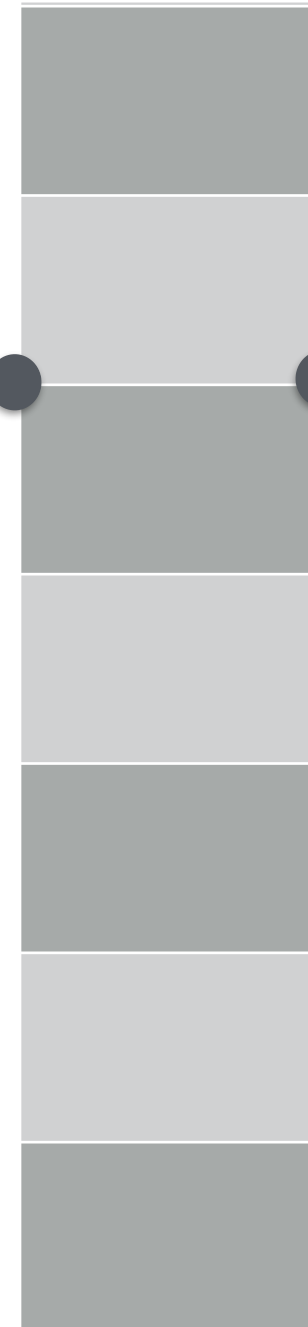
```
(dynamic-wind
  (lambda () entry code)
  (lambda () body)
  (lambda () exit code))
```

call/cc


call/cc
read-k
opens source file and scans to last position
closes file


call/cc
opens source file and
scans to last position
closes file


(read-k
write-k)
opens target file and
scans to last position
closes file

Exceptions

```
(guard [x cond-clause …] e)  | (raise e)
```

- Guard obtains the current continuation and then uses dynamic wind to set / unset a current handler
- Raise simply invokes the current handler on a value
- Guard's continuation should be outside the dynamic-wind so repeated raises don't infinite loop!
- Wrap exceptions in a cons cell if using this idiom:

```
(raise e)   =>   (%handler (cons e '()))


(guard [x clauses…] body)   =>

(let ([cc (call/cc (lambda (k) k))])
  (if (cons? cc)
      ; handle the raised exception
      (dynamic-wind
        setup-new-handler
        (lambda () body)
        revert-to-old-handler)))
```

Stack-passing (CEK) semantics

C Control-expression

Term-rewriting / textual reduction

Context and redex for deterministic eval

CE Control & Env machine

Big-step, explicit closure creation

CES Store-passing machine

Passes addr->value map in evaluation order

CEK Stack-passing machine

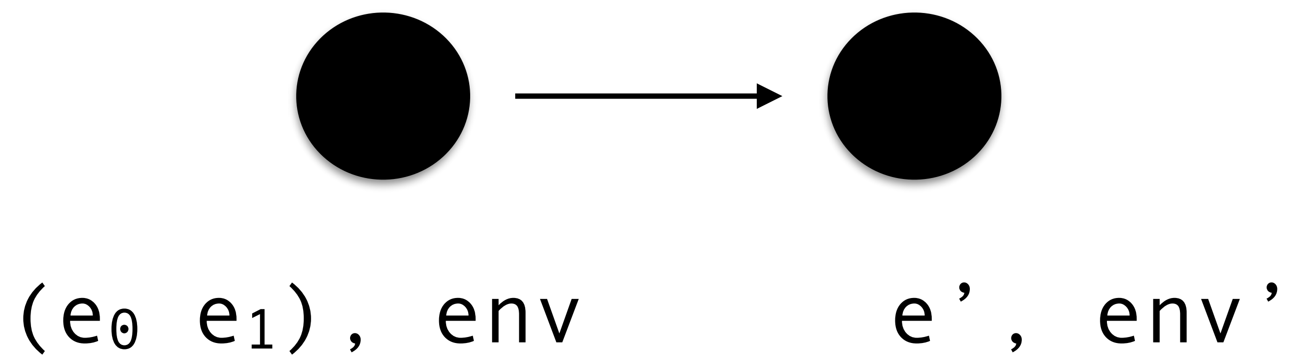
Passes a list of stack frames, small-step

$$\frac{(\texttt{e}_0, \text{env}) \Downarrow (\texttt{(λ (x) e}_2\texttt{)}, \text{env'}) \qquad (\texttt{e}_1, \text{env}) \Downarrow v_1 \qquad (\texttt{e}_2, \text{env'}[x \mapsto v_1]) \Downarrow v_2}{(\texttt{(e}_0\ \texttt{e}_1\texttt{)}, \text{env}) \Downarrow v_2}$$

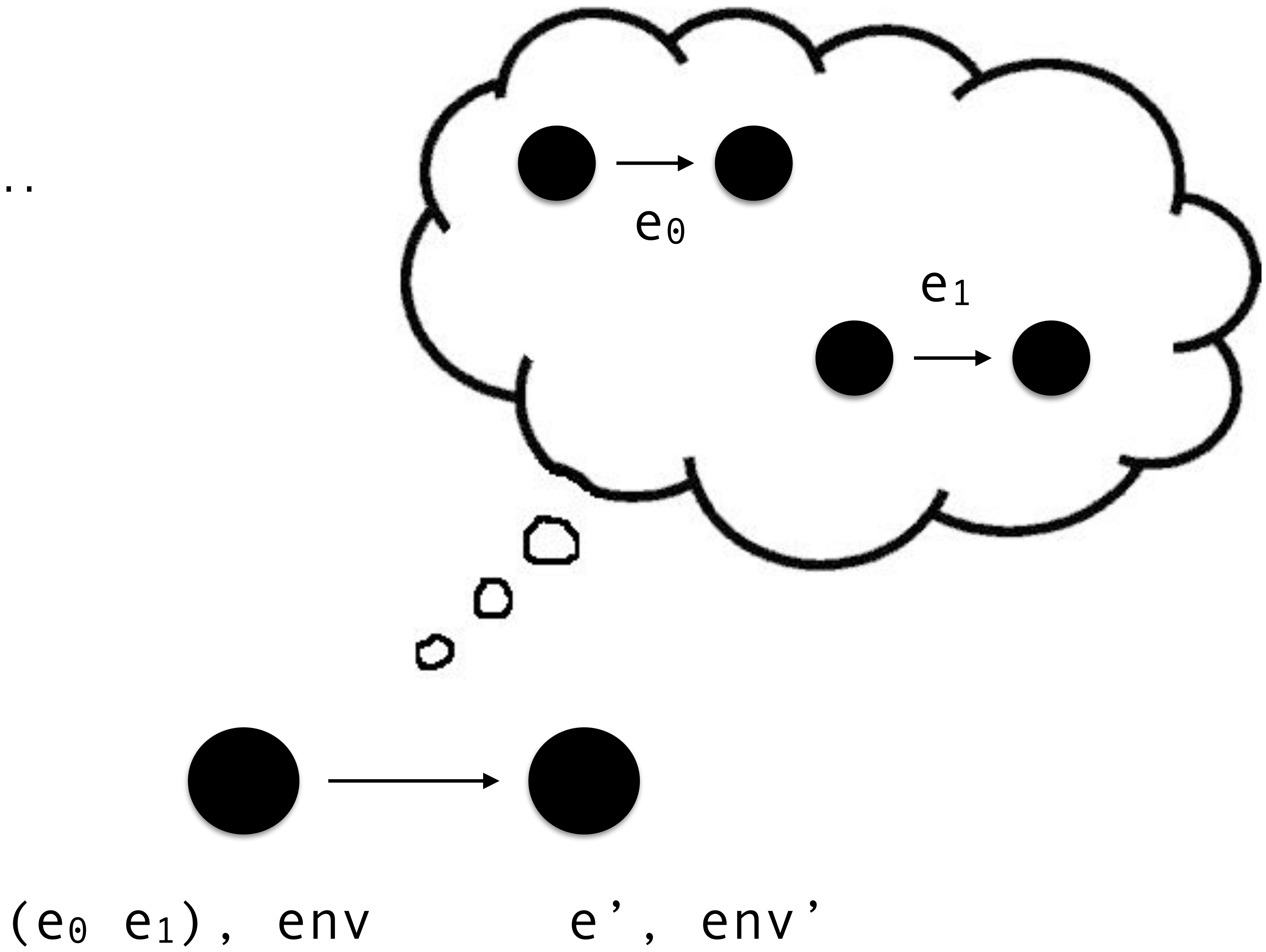
$$\frac{}{(\texttt{(λ (x) e)}, \text{env}) \Downarrow (\texttt{(λ (x) e)}, \text{env})}$$

$$\frac{}{(x, \text{env}) \Downarrow \text{env}(x)}$$

Previously…



Previously…



```
e ::= (λ (x) e)
    | (e e)
    | x
    | (call/cc (λ (x) e))
```

k ::= () | **ar**(e, env, k)

| **fn**(v, k)

```
e ::= (λ (x) e)
    | (e e)
    | x
    | (call/cc (λ (x) e))
```

k ::= () | **ar**(e, env, k)

| **fn**(v, k)

```
e ::= (λ (x) e)
   | (e e)
   | x
   | (call/cc (λ (x) e))
```

$\mathscr{E}$ ::= ($\mathscr{E}$ e)

| (v $\mathscr{E}$)

| □

$$((e_0\ e_1), env, k) \rightarrow (e_0, env, \mathbf{ar}(e_1, env, k))$$

$$(x, env, \mathbf{ar}(e_1, env_1, k_1)) \rightarrow (e_1, env_1, \mathbf{fn}(env(x), k_1))$$

$$((\lambda\ (x)\ e), env, \mathbf{ar}(e_1, env_1, k_1)) \rightarrow (e_1, env_1, \mathbf{fn}(((\lambda\ (x)\ e), env), k_1))$$

$$(x, env, \mathbf{fn}(((\lambda\ (x_1)\ e_1), env_1), k_1)) \rightarrow (e_1, env_1[x_1 \mapsto env(x)], k_1)$$

$$((\lambda\ (x)\ e), env, \mathbf{fn}(((\lambda\ (x_1)\ e_1), env_1), k_1))$$
$$\rightarrow (e_1, env_1[x_1 \mapsto ((\lambda\ (x)\ e), env)], k_1)$$

call/cc semantics

$$((\texttt{call/cc}\ (\lambda\ (x)\ e_0)), env, k) \rightarrow (e_0, env[x \mapsto k], k)$$

$$((\lambda\ (x)\ e_0), env, \mathbf{fn}(k_0, k_1)) \rightarrow ((\lambda\ (x)\ e_0), env, k_0)$$

$$(x, env, \mathbf{fn}(k_0, k_1)) \rightarrow (x, env, k_0)$$

```
e ::= ... | (let ([x e0]) e1)
```

k ::= … | **let**(x, e, env, k)

$(x, env, \textbf{let}(x_1, e_1, env_1, k_1)) \rightarrow (e_1, env_1[x_1 \mapsto env(x)], k_1)$

$((\lambda\ (x)\ e), env, \textbf{let}(x_1, e_1, env_1, k_1)) \rightarrow (e_1, env_1[x_1 \mapsto ((\lambda\ (x)\ e), env)], k_1)$

`e ::= ...`

`| (let ([x e₀]) e₁)`

$(x, env, \mathbf{fn}((\texttt{(λ (x}_1\texttt{) e}_1\texttt{)}, env_1), k_1)) \rightarrow (e_1, env_1[x_1 \mapsto env(x)], k_1)$

$(\texttt{(λ (x) e)}, env, \mathbf{fn}((\texttt{(λ (x}_1\texttt{) e}_1\texttt{)}, env_1), k_1))$
$\rightarrow (e_1, env_1[x_1 \mapsto (\texttt{(λ (x) e)}, env)], k_1)$

$k ::= \ \dots \ | \ \mathbf{let}(x, e, env, k)$

$(x, env, \mathbf{let}(x_1, e_1, env_1, k_1)) \rightarrow (e_1, env_1[x_1 \mapsto env(x)], k_1)$

$(\texttt{(λ (x) e)}, env, \mathbf{let}(x_1, e_1, env_1, k_1)) \rightarrow (e_1, env_1[x_1 \mapsto (\texttt{(λ (x) e)}, env)], k_1)$

CEK-machine evaluation

$$
\begin{aligned}
(e_0, [], ()) &\to \ldots \\
&\to \ldots \\
&\to \ldots \\
&\to \ldots \\
&\to (x, env, ()) \to env(x)
\end{aligned}
$$

Implementing dynamic-wind

```
; Finds the maximum shared tail of two lists
(define (%common-tail st0 st1)
  (let ([lx (length x)]
        [ly (length y)])
    (let loop
         ([x (if (> lx ly) (drop x (- lx ly)) x)]
          [y (if (> ly lx) (drop y (- ly lx)) y)])
      (if (eq? x y)
        x
        (loop (cdr x) (cdr y))))))
```

```
; Winds down old stack and up new stack,
; invoking the proper post and then pre thunks as it winds
(define (%do-wind new-stack)
  (unless (eq? new-stack %wind-stack)
    (let ([tail (%common-tail new-stack
                              %wind-stack)])
      (let loop ([st %wind-stack])
        (unless (eq? st tail)
          (set! %wind-stack (cdr st))
          ((cdr (car st)))
          (loop (cdr st))))
      (let loop ([st new-stack])
        (unless (eq? st tail)
          (loop (cdr st))
          ((car (car st)))
          (set! %wind-stack st)))))))
```

```
(define %wind-stack ‘())

(define (dynamic-wind pre body post)
  (pre)
  (set! %wind-stack (cons (cons pre post)
                          %wind-stack))
  (let ([val (body)])
    (set! %wind-stack (cdr %wind-stack))
    (post)
    v))
```

```
(define (desugar-t e)
  (match e
    ...




    [`(call/cc ,e0)
     `(call/cc
        ,(desugar-t e0))]

    ...))
```

```
(define (desugar-t e)
  (match e
    ...




    [`(call/cc ,e0)
     `(call/cc
       ,(desugar-t
         `(lambda (k)
           (,e0 (lambda (x) (k x))))))]

    ...))
```

```
(define (desugar-t e)
  (match e
    ...

    ; desugar call/cc so that each use saves the stack &
    ; wraps resulting continuation with a call to %do-wind
    [`(call/cc ,e0)
     `(call/cc
       ,(desugar-t
         `(lambda (k)  ; save k and k's stack
           (,e0 (let ([k-stack %wind-stack])
                  (lambda (x)
                    (begin (%do-wind k-stack)
                           (k x))))))))]

    ...))
```