Adding Mutation

Extending our interpreted language with Scheme's `set!`

```
(define (sum from to)
  (define total 0)
  (let loop ()
    (set! total (+ total from))
    (set! from (+ from 1))
    (if (<= from to)
        (loop)
        total)))
```

```scheme
(define (sum from to)
  (begin
    (define total 0)
    (let loop ()
      (begin
        (set! total (+ total from))
        (set! from (+ from 1))
        (if (<= from to)
            (loop)
            total)))))
```

```
(let loop ([x ivx] [y ivy] …)
  body)
```

```
(letrec ([loop (lambda (x y …)
                 body)])
  (loop ivx ivy …))
```

```
(define (sum from to)
  (define total 0)
  (let loop ()
    (set! total (+ total from))
    (set! from (+ from 1))
    (if (<= from to)
        (loop)
        total)))
```

```
(define (sum from to)
  (let loop ([i from]
             [total 0])
    (if (<= i to)
        (loop (+ i 1) (+ total i))
        total)))
```

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

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

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

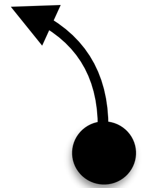
(x, env, st)

env maps variables to addresses

st maps addresses to values

The current size of the store is the next address: |st|

$$(\texttt{e}_2, env'[x \mapsto |st_2|], st_2[|st_2| \mapsto v_1]) \Downarrow (v_2, st_3)$$



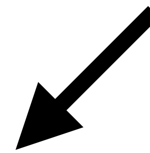
$$\frac{(\texttt{e}_0, env, st_0) \Downarrow (((\texttt{λ (x) e}_2), env'), st_1) \quad (\texttt{e}_1, env, st_1) \Downarrow (v_1, st_2)}{((\texttt{e}_0\ \texttt{e}_1), env, st_0) \Downarrow (v_2, st_3)}$$

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

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

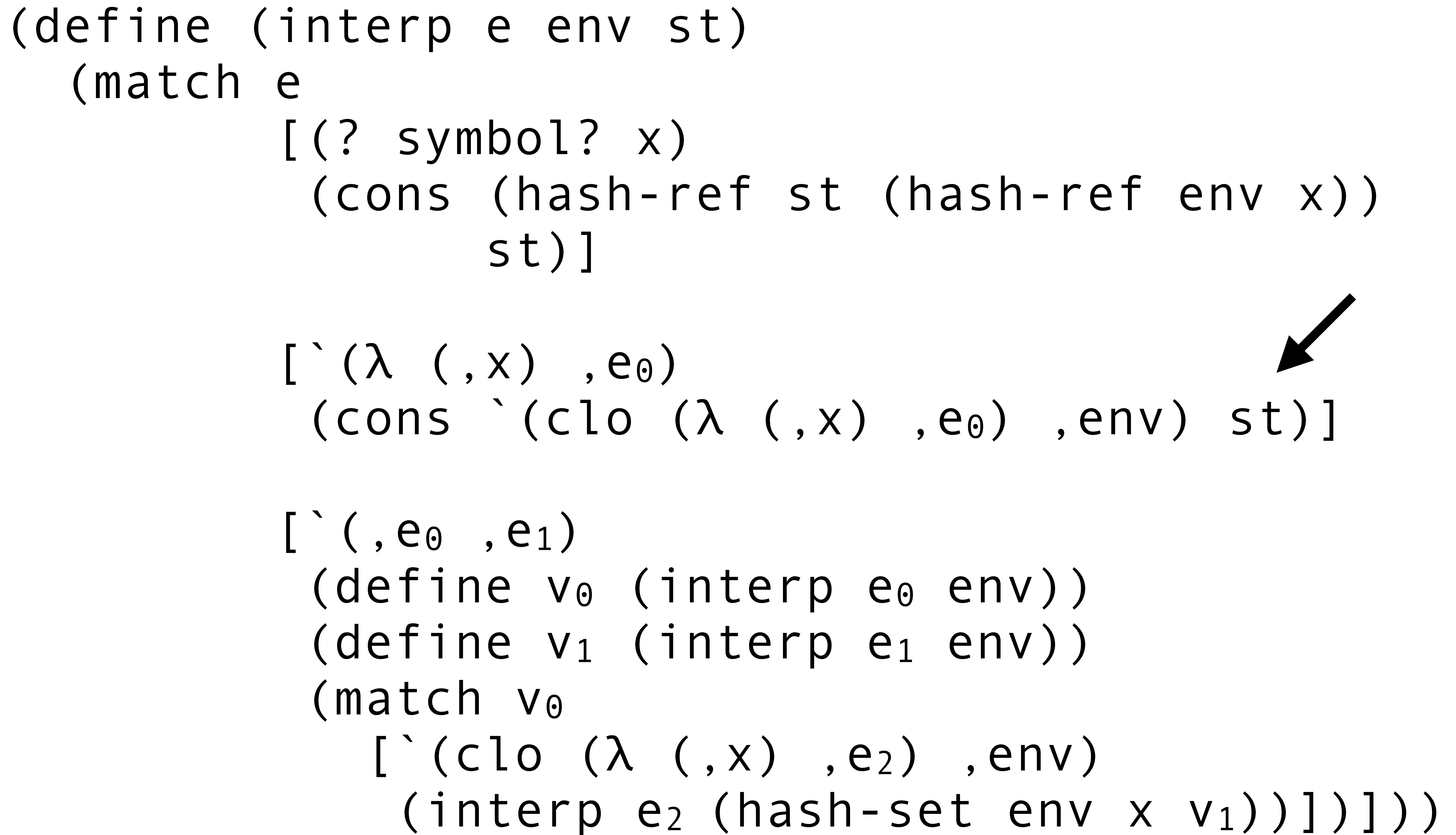
Store-passing style

```
(define (interp e env)
  (match e
         [(? symbol? x)
          (hash-ref env x)]

         [`(λ (,x) ,e0)
          `(clo (λ (,x) ,e0) ,env)]

         [`(,e0 ,e1)
          (define v0 (interp e0 env))
          (define v1 (interp e1 env))
          (match v0
            [`(clo (λ (,x) ,e2) ,env)
             (interp e2 (hash-set env x v1))])]))
```

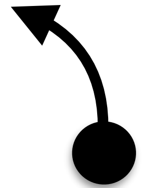


```
(define (interp e env st)
  (match e
         [(? symbol? x)
          (hash-ref env x)]

         [`(λ (,x) ,e₀)
          `(clo (λ (,x) ,e₀) ,env)]

         [`(,e₀ ,e₁)
          (define v₀ (interp e₀ env))
          (define v₁ (interp e₁ env))
          (match v₀
            [`(clo (λ (,x) ,e₂) ,env)
             (interp e₂ (hash-set env x v₁))])]))
```

```
(define (interp e env st)
  (match e
         [(? symbol? x)
          (cons (hash-ref st (hash-ref env x))
                st)]

         [`(λ (,x) ,e₀)
          `(clo (λ (,x) ,e₀) ,env)]

         [`(,e₀ ,e₁)
          (define v₀ (interp e₀ env))
          (define v₁ (interp e₁ env))
          (match v₀
            [`(clo (λ (,x) ,e₂) ,env)
             (interp e₂ (hash-set env x v₁))])]))
```


(define (interp e env st)
(match e
[(? symbol? x)
(cons (hash-ref st (hash-ref env x))
st)]
[`(λ (,x) ,e0)
(cons `(clo (λ (,x) ,e0) ,env) st)]
[`(,e0 ,e1)
(define v0 (interp e0 env))
(define v1 (interp e1 env))
(match v0
[`(clo (λ (,x) ,e2) ,env)
(interp e2 (hash-set env x v1))])]))

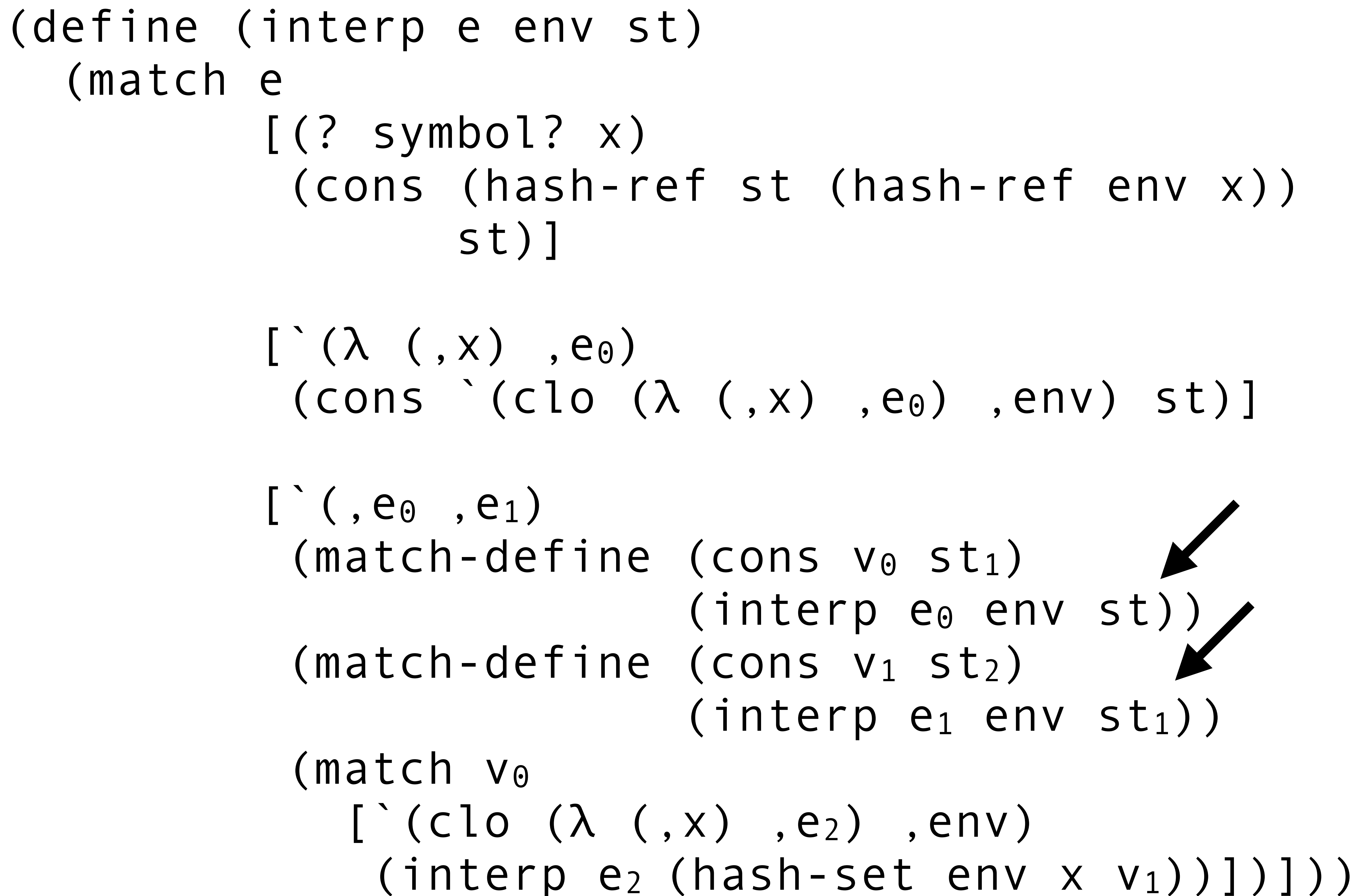
$$(\texttt{e}_2, env'[x \mapsto |st_2|], st_2[|st_2| \mapsto v_1]) \Downarrow (v_2, st_3)$$



$$\frac{(\texttt{e}_0, env, st_0) \Downarrow (((\lambda\ (x)\ \texttt{e}_2), env'), st_1) \quad (\texttt{e}_1, env, st_1) \Downarrow (v_1, st_2)}{((\texttt{e}_0\ \texttt{e}_1), env, st_0) \Downarrow (v_2, st_3)}$$

$$\frac{}{((\lambda\ (x)\ \texttt{e}), env, st) \Downarrow (((\lambda\ (x)\ \texttt{e}), env), st)}$$

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


(define (interp e env st)
(match e
[(? symbol? x)
(cons (hash-ref st (hash-ref env x))
st)]
[`(λ (,x) ,e0)
(cons `(clo (λ (,x) ,e0) ,env) st)]
[`(,e0 ,e1)
(match-define (cons v0 st1)
(interp e0 env st))
(match-define (cons v1 st2)
(interp e1 env st1))
(match v0
[`(clo (λ (,x) ,e2) ,env)
(interp e2 (hash-set env x v1))])]))

```
(define (interp e env st)
  (match e
         [(? symbol? x)
          (cons (hash-ref st (hash-ref env x))
                st)]

         [`(λ (,x) ,e0)
          (cons `(clo (λ (,x) ,e0) ,env) st)]

         [`(,e0 ,e1)
          (match-define (cons v0 st1)
                        (interp e0 env st))
          (match-define (cons v1 st2)
                        (interp e1 env st1))
          (match v0
            [`(clo (λ (,x) ,e2) ,env)
             (define addr (hash-count st2))
             (interp e2
                     (hash-set env x addr)
                     (hash-set st2 adds v1))])]))
```

Let’s code this up.