

# 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)))
```

```
(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)))
```

$$(e_0, env) \Downarrow ((\lambda (x) e_2), env') \quad (e_1, env) \Downarrow v_1 \quad (e_2, env'[x \mapsto v_1]) \Downarrow v_2$$


---

$$((e_0 e_1), env) \Downarrow v_2$$

---


$$((\lambda (x) e), env) \Downarrow ((\lambda (x) e), env)$$

---


$$(x, env) \Downarrow env(x)$$

$(x, \text{env}, \text{st})$

**env** maps variables to addresses

**st** maps addresses to values

The current size of the store is the next address:  $|\text{st}|$



$$(e_2, \text{env}'[x \mapsto |st_2|], st_2[|st_2| \mapsto v_1]) \Downarrow (v_2, st_3)$$



$$(e_0, \text{env}, st_0) \Downarrow ((\lambda (x) e_2), \text{env}', st_1) \quad (e_1, \text{env}, st_1) \Downarrow (v_1, st_2)$$


---

$$((e_0 e_1), \text{env}, st_0) \Downarrow (v_2, st_3)$$

---


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

---

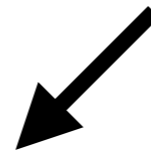

$$(x, \text{env}, st) \Downarrow (st(\text{env}(x)), st)$$

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))]
       [_)
      (interp e1 env)]))
```



```
(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))])))
```

```

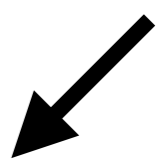
(define (interp e env st)
  (match e
    [(? symbol? x)
     (cons (hash-ref st (hash-ref env x))
           st)]
    [`(λ (, x) , e0)
     `(clo (λ (, x) , e0) , env)]
    [`(, e0 , e1)
     (define v0 (interp e0 env))
     (define v1 (interp e1 env))
     (match v0
       [`(clo (λ (, x) , e2) , env)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

```

```
(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))])))
```



$$(e_2, \text{env}'[x \mapsto |st_2|], st_2[|st_2| \mapsto v_1]) \Downarrow (v_2, st_3)$$



$$(e_0, \text{env}, st_0) \Downarrow ((\lambda (x) e_2), \text{env}', st_1) \quad (e_1, \text{env}, st_1) \Downarrow (v_1, st_2)$$

---


$$((e_0 e_1), \text{env}, st_0) \Downarrow (v_2, st_3)$$

---


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

---


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

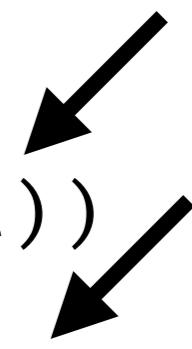
```

(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))]
       [ _ ]))
  ))

```

The image shows a Scheme code snippet for an interpreter. The code is a function `(define (interp e env st))` that uses a `match` expression to handle different forms of `e`. The first branch handles symbols, the second handles lambda expressions, and the third handles application expressions. In the application branch, there are two arrows: one pointing to the environment `env` argument of the inner `interp` call, and another pointing to the `adds` argument of the `hash-set` call that updates the state.

Let's code this up.