A) this.x = 3

B) o.x = 3

C) x = 3

object mutation

variable mutation

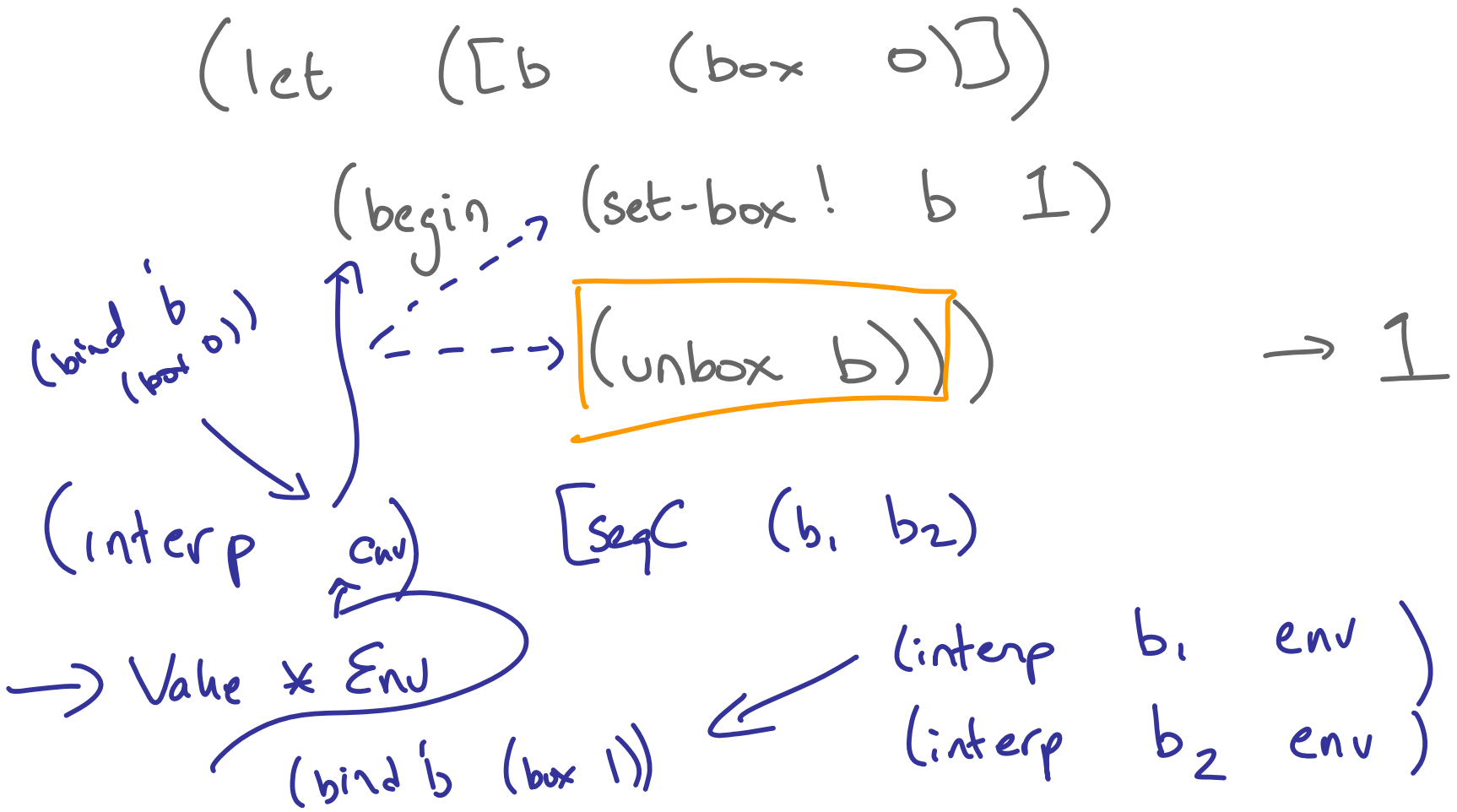
"side-effect"

Boxes

box : V → (boxof V)

unbox : (boxof V) → V

set-box! : (boxof V) * V → void


(let ([b (box 0)])
(begin (set-box! b 1)
(unbox b)))
→ 1
(bind 'b (box 0))
(interp
env)
[SeqC (b1 b2)
→ Value * Env
(bind 'b (box 1))
(interp b1 env)
(interp b2 env)

```
(let ([b (box 0)])
  (let ([c b])
    (begin
      (set-box! b 1)
      (unbox c))))  → 1
```

```
(begin (let ([b 3])
         b)
       b)
```

Env : Id → Loc

Store : Loc → Value

[box (e : ExprC)]

```
[unboxC (arg : ExprC)]
[setboxC (b : ExprC) (v : ExprC)]
[seqC (b1 : ExprC) (b2 : ExprC)]
```

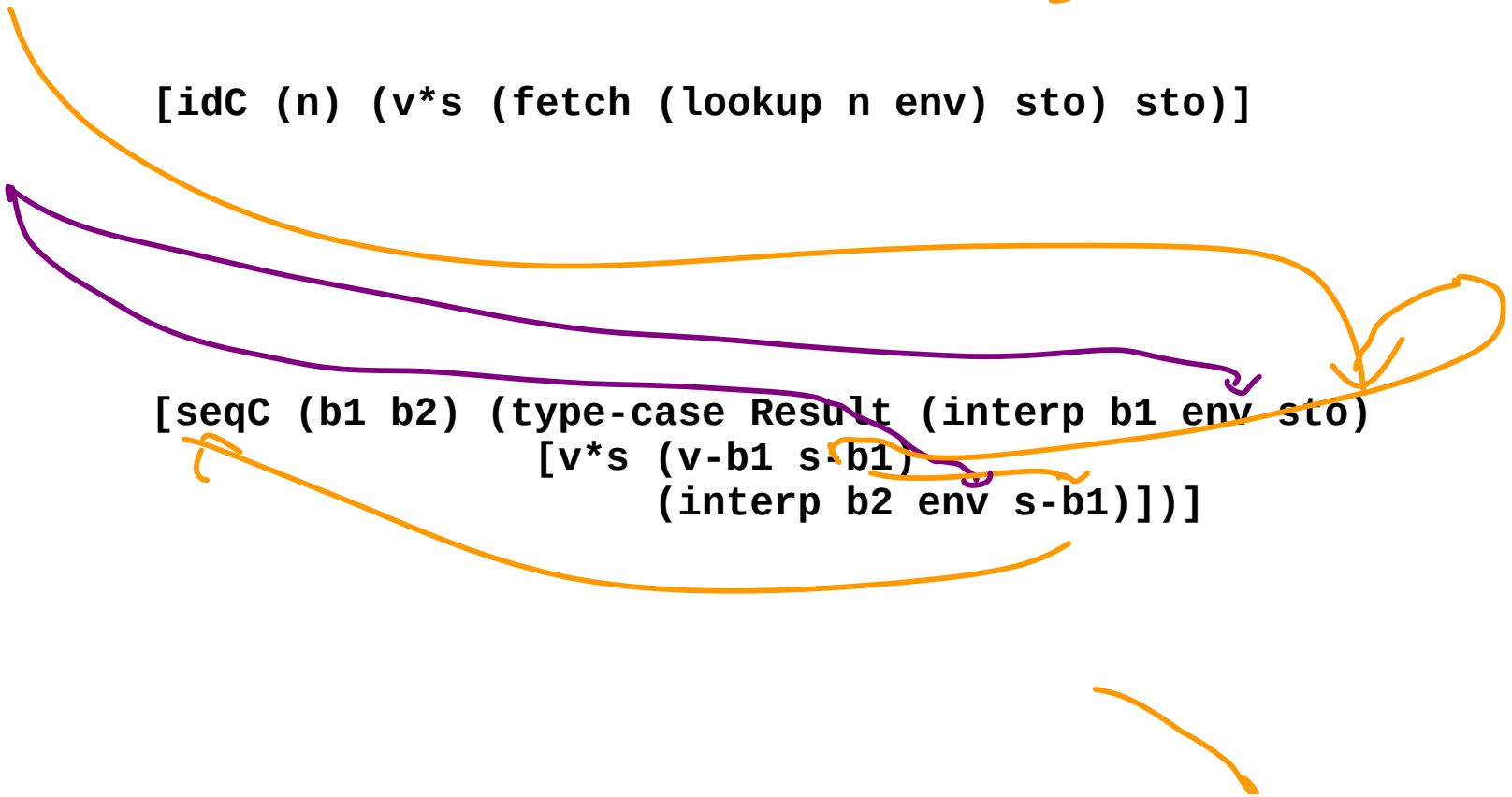
```
(define-type Result [v*s (v : Value) (s : Store)])

(define (interp [expr : ExprC] [env : Env] [sto : Store]) : Result
  ...)
```

```
[numC (n) (v*s (numV n) sto)]
[lamC (a b) (v*s (closV a b env) sto)]
```

store-passing style

```
[idC (n) (v*s (fetch (lookup n env) sto) sto)]
```

```
[seqC (b1 b2) (type-case Result (interp b1 env sto)
                [v*s (v-b1 s-b1)
                     (interp b2 env s-b1)])]
```



```
[plusC (l r) (type-case Result (interp l env sto)
               [v*s (v-l s-l)
                    (type-case Result (interp r env s-l)
                      [v*s (v-r s-r)
                           (v*s (num+ v-l v-r) s-r)])])]
```

```
[unboxC (a) (type-case Result (interp a env sto)
              [v*s (v-a s-a)
                   (v*s (fetch (boxV-l v-a) s-a) s-a)])]
```

(box (begin (set-box! - -) -))

```
[boxC (a) (type-case Result (interp a env sto)
            [v*s (v-a s-a)
                 (let ([where (new-loc)])
                   (v*s (boxV where)
                        (override-store (cell where v-a)
                                        s-a)))])]
```