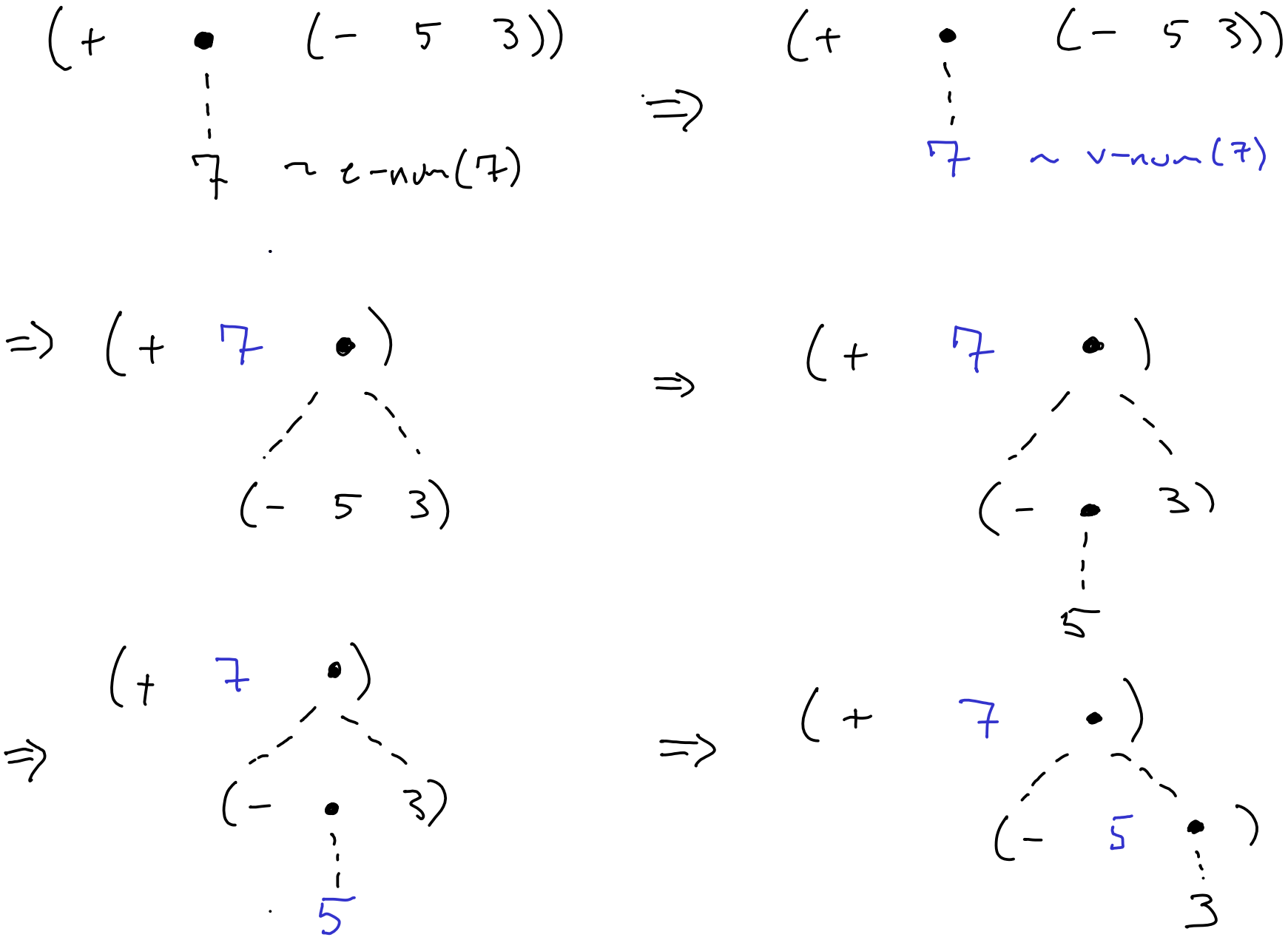
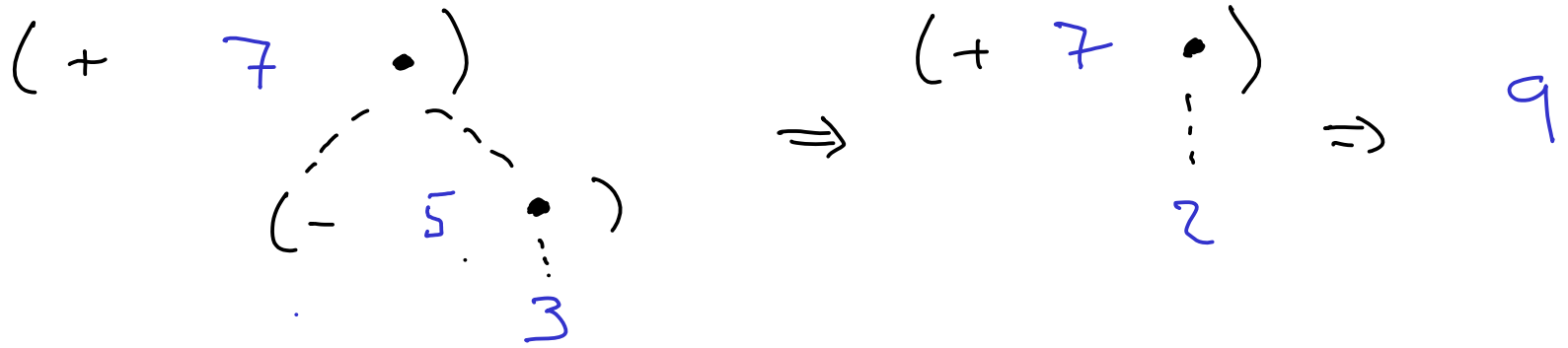

(+ • (- 5 3))
7 ~ e-num(7)
⇒
(+ • (- 5 3))
7 ~ v-num(7)
⇒ (+ 7 •)
(- 5 3)
⇒
(+ 7 •)
(- • 3)
5
⇒ (+ 7 •)
(- • 3)
5
⇒
(+ 7 •)
(- 5 •)
3



Contexts = (op • expr)

(op val •)

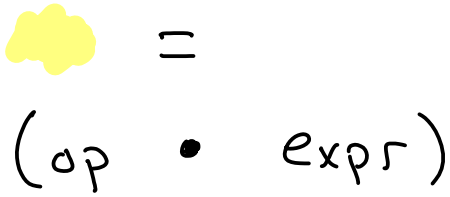
```
data Context:
  | c-op-left(op :: String,
               right :: Expr)
  | c-op-right(op :: String,
               left :: Value)
```

Program List<Context>, Value end

List<Context>, Expr

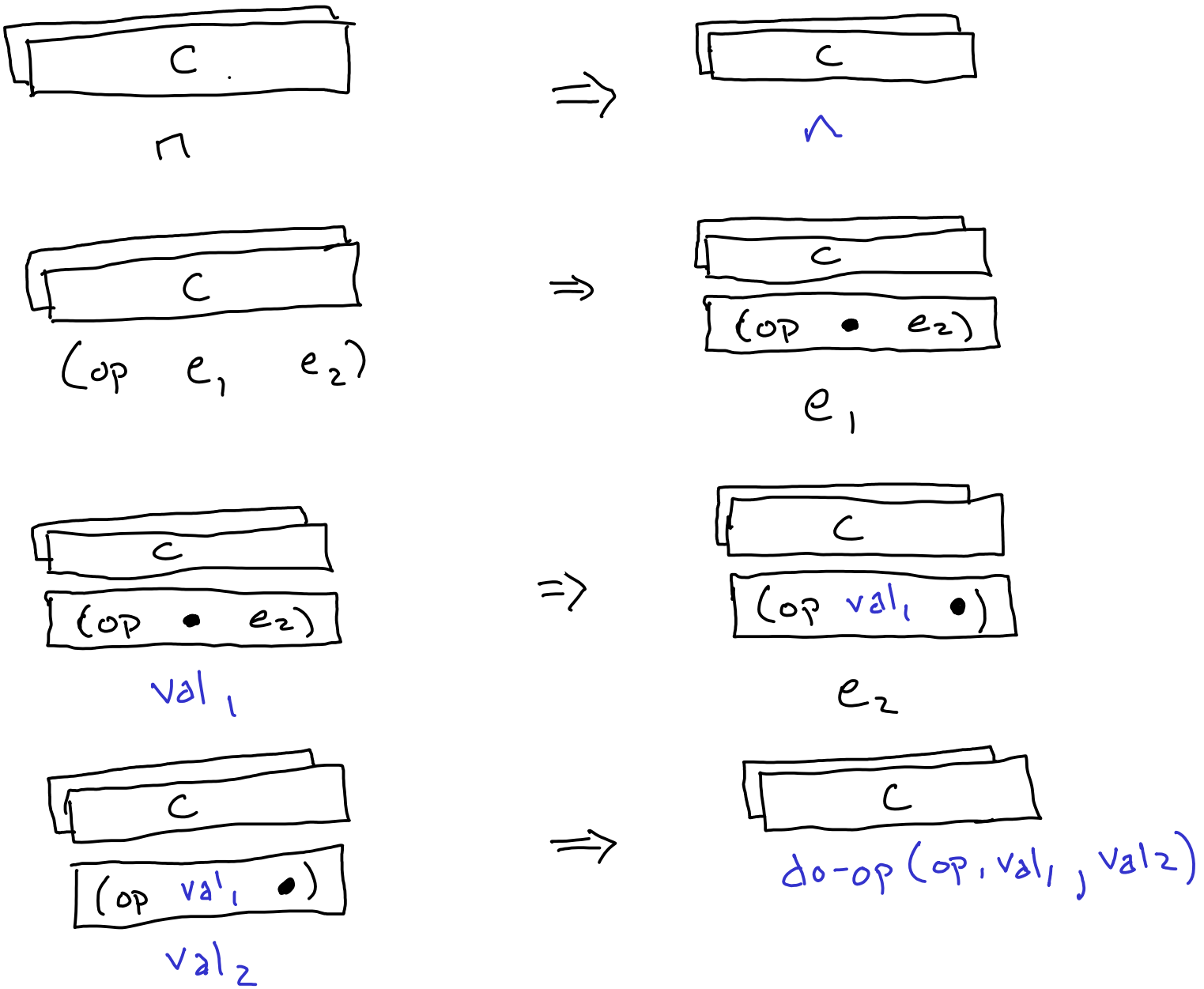
```
data Expr:
  | e-num(n :: Number)
  | e-op(op :: String, left :: Expr, right :: Expr)
end

type Value = Number

fun interp(e :: Expr) -> Value:
  cases(Expr) e:
    | e-num(n) => n
    | e-op(op, left, right) =>
      do-op(op, interp(left), interp(right))
  end
end

fun do-op(op :: String, l :: Value, r :: Value) -> Value:
  ...
end
```

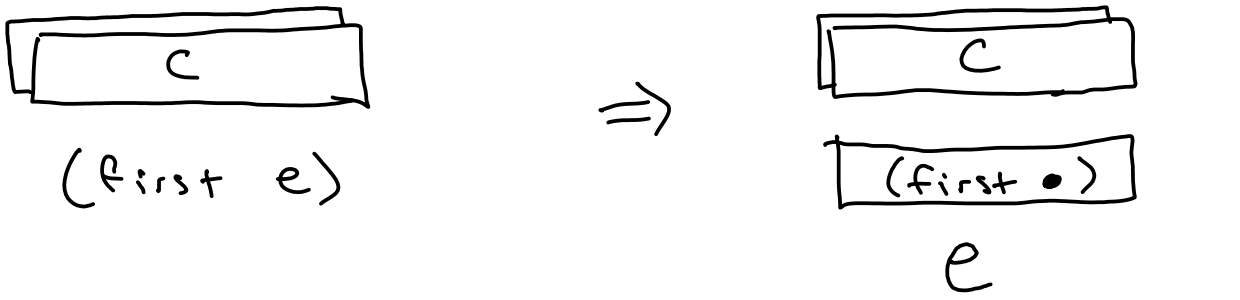
do-op(op, leftval, interp(right))

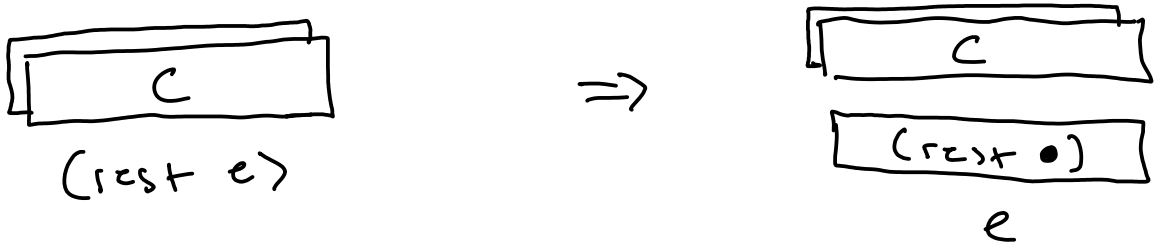


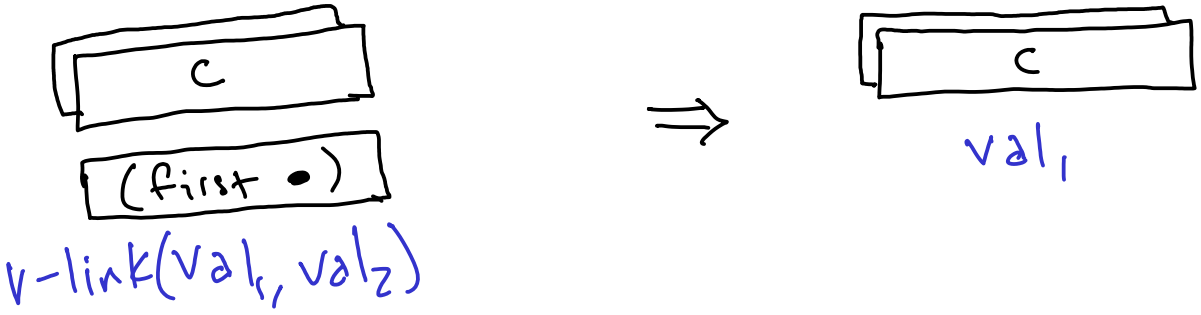
type Program = List<Context> Value or Expr

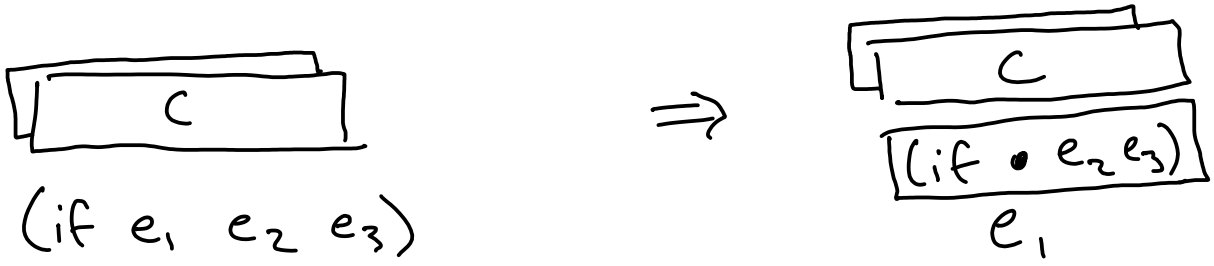
step Program → Program

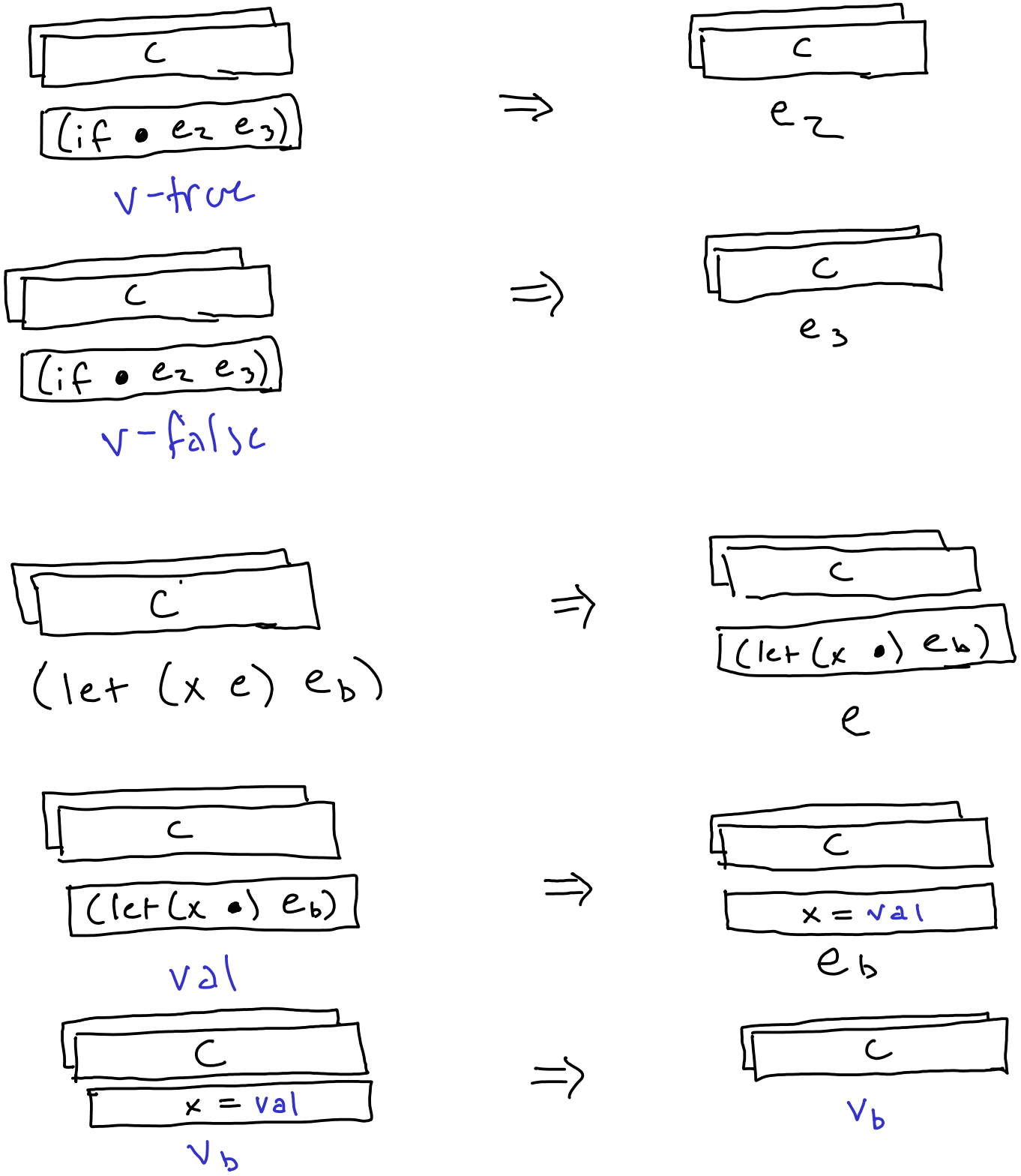

C
n
⇒
C
n
C
(op e1 e2)
⇒
C
(op • e2)
e1
C
(op • e2)
val1
⇒
C
(op val1 •)
e2
C
(op val1 •)
val2
⇒
C
do-op (op, val1, val2)

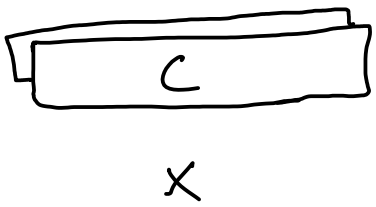

C
(first e)
⇒
C
(first •)
e


C
(rest e)
⇒
C
(rest •)
e


C
(first •)
v-link(val1, val2)
⇒
C
val1


C
(if e1 e2 e3)
⇒
C
(if • e2 e3)
e1


C
(if • e2 e3)
v-true
⇒
C
e2
C
(if • e2 e3)
v-false
⇒
C
e3
C
(let (x e) eb)
⇒
C
(let (x •) eb)
e
C
(let (x •) eb)
val
⇒
C
x = val
eb
C
x = val
vb
⇒
C
vb



x

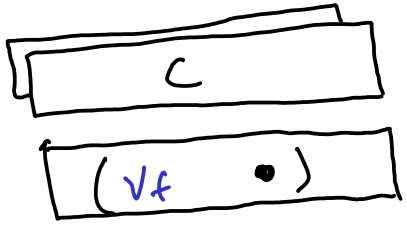
⇒

lookup(x, c)

(lam (x) e_b)

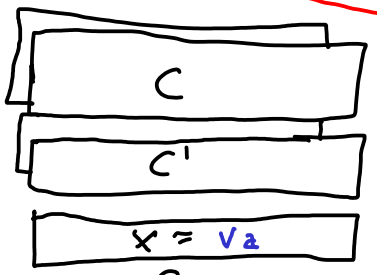
⇒

clos(c, x, e_b)



v_a

⇒



e_b

Do you see the issue?

(see next lecture, don't implement this ☺)

v_f = clos(c', x, e_b)