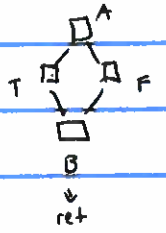
5-1

rco removes the "recursive" structure from R

R_2 (+ (if (< (read) 5) 17 (+ 8 (+ 9 10))) (+ (read) 7))

RLO_2

```
let x0 := read in
let x1 := if x0 < 5
          then 17
          else let x2 := 9 + 10 in
               let x3 := 8 + x2 in
               x3
in
let x4 := read in
let x5 := x4 + 7 in
x5
```




```
if (< read 5)
7
8
```

⇒

```
let x0 := read
in
if x0 < 5
7
8
```

let* : List (Pair Var Exp) × Expr → Expr

rco p : R_{progs} ⇒ R_{progs}

rco p (program i e) = (program i (rco e ∅ T e))

rco e : (X ⇒ E) Tail/Pos? e → e

rco e σ tail? e := let* nv e1
where (nv, e1) = rco a σ tail? e

rco a : (X ⇒ E) Tail/Pos? e → (List (Pair Var E)) × E (arg)

rco a σ tail? (var x) := (mt, σ(x))

rco a σ tail? (num n / bool b) := (mt, (num n / bool b))

(read) := ([(read-var, read)], read-var)

(una a) := (nva ++ [(unavar, (una a'))], unavar)

where (nva, a') = rco a σ F a

8-2/ rcoa σ tail? (bin e_L e_R) :=

(nv_L ++ nv_R ++ [(binvar, (bin a_L a_R))], binvar)

where (nv_L, a_L) = rcoa σ F e_L

(nv_R, a_R) = rcoa σ F e_R

rcoa σ tail? (let x := x_e in b_e) :=

(nv_x ++ nv_b , a_b)

where (nv_x, a_x) := rcoa σ F x_e

σ' := σ [x ↦ a_x]

(nv_b, a_b) := rcoa σ' tail? b_e

rcoa σ tail? (if e_C e_T e_F) :=

if tail? then

(nv_C, if')

else (nv_C ++ [(ifvar, if')], ifvar)

where (nv_C, cmp_C, a_L, a_R) :=

rco$_C$ σ e_C

if' := if (cmp_C a_L a_R)

then rcoe σ tail? e_T

else rcoe σ tail' e_F

rco$_C$: (X ⇒ E) E ⇒ (New vars, cmp op, A, A)

rco$_C$ σ (bin op e_L e_R) :=

(nv_L ++ nv_R, op, a_L, a_R)

where (nv_L, a_L) = rcoa σ F e_L

(nv_R, a_R) = rcoa σ F e_R

rco$_C$ σ (let x := x_e in b_e) :=

(nv_x ++ nv_b, op_b, a_L, a_R)

where (nv_x, a_x) = rcoa σ F x_e

(nv_b, op_b, a_L, a_R) = rco$_C$ σ' b_e

σ' = σ [x ↦ a_x]

rco$_C$ σ other := (nv, ~~bool~~ =, T, a)

where (nv, a) = rcoa σ F other

3-3/

econp : R ⇒ C

if (= 5 6)
  12
or let x = :
  ...

Block0 = ... goto-if (= 5 6) Block1 Block2
Block1 = return 12
Block2 = seq (set x = ...)
...

econp (program ; be) = (program ; L⇒T[BODY ↦ tb])
  where (L⇒T, tB) = econe be (λ (fa) (return fa))

econe k boring (var, num, bool) = (∅, k (econa boring))

econe k (let x := xe in be) = (L⇒T, (seq (set! x (econc xe)) bt))
  where (L⇒T, bt) = econe k be

econe k (if (cmp aL aR) eT eF) = (L⇒TT ++ L⇒TF ++ NEW,
  goto-if (cmp aL' aR') true-lab false-lab
  where (L⇒TT, TT) = econe k eT
  (L⇒TF, TF) = econe k eF
  aL' = econa aL
  aR' = econa aR
  NEW = [true-lab ↦ TT, false-lab ↦ TF]

econe k (let x := (if (cmp aL aR) eT eF) in eb) =
  (L⇒TT ++ L⇒TF ++ L⇒TB ++ NEW,
    goto-if (cmp aL' aR') true lab false lab)
  NEW = [true lab ⇒ TT, false lab ⇒ TF, suffix lab ⇒ TB]
  (L⇒TT, TT) = econe NK eT
  (L⇒TF, TF) = econe NK eF
  (L⇒TB, TB) = econe k eB
  NK = (λ (fa) (seq (set! x fa)
    (goto suffix lab)))