\---

111111111110100 q9(x := 2) || (x := 3 ; x := x + x)

* + 1. x = 2, 4, 5, 6
       1. For:asw
* x = 2: 1 evaluation path. With x := 2 executed last.
* x = 4: 2 evaluation paths. With x := 2 executed:
  + just before (skip; x := x + x)
  + just before (x := x + x)
* x = 5: 1 evaluation path. With x := 2 executed:
  + just before x = 3 + x
* x = 6 : 3 evaluation paths. With x := 2 executed:
  + just before x = 3 + 3
  + just before x = 6
  + First
    1. x = 2, 4, 6
    2. For:
* x = 2: 1 evaluation path. With x:=2 executed last.
* x = 4: 2 evaluation paths. With x:=2 executed
  + just before (skip: x := x + x)
    - just before (x := x + x)
* x = 6: 1 evaluation path. With x:=2 exec uted first
  1. 1. **Base Case (k = 0)**

For k to equal 0, E must be of the form n

<x := n, s> ->c <skip, s[x -> n]> is true by definition.

1 = 0 + 1.

Therefore true.

for :Assuming , we have a chain of steps such that:

By the IH,

Using the rule: ASS-EXPR<E, s> ->e <E’, s><x:=E, s> ->c <x:=E’, s>  
We have:   
Which by definition of multistep evaluation we can say that Therefore true for all k >= 0.

**3**

* + 1. **Base Case (C of the form skip)**

There are no derivation rules for reducing skip, so the implication vacuously holds because its left-hand-side is always false.

**SBase Case (C of the form x:=E)**

*Assume:*

Then by semantics, C’=skip, s’=s[x-> n], n=epsilon(E, s)

Using L1 and n=epsilon(E,s), leads to

By ci, this then leads to

**Inductive Case (C of the form C;C’)**

*Assume:*

**Case A: (C=skip, C’ = C’’, s = s’’)**

True directly

**Case B: ()**

Use L2 and IH on C.

**Inductive Case (C of the form C||C’)**

*Assume:*

**Case A: (skips all round)**

True directly

**Case B: ()**

Uses IH, then L3

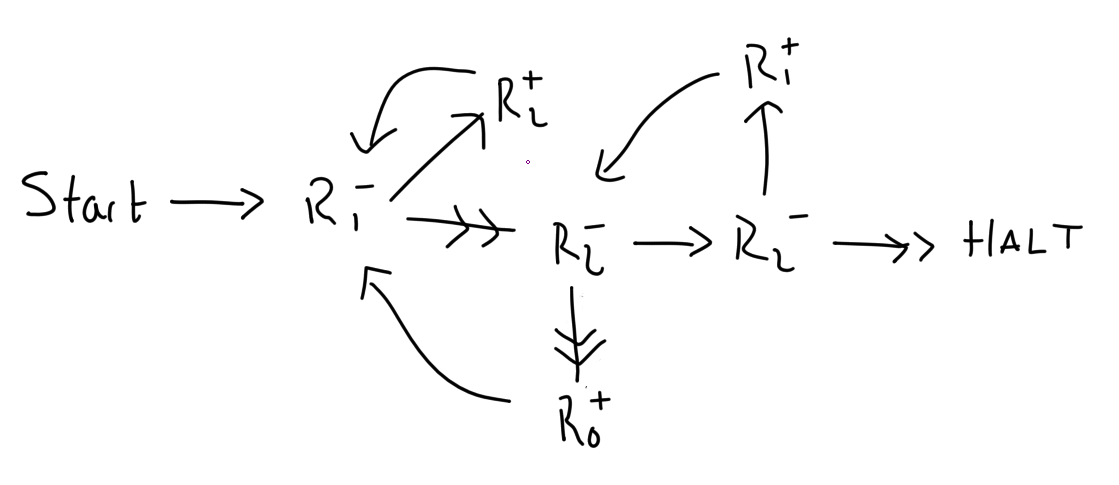
**Case C: ()**

**Same as above, but with L4**

2.

a)

i)



TL;DR if the list x :: L is passed into R1, after it finishes the head of the list (x) will be in R0 and the rest of the list (L) will be in R1

After moving the list (x :: L) into R2 it divides it by 2 by repeated subtraction putting the result into R1. So if R2 (i.e. the original list) was 2x(2L + 1), R1 is now 2x-1(2L + 1) and R2 is 0. R0 is then incremented and it starts again, with the result in R1 decreasing each time from 2x-1(2L + 1) to 2x-2(2L + 1) to 2x-3(2L + 1) and the result in R0 increasing at the same rate (1, 2, 3, …). Eventually this will res when the process starts again, ault in the result in R1 being just (2L + 1) and R0 being x. Sos R2 will be odd, after enough iterations R1 will equal L and R2 will equal 1, so on the 2nd R2- operation it will halt, leaving the head of the list in R0 and the rest of the list in R1.

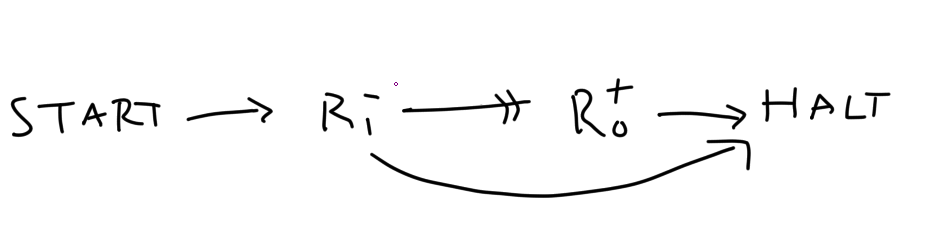
ii)

List is empty if R1 = 0 (I think)

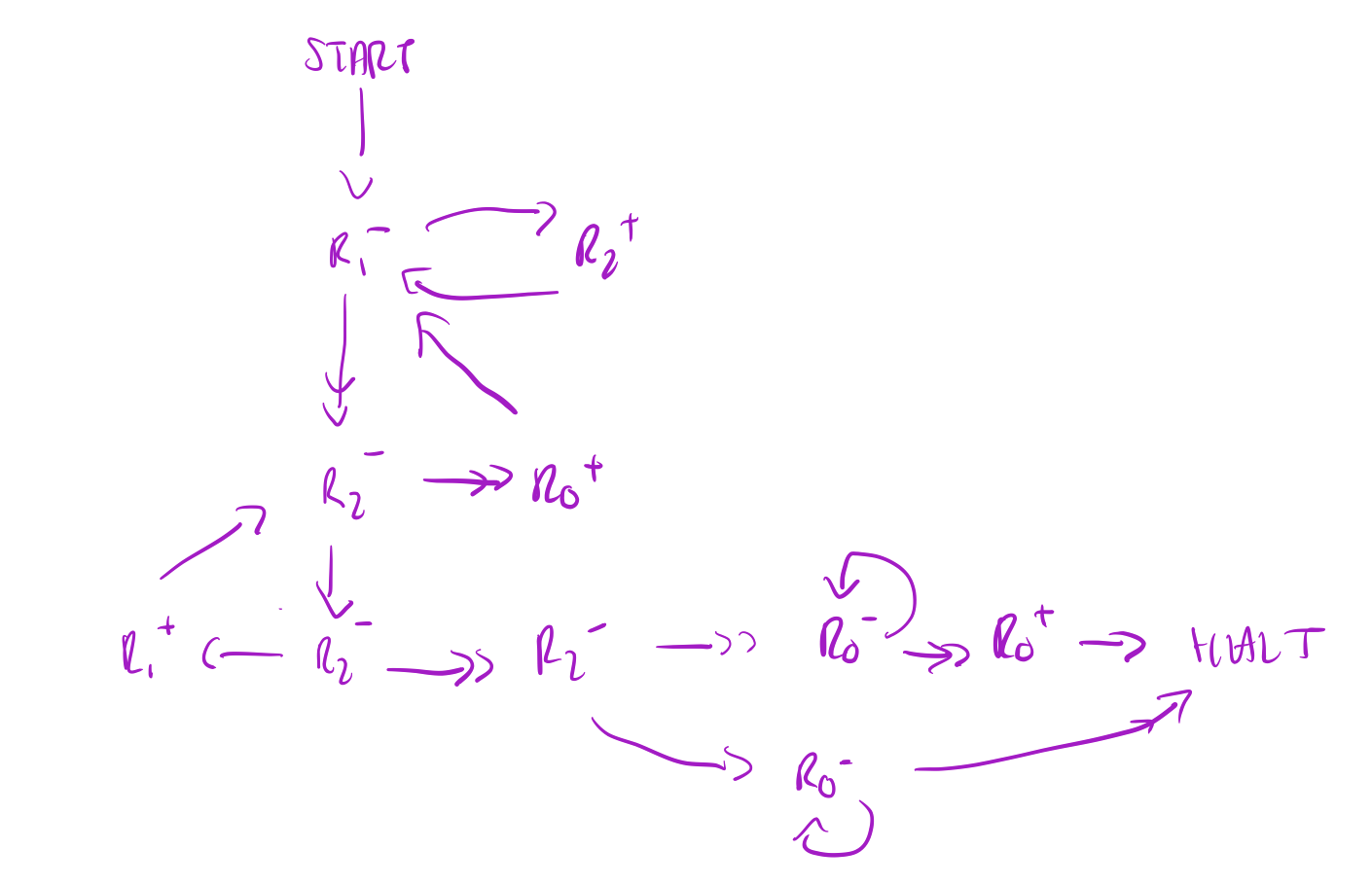
L0 : R-1 -> L2, L1

L1 : R0+ -> L2

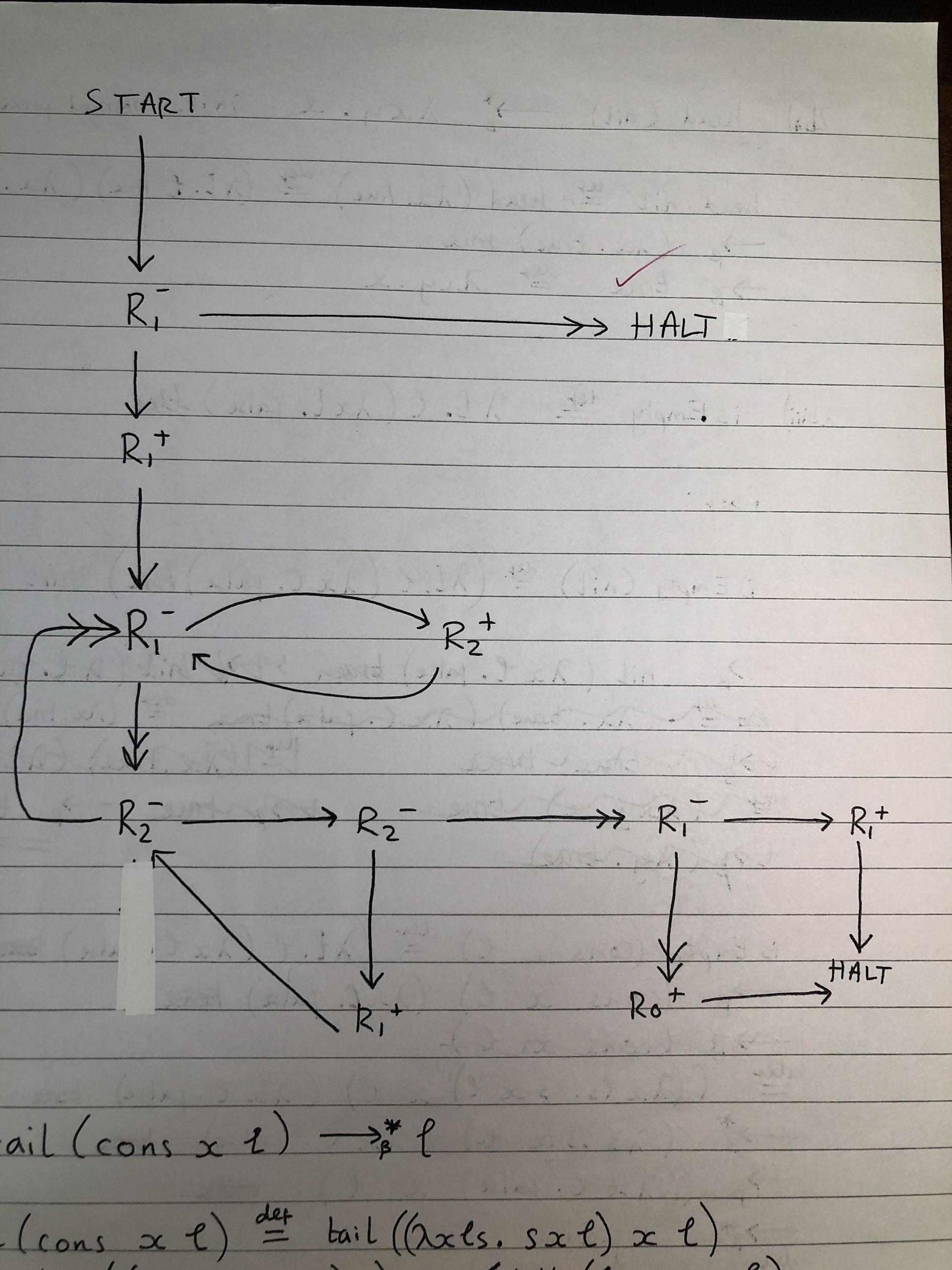
L2 : HALT



iii) Part i) finds the head of the list, and part ii) can tell if the list is empty. So instead of halting at the end of part i) we check if the remaining list L is empty using part ii)



Revised solution with suggested changes:



2b)

i)

tail(cons x l)

= λl. l false (cons x l)

-> (cons x l) false

= (λxls. sxl) x l false

->\* false x l

= (λxy. y) x l

->\* l

ii)

head (nil)

= λl. l true nil

-> nil true

= λx. true true

-> true

= λxy.x

No more redexes so this is the normal form.

Head normally applied can be thought of as passing the λ-term “true” to the list so it can be applied to the first and remaining elements. Nil however will discard any λ-term passed to it and return true instead.

iii) isEmpty = λl. l (λ xy. false)

If we have nil then the second part will be ignored and the nil will just use true. If we don’t have nil the second part will ignore its two arguments and just use false.

isEmpty (nil)

= λl.l (λxy.false) nil

-> nil (λxy.false)

= λx.true (λxy.false)

-> true

isEmpty (cons x l)

= λl.l (λxy.false) (cons x l)

-> (cons x l) (λxy.false)

= ((λxls.s x l) x l) (λxy.false)

-> (λs.s x l) (λxy.false)

-> (λxy.false) x l

-> false