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

a.

i. 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

b.

i. x = 2, 4, 6

ii. 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 executed first

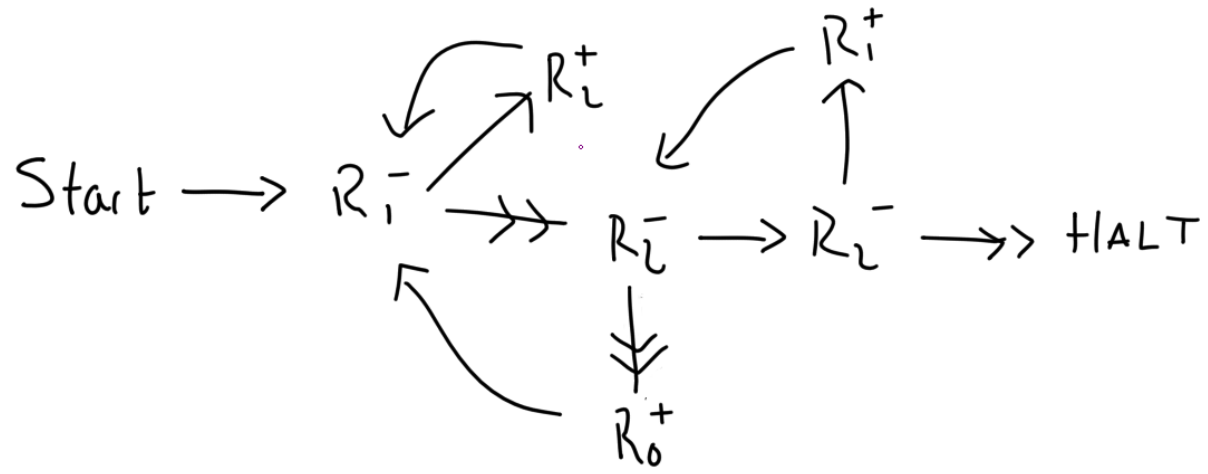
c.

i. **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 $< E', s >\rightarrow_e^k < n, s > => < x := E', s >\rightarrow_c^{k+1} < skip, s[\ x \rightarrow\ n\ ] >$ Assuming $< E, s >\rightarrow_e^{k+1} < n, s >$, we have a chain of steps such that:
$< E, s >\rightarrow_e < E', s >\rightarrow_e^k < n, s >$
By the IH, $< x := E', s >\rightarrow_c^{k+1} < skip, s[\ x \rightarrow\ n\ ] >$
Using the rule: ASS-EXPR<u><E, s> ->e <E', s></u><x:=E, s> ->c <x:=E', s>
We have: $< x := E, s >\rightarrow_c^k < x := E', s >\rightarrow_c^{k+1} < skip, s[\ x \rightarrow\ n\ ] >$
Which by definition of multistep evaluation we can say that $< x := E, s > \rightarrow_e^{k+2} < skip, s[x \rightarrow n] >$Therefore true for all k >= 0.

ii. **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.
Base Case (C of the form x:=E)

Assume: $< x := E, s > \sim > < C', s' >$
Then by semantics, C'=skip, s'=s[x-> n], n=epsilon(E, s)
Using L1 and n=epsilon(E,s), leads to $< E. s > \ ->_e{}^* < n, s >$
By ci, this then leads to $< x := E. s > \ ->_c{}^* < skip, s[x -> n] >$
Inductive Case (C of the form C;C')
Assume: $< C;\ C', s > \sim > < C'', s' >$
Case A: (C=skip, C' = C'', s = s'')
True directly
Case B: ($C'' \ = \ C_1;\ C', < C, s > \sim > < C', s' >$**)**
Use L2 and IH on C.
Inductive Case (C of the form C||C')
Assume: $< C_1 || \ C_2, s > \sim > < C', s' >$
Case A: (skips all round)
True directly
Case B: ($C' \ = \ C_1' || \ C_2, < C_1, s > \sim > < C_1', s' >$**)**
Uses IH, then L3
Case C: ($C' \ = \ C_1 || \ C_2', < C_2, s > \sim > < C_2', s' >$**)**
Same as above, but with L4

2.
a)
i)



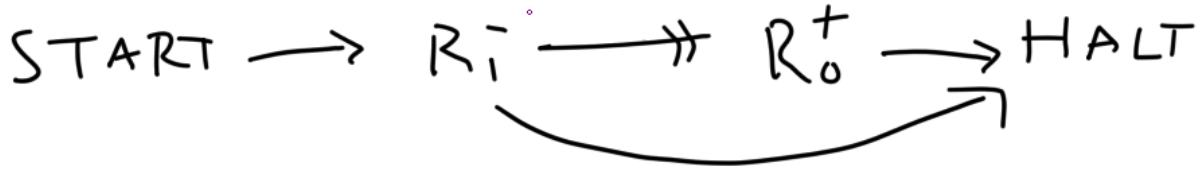
TL;DR if the list x :: L is passed into R_1, after it finishes the head of the list (x) will be in R_0 and the rest of the list (L) will be in R_1

After moving the list (x :: L) into R_2 it divides it by 2 by repeated subtraction putting the result into R_1. So if R_2 (i.e. the original list) was $2^x(2L + 1)$, R_1 is now $2^{x-1}(2L + 1)$ and R_2 is 0. R_0 is then incremented and it starts again, with the result in R_1 decreasing each time from $2^{x-1}(2L + 1)$ to 2^{x-}

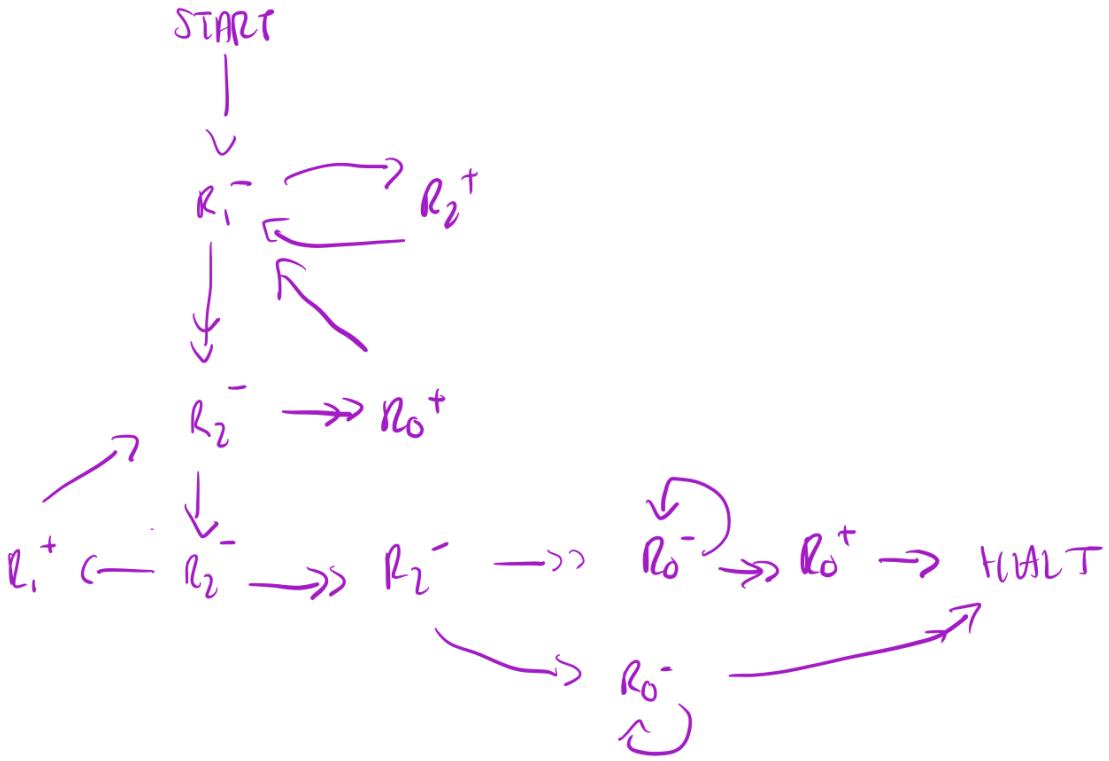
$^{2}(2L + 1)$ to $2^{x-3}(2L + 1)$ and the result in R_0 increasing at the same rate (1, 2, 3, …). Eventually this will res when the process starts again, ault in the result in R_1 being just $(2L + 1)$ and R_0 being x. Sos R_2 will be odd, after enough iterations R_1 will equal L and R_2 will equal 1, so on the 2nd R_2^- operation it will halt, leaving the head of the list in R_0 and the rest of the list in R_1.

ii)
List is empty if R1 = 0 (I think)

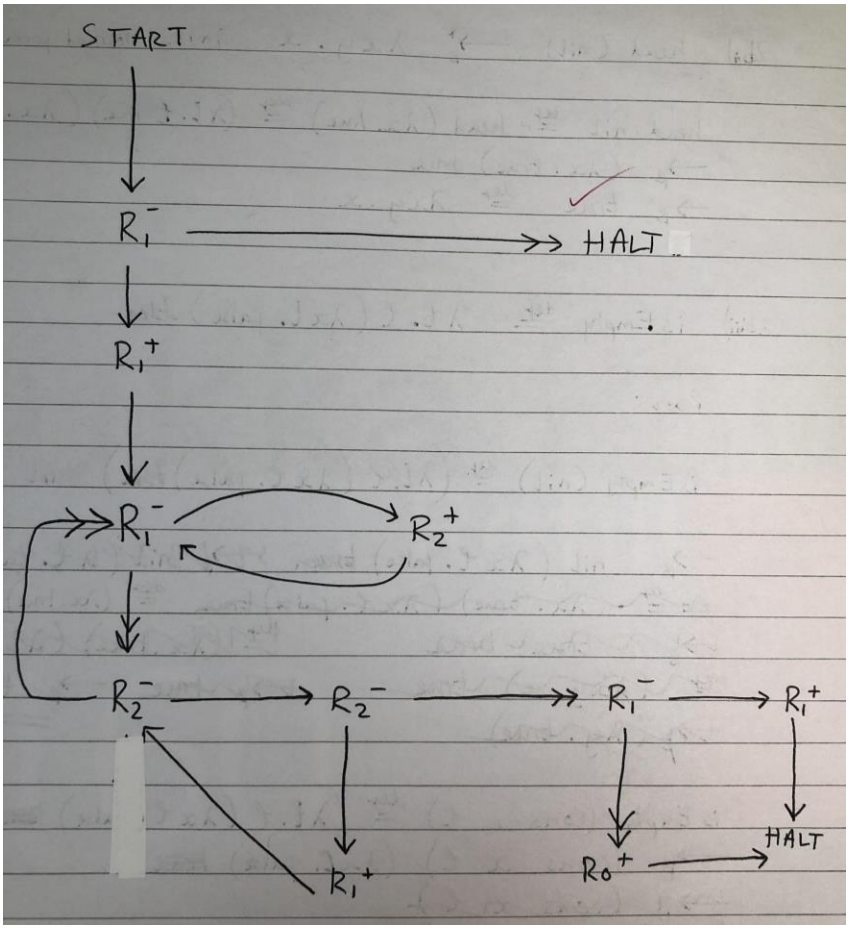
$L_0 : R^-_1 \rightarrow L_2, L_1$
$L_1 : R_0^+ \rightarrow L_2$
L_2 : HALT



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)
->β	(cons x l) false
=	(λxls. sxl) x l false
->*β	false x l
=	(λxy. y) x l
->*β	l

ii)

	head (nil)
=	λl. l true nil
-> β	nil true
=	λx. true true
->β	true
=	λxy.x

No more βredexes 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
->β nil (λxy.false)
= λx.true (λxy.false)
->β true

isEmpty (cons x l)
= λl.l (λxy.false) (cons x l)
->β (cons x l) (λxy.false)
= ((λxls.s x l) x l) (λxy.false)
->β (λs.s x l) (λxy.false)
->β (λxy.false) x l
->β false