xProblem 1

a)
i)
<E,s>-><E’,s>

<x:=E,s> -> <x:=E’,s>

<x:=n,s> -> <skip,s[x->n]>

<C_1,s>-><C_1’,s’>

<C_1;C_2,s>-><C_1’;C_2,s’>

<skip;C, s>-><C, s>

ii)
<(x:=1) || (x:=2 ; x:= (x+2)), (x->0)> ->
<(x:=1) || (skip ; x:= (x+2)), (x->2)> ->
<(x:=1) || (x:= (x+2)), (x->2)> ->
<(x:=1) || (x:= (2+2)), (x->2)> ->
<(x:=1) || (x:= 4), (x->2)> ->
<(x:=1) || skip, (x->4)> ->
<(x:=1), (x->4)> ->
<skip, (x->1)>

<(x:=1) || (x:=2 ; x:= (x+2)), (x->0)> ->
<(x:=1) || (skip ; x:= (x+2)), (x->2)> ->
<(x:=1) || (x:= (x+2)), (x->2)> ->
<skip || (x:= (x+2)), (x->1)> ->
< (x:= (x+2)), (x->1)> ->
< (x:= (1+2)), (x->1)> ->
< (x:= 3), (x->1)> ->
< skip, (x->3)>

<(x:=1) || (x:=2 ; x:= (x+2)), (x->0)> ->
<skip || (x:=2 ; x:= (x+2)), (x->1)> ->
< (x:=2 ; x:= (x+2)), (x->1)> ->
<skip ; (x:= (x+2)), (x->2)> ->
< (x:= (x+2)), (x->2)> ->
< (x:= (2+2)), (x->2)> ->
< (x:= 4), (x->2)> ->
< skip, (x->4)>

iii)
<(x:=1) || (x:=2 ; x:= (x+2)), (x->0)> ->
<skip || (x:=2 ; x:= (x+2)), (x->1)> ->
< (x:=2 ; x:= (x+2)), (x->1)> ->
<skip ; (x:= (x+2)), (x->2)> ->
< (x:= (x+2)), (x->2)> ->
< (x:= (2+2)), (x->2)> ->
< (x:= 4), (x->2)> ->
< skip, (x->4)>

<(x:=1) || (skip ; x:= (x+2)), (x->2)> ->
<(x:=1) || x:= (x+2), (x->2)> ->
< x:=1 || x:= (2+2), (x->2)> ->
<skip || (x:= (2+2)), (x->1)> ->
<(x:= 2+2), (x->1)> ->
<(x:= 4), (x->1)> ->
<skip, (x->4)> ->

<(x:=1) || (skip ; x:= (x+2)), (x->2)> ->
<(x:=1) || x:= (x+2), (x->2)> ->
<x:=1 || x:= (2+2), (x->2)> ->
<x:=1 || (x:= 4), (x->1)> ->
<skip || (x:= 4), (x->1)> ->
<(x:= 4), (x->1)> ->
<skip, (x->4)> ->

b)
i) I think because this is because the translation [-] converts from the parallel language to the normal language, and is identical other than the fact that it translates into one particular interpretation of the parallel command (evaluating the LHS strictly before the RHS by replacing the parallel composition with a regular composition).

Since this one particular interpretation is still a valid way to interpret the parallel composition, there still exists a derivation in the parallel language to the translated version.

ii) The translation [-] translates from the parallel language to one interpretation of the parallel language expressed in the regular language. The translation forces the left hand side of the parallel composition to be evaluated first (which is still one valid way to interpret the parallel composition) - i.e: it converts the parallel command into a simple composition.

Since we know that the regular language is normalising, we know that this interpretation of the parallel language is normalising, since it is expressed using only constructs of the regular language. Thus since the parallel command has a way to be evaluated that is normalising, and that is the only difference to the normalising regular language, the entire parallel language is normalising.

iii) :(

We need to show

P(C) = ∀C', s, s': <[C], s> -> <C', s'> => ∃C'' : <C, s> -> <C'', s'> ∧ [C''] = C'

<Insert base cases here>

Inductive Case: C = $C_1;C_2$
We want to show

P($C_1;C_2$) = ∀C', s, s': <[$C_1;C_2$], s> -> <C', s'> => ∃C'' : <$C_1;C_2$, s> -> <C'', s'> ∧ [C''] = C'

By the definition of [-] it is sufficient to show

<[C_1];[C_2], s> -> <C', s'> => ∃C'' : <$C_1;C_2$, s> -> <C'', s'> ∧ [C''] = C'

Assuming P(C_1) and P(C_2) as our inductive hypotheses. Namely

P(C_1) = ∀C_1', s, s': <[C_1], s> -> <C_1', s'> => ∃C_1'' : <C_1, s> -> <C_1'', s'> ∧ [C_1''] = C_1'

P(C_2) = ∀C_2', s, s': <[C_2], s> -> <C_2', s'> => ∃C_2'' : <C_2, s> -> <C_2'', s'> ∧ [C_2''] = C_2'

Assume C', s, s' are such that <[C_1];[C_2], s> -> <C', s'>. There are two rules that could give such a derivation:

(W-SEQ.LEFT) $$\frac{\langle C_1, s\rangle \to_c \langle C_1', s'\rangle}{\langle C_1; C_2, s\rangle \to_c \langle C_1'; C_2, s'\rangle}$$

(W-SEQ.SKIP) $$\frac{}{\langle \texttt{skip}; C_2, s\rangle \to_c \langle C_2, s\rangle}$$

W-SEQ.LEFT:
In this case C' = C_1';[C_2] for some C_1' with [C_1] -> C_1'. By the inductive hypothesis P(C_1) we have <C_1, s> -> <C_1'', s'> ∧ [C_1''] = C_1'. So
<$C_1;C_2$, s> -> <C_1'';C_2, s'> by W-SEQ.LEFT, and [C_1'';C_2] = [C_1''];[C_2] = C_1';[C_2].

W-SEQ.SKIP:
In this case C_1 = skip and C_2 = C' . Then we have <skip;[C_2], s> -> <[C_2], s'>. So <skip;C_2, s> -> <C_2, s'> and [C_2] = [C_2].

The remaining case is left as an exercise to the reader.

Problem 2

a)
i)

A register machine is specified by:

1. finitely many register R_0, R_1,R_2,...,R_n, each capable of storing a natural number
2. a program that consists of finitely many instructions of the from: label:body where the body takes the form: **R^+ = L'**, **R^- = L', L''** and **HALT**.

ii)

If there is a register machine with at least n+1 registers R_0, R_1, R_2, … R_n (and maybe more) such that for all (x_1,x_2,x_3, ..., x_n) belongs to **R^n**, the register machine will terminate and have R_0 = y iff f(x_1,x_2,x_3, …,x_n) = y.

b)

i)

ii)

Gets the head of the list : x such that $2^x(2^y+1)$. Same machine as in the 2018 paper.

Given the input l > 0, where l = (2^x)(2^y + 1) for some x and y, the program will end with R0 = x, R1 = y, with R2 being just a scratch register. This means that R0 will hold the encoding of the head of the list, whilst R1 will hold the encoding of the tail of the list.

iii)

With R1 = R2 = 0, we get into an infinite loop (R1- =>> R2- =>> R0+ => R1- =>>...) so the register machine doesn't come to a halt. This is reasonable because getting the head of an empty list should be undefined behaviour.

c)

An Universal register machine is a machine that can simulate any register machine in any given input.

This is used in URMs to pop program arguments and retrieving the next instruction of the program.