A Universal Register Machine.

Part I :

Coding register machines as numbers

A key part of the Turing / Church solution of Hilbert's Entscheidungs-problem was to exploit the idea that

(formal descriptions of) algorithms can be the data on which algorithms act.

We are using register machines as the formal description of the informal notion of "algorithm". Since the data that register machines manipulate are numbers, to develop the above idea we have to [have an algorithm to] code register machines as numbers.*

To do that we need to be able to code $\begin{cases} \text{pairs of numbers} \\ \text{finite lists of numbers} \end{cases}$

as numbers. There are many ways of doing that: we fix upon one convenient way ...

---

* such codings are often called Gödel numberings, after Gödel's original use of the idea: his coding of arithmetic formulas as numbers was a key part of his proof of the famous Incompleteness Theorem.

Coding pairs of numbers as numbers

For $x, y \in \mathbb{N}$ define

$$\langle\!\langle x, y \rangle\!\rangle \stackrel{\text{def}}{=} 2^x \cdot (2y+1)$$

$$\langle x, y \rangle \stackrel{\text{def}}{=} 2^x \cdot (2y+1) - 1$$

Thus

$$\boxed{\langle\!\langle x, y \rangle\!\rangle \text{ in binary}} = \boxed{y \text{ in binary} \mid 1 \mid \overbrace{0 \cdots 0}^{x\ 0\text{'s}}}$$

$$\boxed{\langle x, y \rangle \text{ in binary}} = \boxed{y \text{ in binary} \mid 0 \mid \underbrace{1 \cdots 1}_{x\ 1\text{'s}}}$$

Hence

- $\langle\!\langle -, - \rangle\!\rangle$ and $\langle -, - \rangle$ both determine <u>injective</u> functions from $\mathbb{N} \times \mathbb{N}$ to $\mathbb{N}$, i.e.

  $\langle\!\langle x_1, y_1 \rangle\!\rangle = \langle\!\langle x_2, y_2 \rangle\!\rangle \Rightarrow x_1 = x_2 \ \& \ y_1 = y_2$

  $\langle x_1, y_1 \rangle = \langle x_2, y_2 \rangle \Rightarrow x_1 = x_2 \ \& \ y_1 = y_2$

- $\langle\!\langle -, - \rangle\!\rangle$ is a <u>surjective</u> function from $\mathbb{N} \times \mathbb{N}$ to $\{z \in \mathbb{N} \mid z \neq 0\}$, i.e. for all $z \neq 0$ there are $x, y \in \mathbb{N}$ with $\langle\!\langle x, y \rangle\!\rangle = z$.

- $\langle -, - \rangle$ is a surjective function from $\mathbb{N} \times \mathbb{N}$ to $\mathbb{N}$ i.e. for all z there are $x, y \in \mathbb{N}$ with $\langle x, y \rangle = z$.

and hence $\langle -, - \rangle$ is a <u>bijection</u> (a.k.a. one-to-one correspondence) between $\mathbb{N} \times \mathbb{N}$ and $\mathbb{N}$.

(and $\langle\!\langle -, - \rangle\!\rangle$ a bijection between $\mathbb{N} \times \mathbb{N}$ and $\{z \mid z \neq 0\}$).

NOTATION for lists (of numbers)

$\mathbb{N}^* \stackrel{\text{def}}{=}$ set of finite lists of natural numbers

$$= \underbrace{\{nil\}}_{\substack{\text{unique list}\\ \text{of length } 0}} \cup \underbrace{\mathbb{N}}_{\substack{\text{lists}\\ \text{of length } 1}} \cup \mathbb{N}^2 \cup \cdots \cup \underbrace{\mathbb{N}^n}_{\substack{\text{lists } (x_1,\ldots,x_n)\\ \text{of length } n}} \cup \cdots$$

$cons \in Fun(\mathbb{N} \times \mathbb{N}^*, \mathbb{N}^*)$

$head \in Pfn(\mathbb{N}^*, \mathbb{N})$

$tail \in Fun(\mathbb{N}^*, \mathbb{N}^*)$

$cons$ is a bijection from $\mathbb{N} \times \mathbb{N}^*$ to $\{\ell \in \mathbb{N}^* \mid \ell \neq nil\}$	
$head(cons(x,\ell)) = x$	$head(nil)\uparrow$
$tail(cons(x,\ell)) = \ell$	$tail(nil) = nil$
$\ell = cons(head(\ell), tail(\ell))$ if $\ell \neq nil$	

Every list can be built up from nil by repeated $cons$'s:

$$(x_1,\ldots,x_n) = cons(x_1, cons(x_2, \ldots, cons(x_n, nil)\cdots))$$

Coding lists $(x_1,\ldots,x_n) \in \mathbb{N}^*$ as numbers $[x_1,\ldots,x_n] \in \mathbb{N}$

Define $[x_1,\ldots,x_n] \in \mathbb{N}$ by induction on the length of the list $(x_1,\ldots,x_n) \in \mathbb{N}^*$:

CASE $n=0$: $[nil] \stackrel{\text{def}}{=} 0$

INDUCTION STEP: $[cons(x,\ell)] \stackrel{\text{def}}{=} \langle\langle x, [\ell]\rangle\rangle = 2^x(2[\ell]+1)$

Thus in general $[x_1,\ldots,x_n] = \langle\langle x_1, \langle\langle x_2, \ldots \langle\langle x_n, 0\rangle\rangle \cdots\rangle\rangle\rangle\rangle$

From the definition of $\langle\langle -,-\rangle\rangle$ we get:

$$\boxed{[x_1,\ldots,x_n] \text{ in binary}} = \boxed{1 \; \underbrace{0\cdots0}_{x_n \text{ 0's}} \; 1 \; \underbrace{0\cdots0}_{x_{n-1} \text{ 0's}} \; \cdots \; 1 \; \underbrace{0\cdots0}_{x_1 \text{ 0's}}}$$

number of 1's in $=$ length of list $(x_1,\ldots,x_n)$

Hence $\ell \mapsto [\ell]$ determines a bijection from $\mathbb{N}^*$ to $\mathbb{N}$

Examples

$$[3] = [\mathrm{cons}(3, \mathrm{nil})] = \langle\!\langle 3, 0 \rangle\!\rangle = 2^3(2\cdot 0+1) = 8 \quad \text{decimal}$$

$$= 1\underbrace{000}_{3} \quad \text{binary}$$

$$[1,3] = \langle\!\langle 1, [3] \rangle\!\rangle = \langle\!\langle 1, 8 \rangle\!\rangle = 2^1(2\cdot 8+1) = 34 \quad \text{decimal}$$

$$= 1\underbrace{000}_{3}1\underbrace{0}_{1} \quad \text{binary}$$

$$[2,1,3] = \langle\!\langle 2, [1,3] \rangle\!\rangle = \langle\!\langle 2, 34 \rangle\!\rangle = 2^2(2\cdot 34+1) = 276 \quad \text{decimal}$$

$$= 1\underbrace{000}_{3}1\underbrace{0}_{1}1\underbrace{00}_{2} \quad \text{binary}$$

(<u>NB</u> reversal of order of list when reading left-to-right in binary representation.)

In order to code register machine programs as numbers, without loss of generality (i.e. without affecting which partial functions can be computed) we can assume:

- the registers of a machine with $n+1$ registers are always called $R0, \ldots, Rn$
- the labels occurring in a register machine program are called $L0, L1, L2, \ldots$, and the $(i+1)^{th}$ instruction in the program listing is labelled Li.

Coding register machine programs Prog numbers $\ulcorner Prog \urcorner \in \mathbb{N}$

If Prog is

$$\boxed{\begin{array}{l} L0: body_0 \\ L1: body_1 \\ \quad\vdots \\ Lm: body_m \end{array}}$$

then $\ulcorner Prog \urcorner \stackrel{def}{=} [\, code(body_0), \dots, code(body_m) \,]$

where
$$\begin{cases} code(Ri^+ \to Lj) \stackrel{def}{=} \langle\!\langle 2i, j \rangle\!\rangle \\ code(Ri^- \to Lj, Lk) \stackrel{def}{=} \langle\!\langle 2i+1, \langle j, k \rangle \rangle\!\rangle \\ code(HALT) \stackrel{def}{=} 0 \end{cases}$$

Any $x \in \mathbb{N}$ decodes uniquely as an instruction:

if $x = 0$ then the instruction is HALT
else decode x as a pair $x = \langle\!\langle y, z \rangle\!\rangle$ and
 if y is even then instruction is
  $Ri^+ \to Lj$ where $i = y/2$ & $j = z$
 else (y is odd and) decode z as a pair $z = \langle u, v \rangle$
 and then the instruction is
  $Ri^- \to Lj, Lk$ where $i = (y-1)/2$, $j = u$ & $k = v$.

Hence any $e \in \mathbb{N}$ decodes uniquely as a program $Prog_e$, called the (register machine) program with index e:

first decode e as a list $e = [x_1, \dots, x_n]$
and then decode each x_i as an instruction,
as above.

NB (1) The program resulting from this decoding process may well have jumps to labels greater than the length of the list of instructions, i.e. the associated register machine may well be capable of halting erroneously – but no matter.

(2) In case $e = 0 = [nil]$ we get an empty list of instructions, which by convention we regard as a machine that does nothing.

Example : decode 666 as a program (for the register machine from hell !)

decimal 666 = binary 1010011010

$= [1, 1, 0, 2, 1]$

Now
0 is code for instruction HALT
$1 = \langle\langle 0, 0 \rangle\rangle$ is code for instruction $R0^+ \to L0$
$2 = \langle\langle 1, 0 \rangle\rangle = \langle\langle 1, \langle 0, 0 \rangle \rangle\rangle$ is code for $R0^- \to L0, L0$

So 666 decodes to the program

L0 : $R0^+ \to L0$
L1 : $R0^+ \to L0$
L2 : HALT
L3 : $R0^- \to L0, L0$
L4 : $R0^+ \to L0$

(which never halts).

A Universal Register Machine.

Part II:

Description of the machine.

High-level description of a universal register machine, U:

- U has registers P(rogram), A(rgument), ...
- Loading P with value e, A with value a and all other registers with 0, then U acts as follows:
  - — decode e as a program: $e = \ulcorner Prog_e \urcorner$
  - — decode a as a list of register values: $a = [a_1, \dots, a_n]$
  - — carry out the computation of the register machine program $Prog_e$ starting with registers $R0, R1, \dots, Rn$ set to $0, a_1, a_2, \dots, a_n$ (and any other registers occurring in $Prog_e$ set to 0).

Overall structure of U's program:

1	copy P to T, copy PC^{th} item of list in T to N (HALTing if $PC >$ length of list) and goto **2**
2	if $N = 0$ ($=$ code(HALT)) then HALT, else decode N as $\langle\!\langle y, z \rangle\!\rangle$, assign y to C & z to N, and goto **3** {At this point either $C = 2i$ is even & current instruction is $Ri^+ \to Lj$ where $j = N$ or $C = 2i+1$ " odd " " " " $Ri^- \to Lj, Lk$ " $\langle j, k \rangle = N$ }
3	remove register values from list in A up to the one required* {the i^{th}}, putting it in R & saving preceding values as a list in S, then goto **4**
4	execute current instruction on value in R, update PC {to j or k as above}, restore register values from R and S to A, and goto **1**

* see Note on p 42

The registers of U and the rôle they play in its program:

P – holds the code of the register machine to be simulated

A – holds current contents of registers of the register machine being simulated

PC – program counter – holds the number of the current instruction (counting from 0)

N – holds the code of the current instruction

C – indicates the type of the current instruction

R – holds the contents of simulated machine's register that is to be incremented/decremented by current instruction (if not a HALT instruction)

T – holds a working copy of the program code

S, Z – auxiliary registers for intermediate computations

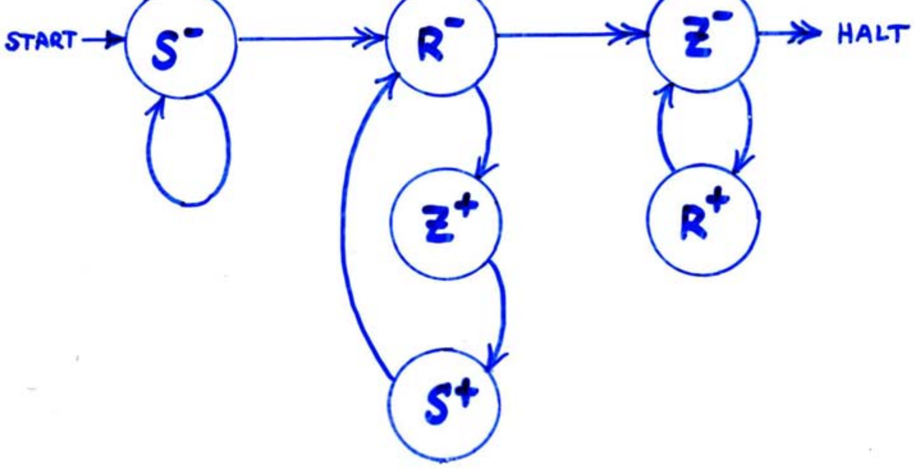
<u>Note</u>

At step 3 it may be that $i >$ length of the list in A, i.e. that current instruction wants to increment/decrement a register in the simulated machine that was not assigned a value by the initial value of A or has not been mentioned so far in the interpreted program. By assumption, such a register has value 0. So in this case, say $A = [a_1, \ldots, a_n]$ with $i > n$, at step 3 A will be set to $0 (= [nil])$, R set to 0, and S set to $[\underbrace{0, \ldots, 0}_{i-n-1 \text{ 0's}}, a_n, \ldots, a_1]$. Then when the register values are restored at step **4**, A will hold $[a_1, \ldots, a_n, \underbrace{0, \ldots, 0}_{i-n-1 \text{ 0's}}, r]$ where r is the value in R after executing the current instruction.

The detailed construction of U's program depends on the fact that various **procedures** for manipulating (codes of) lists of numbers are register machine programmable ...

The program START → $\boxed{\text{copy R to S}}$ → HALT

to carry out $S := R$ can be implemented by



Precondition :

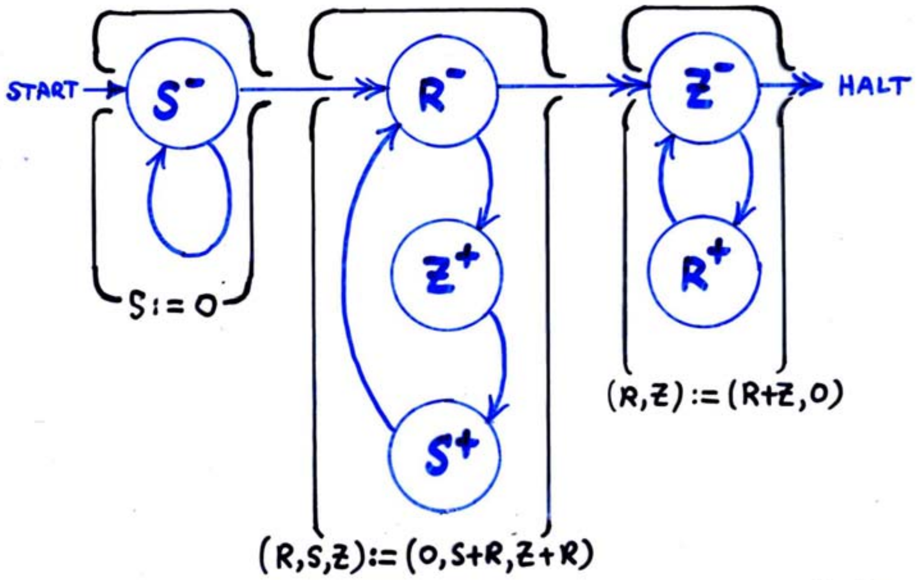
$R = x$
$S = y$
$Z = 0$

Postcondition :

$R = x$
$S = x$
$Z = 0$

The program START → $\boxed{\text{copy R to S}}$ → HALT

to carry out $S := R$ can be implemented by



Precondition :

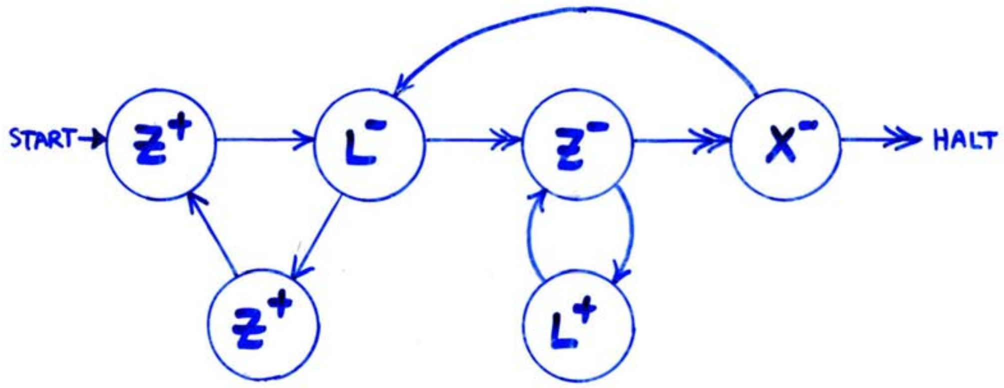
$R = x$
$S = y$
$Z = 0$

Postcondition :

$R = x$
$S = x$
$Z = 0$

The program START → **push X to L** → HALT

to carry out $(X,L) := (0, \mathrm{cons}(X,L))$ can be implemented by



Precondition:

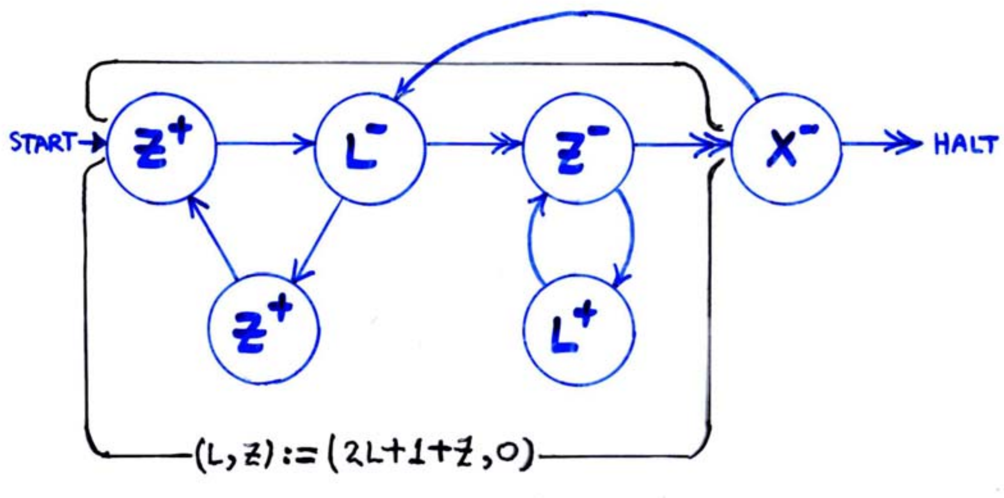
$X = x$
$L = \ell$
$Z = 0$

Postcondition:

$X = 0$
$L = \langle x, \ell \rangle = 2^x(2\ell+1)$
$Z = 0$

The program START → **push X to L** → HALT

to carry out $(X,L) := (0, \mathrm{cons}(X,L))$ can be implemented by



Precondition:

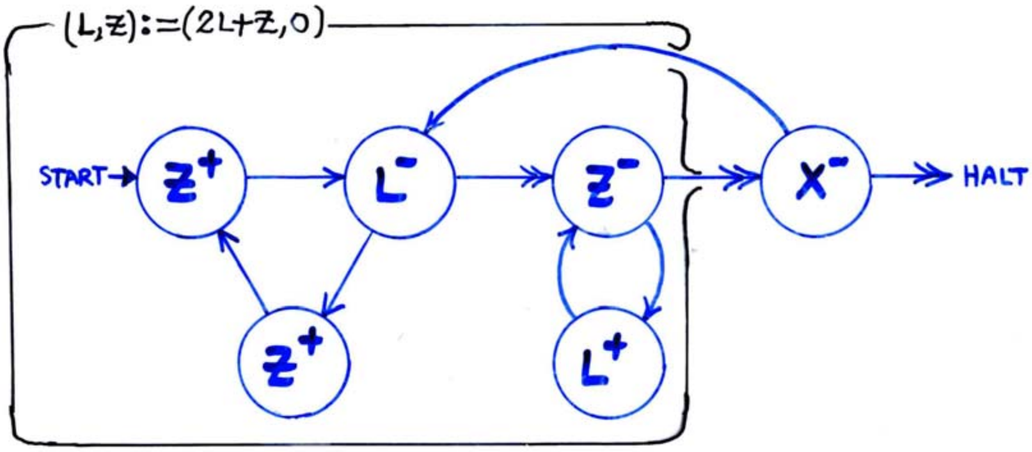
$X = x$
$L = \ell$
$Z = 0$

Postcondition:

$X = 0$
$L = \langle x, \ell \rangle = 2^x(2\ell+1)$
$Z = 0$

The program 

to carry out $(X,L) := (0, \mathrm{cons}(X,L))$ can be implemented by



Precondition:

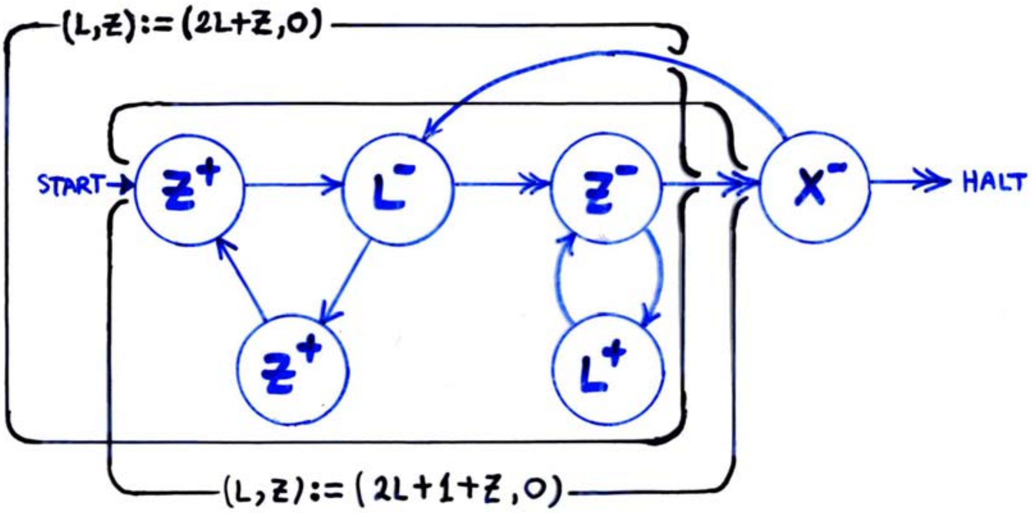
$X = x$
$L = \ell$
$Z = 0$

Postcondition:

$X = 0$
$L = \langle\!\langle x, \ell \rangle\!\rangle = 2^x(2\ell+1)$
$Z = 0$

The program 

to carry out $(X,L) := (0, \mathrm{cons}(X,L))$ can be implemented by

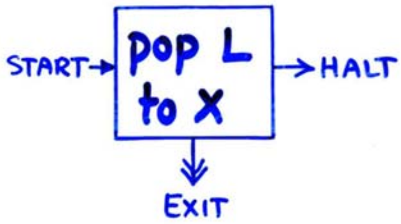


Precondition:

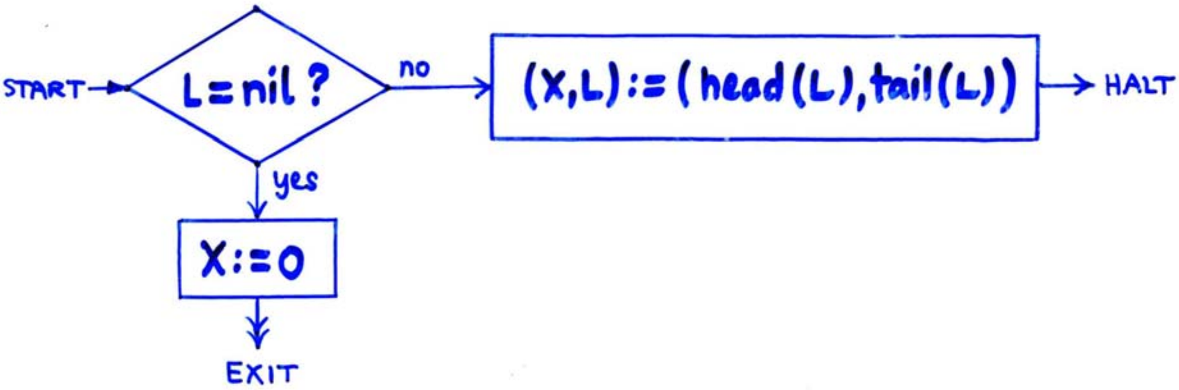
$X = x$
$L = \ell$
$Z = 0$

Postcondition:

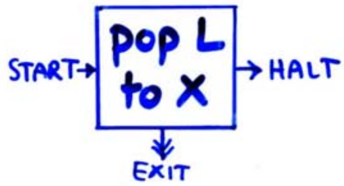
$X = 0$
$L = \langle\!\langle x, \ell \rangle\!\rangle = 2^x(2\ell+1)$
$Z = 0$

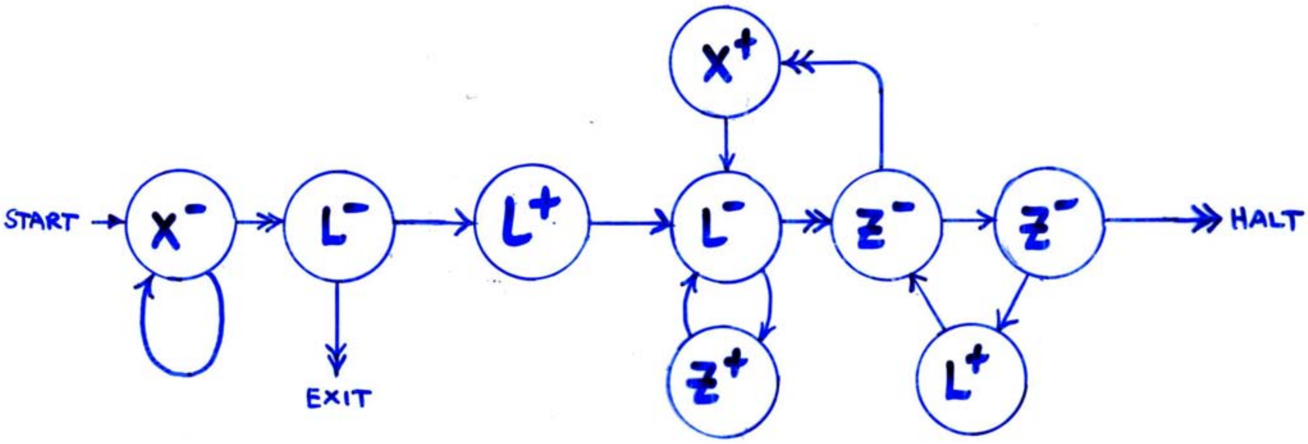
The program 

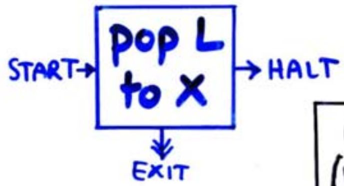
Specification:

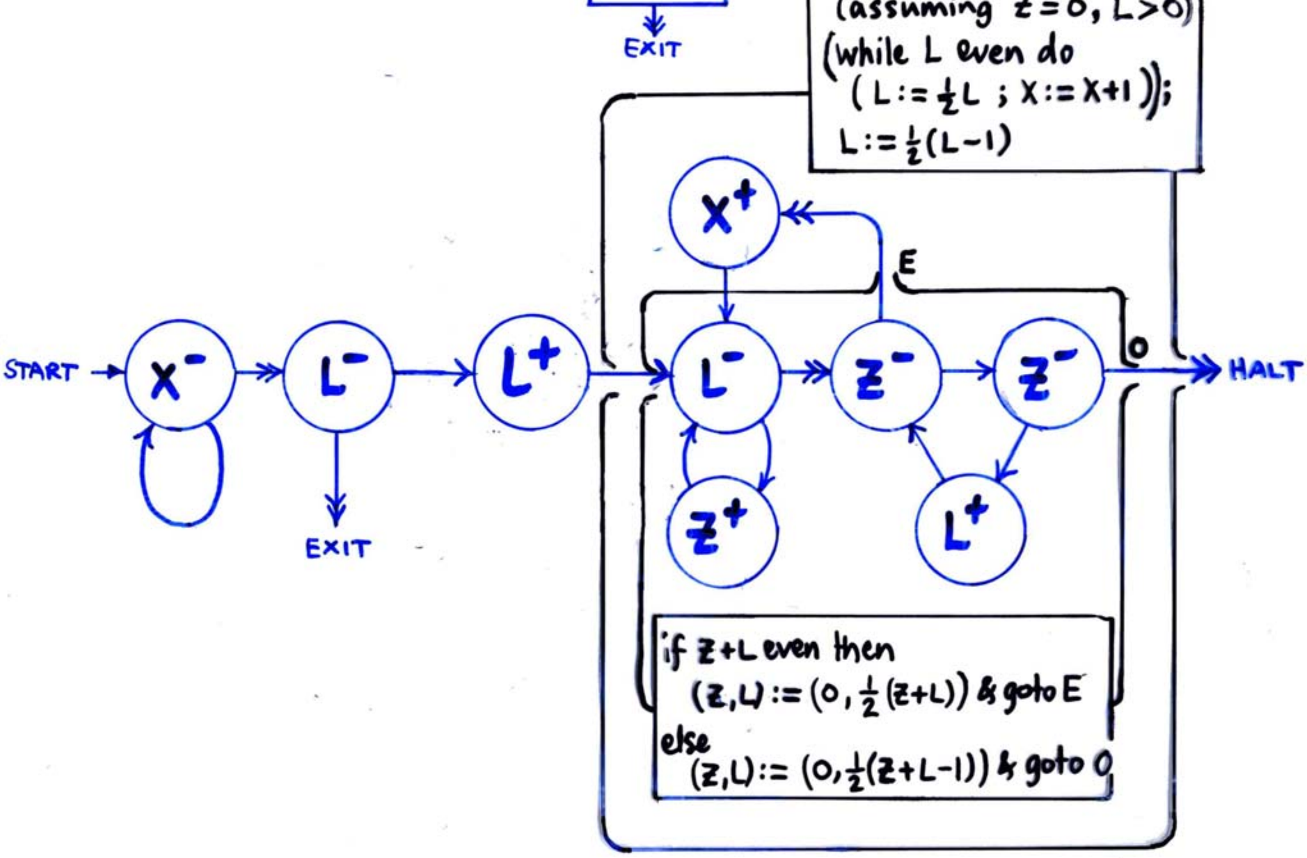


" if L = 0 then assign 0 to X and goto EXIT,
else L = $\langle x, \ell \rangle$ say, assign x to X and ℓ to L,
and goto HALT "

Implementation of  :



Implementation of  :



The program for $\mathcal{U}$: