Q1. Turing Machine for copying: (q_0 is the starting state; Done denotes that the machine has finished its copying).

	a	b	X	B
q_0	q_1, X, R	Done		
q_1	q_1, a, R	q_1, b, R		q_2, a, L
q_2	q_2, a, L	q_2, b, L	q_0, a, R	

Intuitively, the above Turing Machine in state q_0 would convert the a it is reading to X, and go to state q_1. In state q_1 it will skip the string to reach the first blank on the right and convert it to an a, and go to state q_2. In state q_2 it will go left to reach the X, convert it to an a, and move right to start the process for copying the next a.

Q2. Following is a two tape machine. 2nd tape is essentially used only for writing 1, and so symbol read there doesn't matter (X below stands for arbitrary symbol). Start state is q0.

In state q0: we move to the right end of input, and go to state q1.

In state q1: if input tape contains .....10^i, then this would be converted to ....01^i with 1 added to the second tape. Machine moves to state q0.

if input tape contains 0^i, then machine will halt (after moving through these 0's).

	(0,X)	(1,X)	(B,X)
q0	q0,(0,X), (R,S)	q0, (1,X), (R,S)	q1, (B,X), (L,S)
q1	q1,(1,X), (L,S)	q0, (0,1), (R,R)	Halt

Table 1: TM

Q3.

Intuitively, the following steps check if n, the number of a's in the input, is divisible by $n-1$, $n-2$, ..., 2 in that order. If it is, then n is not prime. Otherwise it is.

1. Initially input is in tape 1.
2. If input contains 0 or 1 a, then reject.
3. Copy input to tape 2.
4. Decrement the number of a's in tape 2 by 1.
5. If tape 2 contains only one a, then accept (input is a prime).
6. Move to left end of the strings in tape 1 and tape 2.
7. Move step by step, in both tape 1 and tape 2, to right, until one of them hits a blank.

8a. If both tapes hit blank at same time, then reject (it is not prime).

8b. If tape 2 hits blank first then, move to the left end of the string of tape 2, and go to step 7.

8c. If tape 1 hits blank first then, go to step 4.

Q4. f is partial recursive implies L_f is RE.

Suppose M is the machine which computes f. Then M' on input $x\#y$ does the following (if the input does not have exactly one #, then it rejects the input):

M' first copies x into second tape.

M' then runs M on input x (from second tape), with output on 3rd tape (with other tapes for temporary usage if needed, where first tape is not touched). If M never stops, then M' also never stops.

If and when M stops, then M' compares y on first tape with the output of M on the 3rd tape. If both are same, then M' accepts. Otherwise, it rejects.

L_f is RE implies f is partial recursive

Suppose M' is the machine which accepts L_f. Then M on input x uses the following algorithm to determine its answer.

For $t = 0$ to ∞

For all y of length at most t

If M' accepts $x\#y$ within t steps, then output y

EndFor

EndFor

Here, to check whether M' accepts $x\#y$ within t steps, one can initally place t in unary in the second tape, and place the head of second tape at the beginning of the number. Then, in each step of M', one moves the head of the second tape to the right. If M' accepts before reaching the blank at the end of t, then we know that it accepts within t steps. If M' reaches the blank before it accepts, then it does not accept within t steps.

Note that M needs to keep a copy of $x\#y$ before simulating M' so that modifications by M' doesn't effect the copy of M.

Q5. (sketch) To simulate a NPDA with two stack (accepting by final state) using non-deterministic TM, one can use one working tape per stack.

For any instantaneous description of the NPDA, for each stack, the contents of the stack is written on the tape (left end of the tape denoting the bottom of stack, and the head location representing the top of the stack). The remaining part of the input for NPDA is represented using the head on the input tape of TM (that is, for input tape TM will move right after consumption of each symbol by the NPDA).

Now each move of the NPDA can be easily simulated (as head on working tapes can let us know about top of stack values and the head on input tape can tell us about the input read by NPDA). Popping of a symbol can be simulated by moving left and pushing of symbols can be simulated by moving right and writing the appropriate symbols. Nondeterminism of PDA can be simulated by nondeterminism of TM.

Note though that we need to be careful of when TM accepts. It should accept only if the NPDA is in final state and the input is fully read.

For simulating one tape TM by 2-NPDA, ID $B^\infty \alpha q \beta B^\infty$ (where B is blank) of TM will be represented in the 2-NPDA by keeping $Z\alpha$ in stack 1 (with the letter of α closest to the head at the top of the stack and Z at the bottom of the stack) and βZ in stack 2 (with the letter of β where the head of TM is, at the top of the stack, and Z at the bottom of the stack). Z denotes infinitely many blanks. Initially, one can read the whole input and push it on stack 1 and then transfer it to stack 2 (that will make the 1st input letter on top of stack 2).

NOTE: Here the starting symbol on the stacks (or the bottom symbol of the stack), Z, is a special symbol, which represents infinitely many blanks.

Now each move of TM can be easily simulated as top of stack 2 tells us the symbol read by TM, state of TM can be remembered in the state of 2-NPDA. Writing on the tape can be done by pushing the symbol written on stack 2 (after poping the read symbol). Left move can be simulated by poping from stack 1 and pushing on stack 2. Right move can be simulated by poping from stack 2 and pushing on stack 1.

One needs to take special care if (while simulating a move as above) the symbol Z is encountered on the stack on popping: we need to push it back and "think" that blank has been popped — this is because the Z represents infinitely many blanks (and not just one blank).

Q6. (sketch) One can simulate the Z language by TM as follows.

Input tape contains X.

Each variable is kept on a different tape (in unary), with head of the tape being at the right end of the number. The left end of the tapes for each variable initially has \$ to represent 0.

Steps of the program are represented using states of the TM (where in some cases more than one state may be needed).

To implement instruction $V = V - 1$, one can erase the 1 on the corresponding tape for V and move the head left (if the corresponding tape for V has \$ on the cell being read, then no change takes place). To implement instruction $V = V + 1$, one moves right and then writes 1 on the corresponding tape (this will require two states to do the instruction). To implement instruction 'IF $V \neq 0$, then go to L' one could check if the head on tape corresponding to V is reading a \$, and if not, then next state would be the state corresponding to instruction L, otherwise the next instruction in the program.

To simulate a TM using language Z, one represents the ID $\alpha q \beta$ of TM using two numbers ℓ (for α) and r (for β), where right end of α is least significant bit, and left end of β is least significant bit. In representing these numbers, we use arithmetic with base $b = |\Gamma|$, where Γ is the tape alphabet for the TM. We assume that 'B' (blank) is given number 0, and the other characters in Γ are given numbers from 1 to $b - 1$.

Initialization of ℓ and r is done based on coding of the input. That is, initially ℓ is 0 and r is the input converted to number as above (similar to conversion of β above).

Now the symbol being read by TM is r mod b.

To write a on the cell being read by TM, one can change r to $\lfloor \frac{r}{b} \rfloor * b + a$.

To move right, we change ℓ to $\ell * b + (r \bmod b)$, and then r to $\lfloor \frac{r}{b} \rfloor$.

To move left, we change r to $r * b + (\ell \bmod b)$, and then ℓ to $\lfloor \frac{\ell}{b} \rfloor$.

Change of state by TM can be implemented by jumping to appropriate instruction in the program (we may need several instructions to implement each step of TM as we may need to do several of operations as mentioned above for each TM step).

It is left as an exercise to show that each of the arithmetic operations as above can be done in assembly language Z, without destroying the original number (by making copies).