jn 2004.05.13

8. Equivalence of TMs, PMs and Markov algorithms

Goals: Prove the equivalence of three rather different models of computation. This equivalence supports the Church-Turing thesis that any of these models captures the concept of effective computation.

8.1 Overview of concepts and proofs

Of the three models of computation considered in this chapter: Turing machines (TM), Post machines (PM) or queue machines (QM), and Markov algorithms (MA), TMs are the most versatile ones. The TM programmer can lay out his data along the tape in any convenient manner, designing “labels” of his choice to identify the various parts. The finite state controler can be programed to seek any desired data item by scanning the tape until the corresponding label is encountered. Thus, a TM can be programed to simulate direct access to its store, albeit with a tremendous slow-down as compared to a RAM.

Post machines, by contrast, suffer from an access restriction to their store, a FIFO queue. A symbol is read and deleted from the head of the queue, symbols are appended to the tail of the queue. It turns out, however, that a FIFO queue (unlike a LIFO queue, i.e. a stack) can simulate a tape that supports local left-or-right-moves of the read/write head. To see how this works, imagine the head and the tail of the FIFO queue to be glued together, producing a circular tape. The operation ‘delete the symbol at the head and append it at the tail’ corresponds to a circular left shift, if the head of the queue is pictured as facing left. A circular right shift can be simulated by L-1 circular left shifts, where L is the current length of the queue. Thus, crafty programming can overcome the access limitation of the FIFO queue and simulate a TM tape.

The finite state controler of TMs and PMs acts like a goto-program - the program counter can jump from anywhere to anywhere. Since gotos can be used to implement any control structure, the primitive but powerful FSM control of TMs and PMs imposes no limitations. The control structure of Markov algorithms, on the other hand, is strictly sequential. Scanning the rewrite rules from first to last, and the data string from left to right, the **first** rewrite rule that applies, is executed at the **first** pattern match. There is no explicit mechanism to express a goto of the kind “now continue using rewrite rule number so-and-so”.

Nevertheless, the sequential nature of Markov algorithm control structures is able to simulate the unrestricted jumps of an FSM. Consider a subset of rewrite rules that mentally we wish to associate with a ‘state’. We design labels for each of the states, say q1, q2, .., making sure that these labels remain distinguishable from anything else that may appear on the data string. Any rewrite rule L -> R intended to transforms the data string D in a desired manner is written as qi L -> qj R. If D initially contains the label q1, refering to a ‘starting state’ q1, then rules are executed from their subsets exactly as outgoing transitions are executed from their states. We use this simulation of goto-structures when designing Markov algorithms that mimic FSM control.

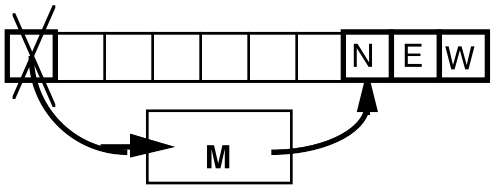
In view of the comments above, and our experience with TMs, it is no surprise that TMs can simulate both Markov algorithms and PMs. In the following proofs of equivalence we will directly use only the first of these two assertions. We denote it by TM ≥ MA, meaning “TMs are at least as powerful as Markov algorithms”. We then continue with two less obvious assertions: MA ≥ PM ≥ TM. Once these assertions are proved we close the circle TM ≥ MA ≥ PM ≥ TM that proves the equivalence of all three models of computation.

A last introductory comment about the concept of “universal machine or algorithm”. In proving any of the inequalities TM ≥ MA ≥ PM ≥ TM, we could aim to present a universal instance of its class of machines. In other words, in proving PM ≥ TM we could present a universal PM that has on its data string descriptions d(M) and d(T) of an arbitrary TM M and its tape T, and which proceeds to intrepret these descriptions step by step. This results in complex machines that spend their time mostly in pattern matching and copying.

We will instead proceed in a simpler and more efficient way, akin to the way that compilation is more efficient than interpretation. In order to prove PM ≥ TM, or any of the other inequalities, we assume an arbitrary TM M is given, and we construct a PM P tailor-made to simulate the specific TM M. Since we know that there exist universal TMs, U, and since Markov algorithms and PMs can simulate any TM, including a universal TM U, it follows that there are universal Markov algorithms and universal PMs.

8.2 Post machines or tag machines

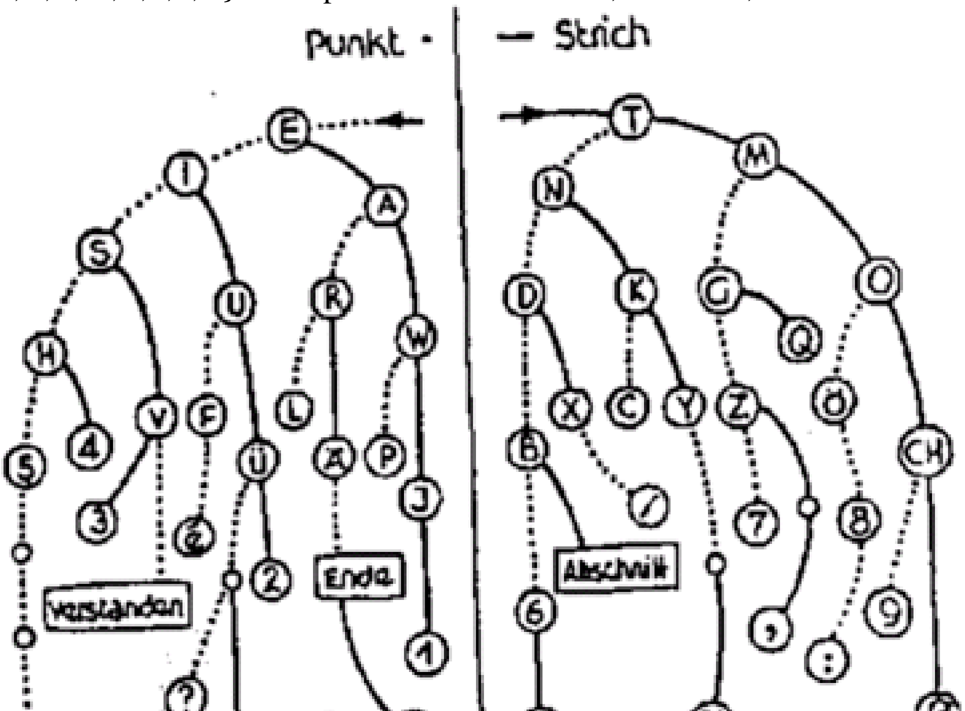
A Post machine (Emil L. Post, 1897-1954) or tag machine is a FSM with a single tape of unbounded length with FIFO access (first-in first-out, as opposed to the LIFO access of a stack). In a single transition, M reads and deletes the symbol at the head of the FIFO queue and may append symbols to the tail of the queue. Technical detail: there is a special symbol #, not part of the alphabet A of the input string, typically used as a delimiter. Perhaps surprisingly, Post machines are universal, i.e. equivalent in computational power to TMs.



We use the following conventions in order to simplify the examples that follow. $A = \{0, 1\}$ is the alphabet of the language to be accepted, or the functions $f: A^* \to A^*$ to be computed. We introduce additional marker symbols that may be useful. We allow the FSM controler to inspect not only the symbol at the head of the queue, but any prefix of constant length. This generalization greatly simplifies the programming task. In order for the PMs to be deterministic, we insist on the prefix property: for any state q, and any two outgoing transitions q, x -> ..., and q, y -> ..., with $x, y \in A^*$, x is not a prefix of y. The ability to inspect a a prefix of fixed length can be eliminated at the cost of introducing additional states, one for each symbol of the prefix under inspection.The examples illustrate a general procedure for reducing a PM that observes a string at the head of the queue to another PM that only observes a single symbol.

Transitions are of the form: qi, x -> qj, y with $x, y \in A^*$. For x or y = ε we may omit ε. E.g. qi, x -> qj cuts x from the head without appending anything to the tail, and qi -> qj, y appends y to the tail regardless of the state of the head of the queue. There are two special types of transition, halting and test for empty queue. We write them as as: qi, x -> y, Halt and qi, Empty -> qj, y. Notice: qi, Empty applies to the total state of the queue, whereas qi, ε -> refers to the prefix ε of the queue which is always present and ignored.

Ex1: Translate an alphabetic source text into Morse code in a single left-to-right scan.
Samuel Morse (1791-1872) demonstrated the ability of a telegraph sytem to transmit information over wires. Alphabet A = { _ , a, b, c, .. , z, •, -, / }. Sample Morse codes: a: • -; b: - • • •, ... as shown in this code tree:



The Morse code lacks useful properties that make decoding easy, such as the prefix property (no code word is a prefix of any other). Instead, it relies on a pause of sufficient length to indicate the end of a letter; we use _ to indicate this pause. We use / as a word separator in the source text and translate it to - • • - • in the Morse code.

A single-state PM whose transition from q to q is labeled by the entire dictionary as given in the code tree above will transform any text into Morse code. E.g., the transitions: q, S -> q, •••_ and q, O -> q, ---_ , along with q, # -> #, Halt translates the input 'SOS#' into '•••_ ---_•••_ #' .

Ex2: Parsing a context-free language of suffix expressions

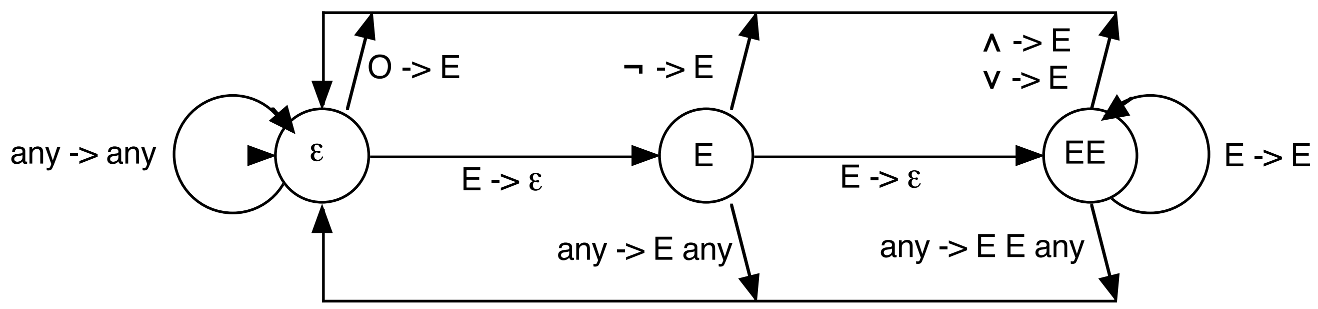
The following CFG generates the language of Boolean suffix expressions that involve the operators ¬ ∧ ∨, and a single operand symbol, o: E –> o | E¬ | EE∧ | EE∨ .
A single-state PM, whose transitions are "reverse productions", parses any suffix expression (terminated by the special symbol #) using repeated scans until the expression is reduced to the single symbol E, and the string to E#. Call this process 'reduce or rotate': if the subexpression currently at the head of the queue can be reduced (parsed), do so; if not, rotate the symbol at the head of the queue to the tail. Eventually, it will make it's way back to the head and get another chance to be reduced. The transitions from q to q are of four types:

parsing: o –> E, E¬ –> E, EE∧ –> E, EE∨ –> E,
rotating characters that cannot [currently] be parsed: 'any other' -> 'any other'
successful recognition of [a part of] the input as an instance of E: E# -> ε
test whether the entire input string has been processed: q, Empty -> Halt

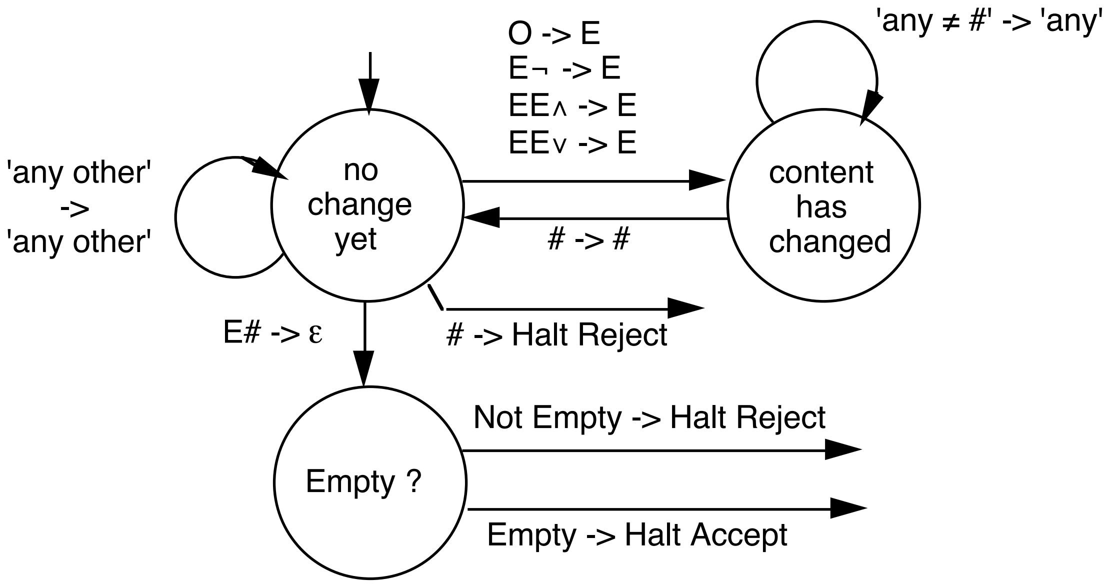
E# -> ε is the final transition of the parsing phase, whose purpose it is to transform a syntactically correct string into ε. The halting transition q, Empty -> Halt terminates the parsing of syntactically correct inputs, whereas incorrect inputs cause the machine to loop. This fair weather parser works well on syntactically correct inputs, as shown by the following execution trace on the initial string o ¬ o ∧ # :

```
o ¬ o ∧ #                 E ¬ E ∧ #              E E ∧ #           E #      ε
  ¬ o ∧ # E                   E ∧ # E                # E
    o ∧ # E ¬                   ∧ # E E
      ∧ # E ¬ E                   # E E ∧
        # E ¬ E ∧
```

A single state suffices for this FSM controler because this PM inspects prefixes of length > 1. The following PM that inspects only the single symbol at the head of the queue does the same job. Its state labeled EE remembers that the string EE has been cut off from the head of the queue, and the PM awaits the next symbol to decide what to do about this incomplete substring EE. Three cases must be distinguished:
1) ∧ or ∨ complete a subexpression E E ∧ or E E ∨ , respectively, which is parsed to an E;
2) yet another E creates a substring E E E (as in E E E ∧ , for example) of which the first E cannot be processed now, and hence is rotated to the end of the queue for later re-consideration. But the two most recent Es might be expanded to E E ∧ , for example, and this explains the self-loop E -> E.
3) A transition labeled 'any -> x any' out of a state q applies to any symbol not explicitly attached to a transition out of this same state q. The symbol denoted by 'any' is cut off as usual, and the string 'x any' is appended to the tail of the queue, where this second 'any' denotes the symbol cut off. This transition effects a delayed rotation of the substring x, which had been accumulated in the PM's state space to be appended later.



A Post machine that distinguishes correct inputs from incorrect inputs and always halts must break the endless loop when no progress occurs during a complete rotation of the queue content, from one ocurrence of # at the head of the queue to the next. The following two-state machine keeps track of any change that occurs, triggered by a transition of the type "reverse production" to the state labeled 'content has changed'. If no such change occurs during a complete rotation, the next appearance of # at the head of the queue breaks the loop. The 'E' in a pair of consecutive symbols ‘E#’ can never be removed by any reverse production, so it is the root of a parse tree for a suffix expression. If the queue consists of just E#, then the entire input must have been an instance of E. If, on the other hand, the queue contains other symbols besides E#, then the original input contained extraneous symbols that cannot be part of a suffix expression. Thus, after removing E#, a test whether the queue is empty or not decodes syntactic correctness.

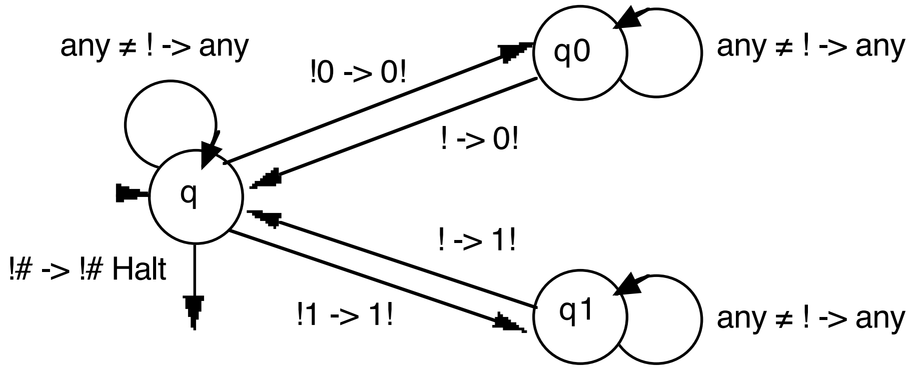


Ex3: Duplicating a string

The two previous tasks are simple because they are naturally understood as involving left-to-right scans. Many string processing operations are intuitively understood by moving the read/write head either left or right, as a Turing machine does, wherever the next task happens to be. In a PM, any desired sequence of moves must be resolved in terms of left-to-right scans, similar to the way a Markov algorithm processes its data.

Consider the task of copying or duplicating a string, e.g. 011. If we wanted to turn it into 001111, the single scan translation used for Morse coding does the job. But we want to generate 011 011.

Initial string:	# ! 0 1 1 # !	# separates the two bit strings. ! marks the place where things happen
Intermediate:	# 0 ! 1 1 # 0 !	Invariant: the bit string to the left of ! has already been copied
Final string:	0 1 1 ! # 0 1 1 ! #	There is nothing left to do to the right of !
		We omit a clean up phase to remove “!”



Execution trace on the initial string # ! 0 1 1 # ! (consecutive rotations are compressed into one):

```
q : # ! 0 1 1 # !
q :   ! 0 1 1 # ! #
q0:       1 1 # ! # 0 !
q0:             ! # 0 ! 1 1 #
q :               # 0 ! 1 1 # 0 !   the first symbol has been copied
q :                   ! 1 1 # 0 ! # 0
q1:                       1 # 0 ! # 0 1 !
q1:                             ! # 0 1 ! 1 # 0
q :                               # 0 1 ! 1 # 0 1 !   2nd symbol copied
```

```
q :                                    ! 1 # 0 1 ! # 0 1
q1:                                        # 0 1 ! # 0 1 1 !
q1:                                               ! # 0 1 1 ! # 0 1
q :                                                 # 0 1 1 ! # 0 1 1 !
q :                                                           ! # 0 1 1 ! # 0 1 1
q :                                                               0 1 1 ! # 0 1 1 ! #
Halt
```

8.3 Turing machine simulates Markov algorithm

Intuitively, Turing machines are the most agile of the three models of computation considered in this chapter, since their read/write head can move either **left or right**, in contrast to the **left-to-right** access of Post machines and Markov algorithms. Thus, it is easy to understand the principle of how a Turing machine can simulate a Markov algorithm, though the details may be laborious. A Turing machine that reads the description of an arbitray Markov algorithm, suitably encoded on its 2-d data sheet, and simulates it, can be seen in the Turing Kara software: www.educeth.ch/compscience/karatojava/turingkara .

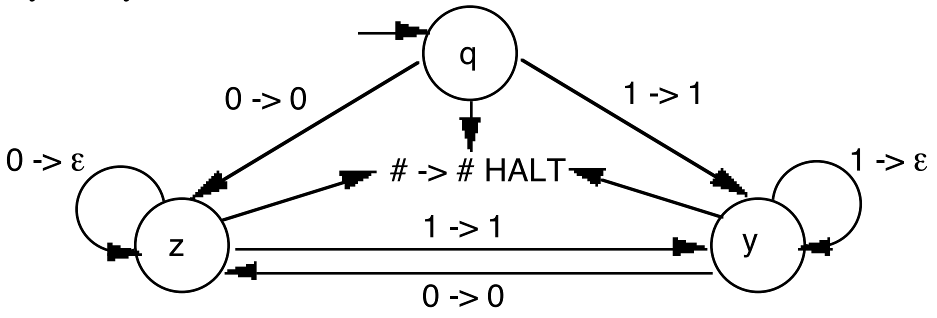
8.4 Markov algorithm simulates Post machine

Given an arbitrary Post machine P we construct a Markov algorithm M that simulates P. M has one rewrite rule for each transition of P, and a few other rules that move newly created characters to the right end of M's data string, corresponding to the tail of P's queue. For simplicity's sake we restrict P as follows:
P's alphabet is { 0, 1, #}, and P's transitions are of the form q, x -> q', y with $|x| \le 1$ and $|y| \le 1$.

In general, M's alphabet can be chosen as {0, 1, #, q} plus a single marker α. The following algorithm that compresses runs of 0s into a single 0, and runs of 1s into a single 1 serves as an example to illustrate the construction of M's rewrite rules.

Ex: Compress runs of the same symbol into a single symbol, e.g. 00011011100# - > 01010#.

Structure of P. The starting state q determines whether the very first run is a run of 0s or of 1s, rotates this symbol, and transfers control to state z, "zeros", or y, "ones", accordingly. State z remembers that P is currently reading a run of 0s, that this run has already generated a 0 in the output, hence it deletes all further 0s of this run. Analogously for state y. In any state, # terminates the transformation.



For the sake of simplicity, we choose the alphabet of the Markov algorithm M as {0, 1, #, q, z, y }, where q, z, y are the names of the corresponding states. This convention suggests that M's alphabet expands with the size of the Post machine P to be simulated. From a theoretical point of view, however, it is more elegant to have a fixed Markov alphabet for any Post machine to be simulated. If P has states q1, q2, .. qs, for example, M can code these, for example, as q0q, q00q, q0...0q with the fixed alphabet {0, 1, #, q}.

The Markov algorithm M that simulates P codes the input and result string as: q00011011100# - > 01010 .
The input starts with the identifier of P's starting state q. The other states are coded as z and y.

M consists of 4 sets of rewrite rules, one for each state of P, and one for moving newly created characters to the right end of M's data string. The rewrite rules associated with the states are in 1-to-1 correspondence with P's transitions - thus, this part of the Markov algorithm is a mirror image of P. We follow the convention that rules written on the same line can appear in any order, whereas rules written on different lines must be executed from top to bottom. The set of rules that move a character must appear first, the rules correesponding to P's states can follow in any order.

Move a newly created character to the right:

1) α 0 0 -> 0 α 0, α 0 1 -> 1 α 0, α 0 # -> # α 0, α 1 0 -> 0 α 1, α 1 1 ->1 α 1, α 1 # -> # α 1
2) α -> ε

Rules for starting state q: 3) q 0 -> z α 0 , q 1 -> y α 1 , q # ¬ # (terminating rule)

Rules for state z: 4) z 0 -> z, z 1 -> y α 1, z # ¬ # (terminating rule)

Rules for state y: 5) y 1 -> y, y 0 -> z α 0, y # ¬ # (terminating rule)

Initialization: 6) ε -> q

Execution trace on the input 1 1 0 1 1 # with result 1 0 1 #.

```
            1 1 0 1 1 #          Rule 6 applies
   6)    q 1 1 0 1 1 #           Rule 3 applies
   3)    y  α 1 1 0 1 1 #        Rule 1 applies repeatedly, denoted by 1)*
   1)*   y 1 0 1 1 # α 1
   2)    y 1 0 1 1 # 1
   5)    y 0 1 1 # 1
   5)    z α 0 1 1 # 1
   1)*   z 1 1 # 1 α 0
   2)    z 1 1 # 1 0
   4)    y α 1 1 # 1 0
   1)*   y 1 # 1 0 α 1
   2)    y 1 # 1 0 1
   5)    y # 1 0 1
   5)    1 0 1 #
```

Explanation of the rules. Every transition of the form q, u -> q', v of a Post machine, when executed, causes a corresponding Markov rule of the form q u -> q' α v to be executed. The only difference between these two is that the Post machine appends v at the right end, whereas the Markov algorithm inserts v towards the left end of the data string. Therefore, the Markov rule generates a marker α whose task it is to push v to the far right. The rules involving α have top priority, so that the right shift of v gets finished before any new Post transition is simulated.

8.5 Post machine simulates Turing machine

Let the TM M have the alphabet A = { 0, 1, <, > }, where the angular brackets are used to delimit the finite portion of the tape that initially contains the input, and later gets expanded to delimit that portion of the tape that has been visited so far. Let the M's transition function be of the form f: Q x A -> Q x A x { L, R }.
We sketch the design of a Post machine P that simulates M using the alphabet B = { 0, 1, <, >, # }. The basic idea is straightforward, the details are intricate and will be skipped.

Consider a configuration where M is in state q and currently reads the (bold) symbol x of the tape:
M: < B y **x** z C >, with y, x, z ∈ A and B, C ∈ A*. That is, we focus attention on what happens to x and to its two neigbor symbols y to the left and z to the right, and merely carry along the remaining portions of the tape, B and C. In this configuration, M executes either
a move-right transition q, x -> q' x' R or a move-left transition q, x -> q' x' L

The Post machine P's state space includes a subspace that is in 1-to-1 correspondence with the state space Q of the TM M. Without danger of confusion, we call this subspace Q and designate its states with the same label as the corresponding state of M. When refering to a state q, the context will make it clear whether we mean the state q_M of M or q_P of P. P's state space includes additional states, to be introduced as needed, because in general a single transition of M requires a sequence of transition s of P.

M’s configuration: state q, tape   < B y **x** z C >   is modeled as
P’s configuration: state q, tape   **x** z C > < B y

This encoding is practically forced, since P must inspect the same symbol **x** as M does, but the only symbol it can inspect is at the head of the queue. As M executes a transition q, x -> q’ x’ R or   q, x -> q’ x’ L , P must be programed to transform its queue in a corresponfding manner. First try, which doesn’t quite work.

The easy case, a move-right transition q, x -> q’ x’ R :
M: q,   < B y **x** z C >   becomes   q’, < B y x’ **z** C >
P: q,   **x** z C > < B y   becomes   q’, **z** C > < B y x’
A single P transition achieves precisely the required tranformation.

The cumbersome case, a move-left transition q, x -> q’ x’ L :
M: q,   < B y **x** z C >   becomes   q’,   < B **y** x’ z C >
P: q,   **x** z C > < B y   must become   q’,   **y** x’ z C > < B

P’s transformation requires a complete rotation of the queue in order to bring the tail symbol to the head of the queue. A rotation is possible if there is a distinguished symbol # that tells us when to stop. Thus, the tentative encoding used so far fails for a move-left transition and must be corrected.

Correct encoding of M’s configuration in P’s configuration: place a marker # two slots to the left of the scanned symbol x. “To the left” is interpreted in a circular way, as if head and tail were glued together, and hence # appears as the next-to-last symbol in the queue.

M’s configuration: state q, tape   < B y **x** z C >   is modeled as
P’s configuration: state q, tape   **x** z C > < B # y

Simulating a move-right transition q, x -> q’ x’ R :
M: q,   < B y **x** z C >   becomes   q’,   < B y x’ **z** C >
P: q,   **x** z C > < B # y   becomes   q’, **z** C > < B y # x’
Unfornately, this former “easy case” has become more complicated, since the consecutive pair # y must be permuted to y #. This can be achieved using auxiliary states and a complete rotation.

Simulating a move-left transition q, x -> q’ x’ L,
focus on the **two** neighboring symbols **to the left** of the scanned symbol x:

M: q,   < B y z **x** C >   becomes   q’, < B y **z** x’ C >
P: q,   **x** C > < B y # z   becomes   q’, **z** x’ C > < B # y
Again, the consecutive pair # y must be permuted to y #, achieved using auxiliary states and a complete rotation.

Two more cases must be considered, when M’s read/write head is positioned at either end of the tape visited so far, on a symbol < or >, and possibly extends the visited portion. In summary, the design of a PM that simulates an arbitrary TM is conceptually straightforward once one has mastered the acrobatics of P’s queue rotations.

Thus, we have completed the cycle TM ≥ MA ≥ PM ≥ TM that proves the computational equivalence of these three universal models of computation.

Reference: Solvability, Provability, Definability: The Collected Works of Emil L. Post, Martin Davis (ed.), Birkhaeuser 1994. (Includes the article “Emil L. Post: His Life and Work”)

Homework: For each of the following problems, design a Markov algorithm (MA), a Post machine (PM), and a Turing machine (TM). You are free to choose the detailed conventions of these machines, as well as the coding of the input and output, as long as you explain precisely what conventions you are using.

1) Given a natural number k > 0 in unary notation, i.e. as 1^k, produce the output a^k b^k.
Example of possible input/output coding:
MA: initial data string 1 1 ... 1;   final string a a ... a b b ... b
PM: initial queue content 1 1 ... 1 #;   final content   a a ... a b b ... b #
TM: initial tape content   # 1 1 ... 1 --- with the read/write head positioned on #, the tape extending to the right;

final tape content # a a ... a b b ... b ---.

2) Given two natural numbers $i > 0, j > 0$ in unary notation, produce the output min(i, j) in unary.

3) Given two natural numbers $i > 0, j > 0$ in unary notation, produce the output max(i, j) in unary.

4) Optional: Given two natural numbers $i > 0, j > 0$ in unary notation, produce the output LCM(i, j) in unary.
LCM denotes the Least Common Multiple.
Hint: In Exorciser you will find a Markov algorithm for computing the Greatest Common Divisor GCD(i, j).

End of Chapter 8