The Church-Turing Thesis

Turing Machines and Effective Computation

Monday, December 3, 2007
Reading: Sipser 3; Kozen 28
Revised December 3, 2007

CS235 Languages and Automata

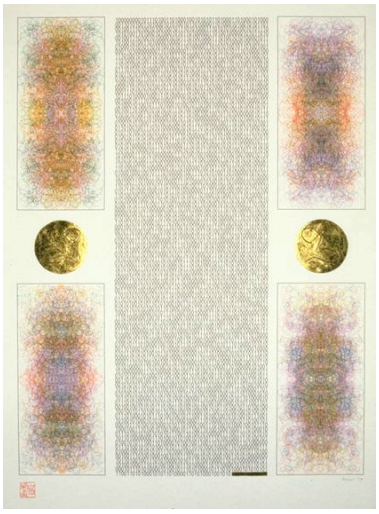
Department of Computer Science
Wellesley College

A Long Long Time Ago

- Mathematicians were wrestling with the notion of effective computation: formalisms for expressing algorithms.
- Many formalisms evolved:
  - Turing Machines (Turing);
  - Post systems (Post);
  - μ-recursive functions (Gödel, Herbrand);
  - λ-calculus (Church, Kleene);
  - combinatory logic (Schöfinkel, Curry).
- All of these formalisms were proven to be equivalent to each other!

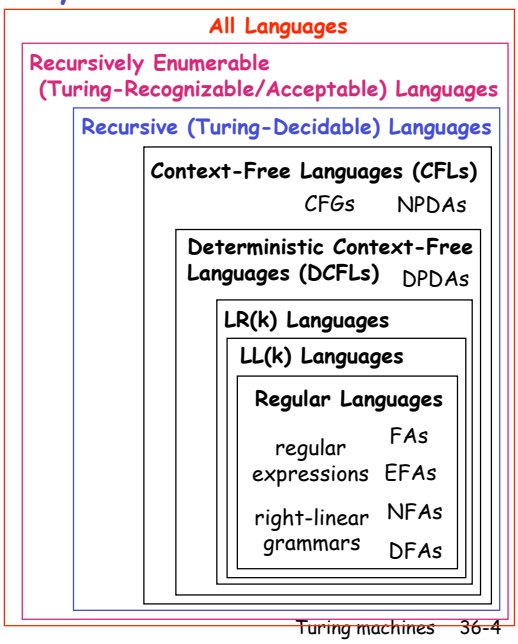
The Church-Turing Thesis

- Computability is the common spirit embodied by this collection of formalisms.
- This thesis is a claim that is widely believed about the intuitive notions of algorithm and effective computation. It is not a theorem that can be proved.
- Because of their similarity to later computer hardware, Turing machines have become the gold standard for effectively computable.
- We'll see in CS251 that the λ-calculus formalism is the foundation of modern programming languages.



What We'll Start Today

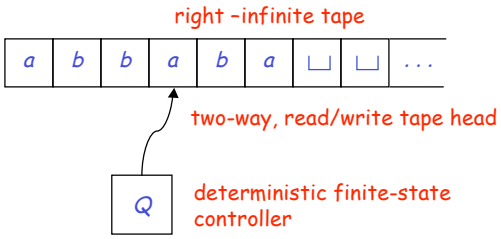
- Define Turing Machines and give examples of them.
- Show that adding various features to Turing Machines does not increase their power.
- Begin to extend the language hierarchy we've seen thus far.



What Is A Turing Machine?

Model of computation proposed by Alan Turing in 1936:

- A one-way infinite tape of cells holding symbols or blanks.
- A tape head that can read/write the symbol in the cell under it.



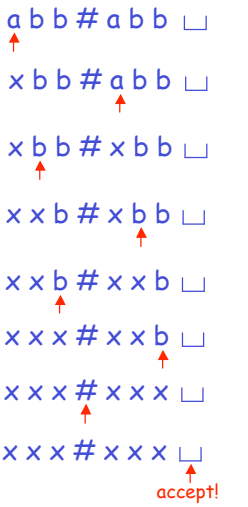
- A deterministic finite-state controller that, based on the current state and symbol under the tape head, writes a symbol under the tape head and moves it left or right.

Execution begins in the initial state with a string written on the tape starting at the leftmost cell

- The string is accepted if the controller enters the accept state.
- The string is rejected if the controller enters the a reject state.
- The machine may also loop -- continue processing the string forever without accepting or rejecting it.

Informal TM Example: {w#w | w ∈ {a,b}*}

1. If there is no unprocessed symbol in the left substring
   a. If there is no unprocessed symbol in the right-hand substring, accept.
   b. Otherwise, reject.
   Otherwise, cross off first unprocessed symbol in the left substring and remember it.
2. Move to first unprocessed symbol in the right substring.
   a. If there is no such symbol (all the right-hand symbols have been crossed off) or the symbol does not match the remembered one, reject.
   b. If the symbol matches the remembered one, cross it off.
3. Move back to the first unprocessed symbol in the left substring (if there is one, or back to #, otherwise) and go to step 1.

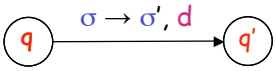


TM Transition Function

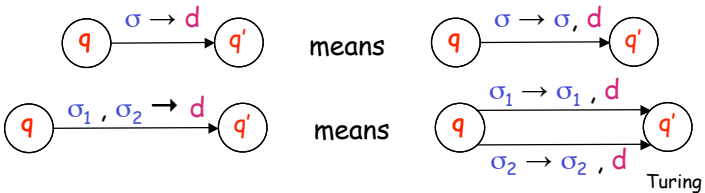
The core of a Turing Machine specfication is the definition of a deterministic transition function δ :

(current-state, current-symbol) → (next-state, new-symbol, left-or-right)

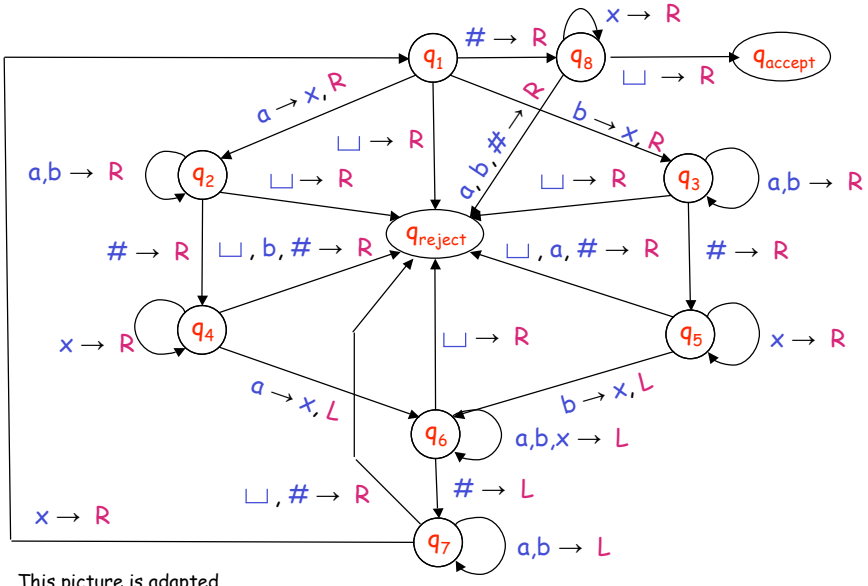
The transition $(q,\sigma) \rightarrow (q', \sigma', d)$ is typically represent as either:

- An entry in a 2D table keyed by q and σ;
- A state transition in a diagram: 

Abbreviations:

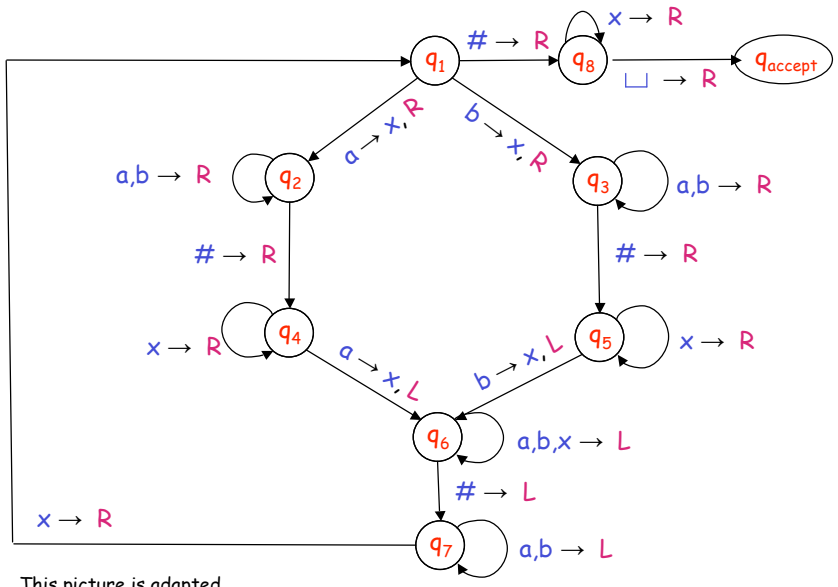


TM State Transitions for {w#w | w ∈ {a,b}*}



This picture is adapted from p. 145 of Sipser.

TM State Transitions w/Implicit Reject State



This picture is adapted from p. 145 of Sipser.

Formal Definition of a Turing Machine

A deterministic one-tape Turing machine is a septuple

$TM = (Q, \Sigma, \Gamma, \delta, q_{start}, q_{accept}, q_{reject})$

where

1. Q is a finite set of states;
2. Σ is a finite input alphabet, which does not include the blank symbol, ␣
3. Γ is a finite tape alphabet, where $(\Sigma \cup \{ ␣ \}) \subseteq \Gamma$.
4. $\delta \subseteq (Q \times \Gamma) \times (Q \times \Gamma \times \{L, R\})$ is the transition function;
5. $q_{start} \in Q$ is the start state;
6. $q_{accept} \in Q$ is the accept state;
7. $q_{reject} \in Q$ is the reject state, $q_{reject} \neq q_{accept}$.

(This is Sipser's formalism for a Turing machine. There are many variants.)

Turing Machine Languages

A configuration specifies the state and tape head position of a TM.

E.g. $xbb\#q_4abb$

The transition function defines a relation $\Rightarrow$ between transitions:

E.g. $xbb\#q_4abb \Rightarrow xbbq_6\#xbb$

A Turing Machine accepts a string w iff $q_{start}w \Rightarrow^* \ldots q_{accept} \ldots$

A Turing Machine rejects a string w iff $q_{start}w \Rightarrow^* \ldots q_{reject} \ldots$

The language of TM, L(TM) = all strings accepted by TM.

Comparison: TMs vs. FAs and PDAs

- TM is necessarily deterministic.
- TM can both read and write input cells.
- The tape head can move both left and right.
- The tape is infinite.
- Accept/reject states take effect immediately
  (don't have to read/consume all input).
- In all cases, the controller is finite-state!

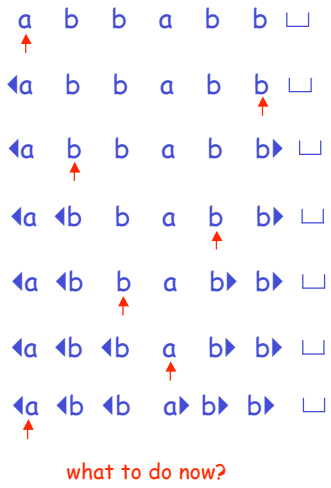
What about {ww | w ∈ {a,b}*}?

Idea: first transform a b b a b b

to ◂a ◂b ◂b a▸ b▸ b▸

where there are 4 new symbols:

◂a, ◂b , a▸ , b▸



What about {$a^n b^n c^n$ | n ∈ Nat}?

Subtlety: Detecting Left End of Tape

An attempt to move left from the leftmost tape cell leaves the position of the tape head unchanged.

How can the left end of the tape be detected? Two ways:

1. Put a special left-end marker symbol (often written $\vdash$) in the leftmost cell at the beginning of the computation. Sometimes another symbol (such as x in the w#w example) can serve this purpose.
2. When moving leftward, make a change to symbol in the current cell. If that change is still present after the left move, you're in the leftmost cell.

High-Level Algorithm Descriptions are OK!

We *can* give a formal TM description for any algorithm, but it would be extremely tedious to do so.

Instead, we will usually describe algorithms at a higher-level:

- A high-level English description of a Turing Machine or some other step-by-step process
- Pseudocode, such as in CLRS (CS231)
- A program in (or high-level description of a program in) our favorite programming language (SML, Scheme, Java, etc.), assuming it has infinite memory.

In high-level algorithms, we often want to manipulate various kinds of complex values, such as lists, trees, graphs, grammars, and descriptions of various kinds of automata themselves. We can represent these via appropriate strings that are checked for well-formedness.

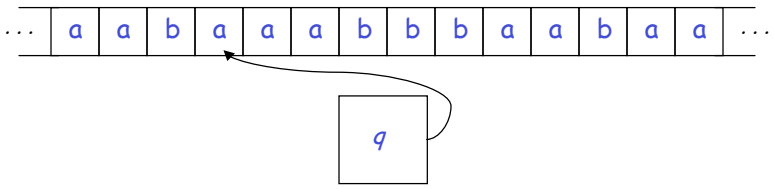
Turing Machine Variants

There are many variations on Turing Machines. E.g.:

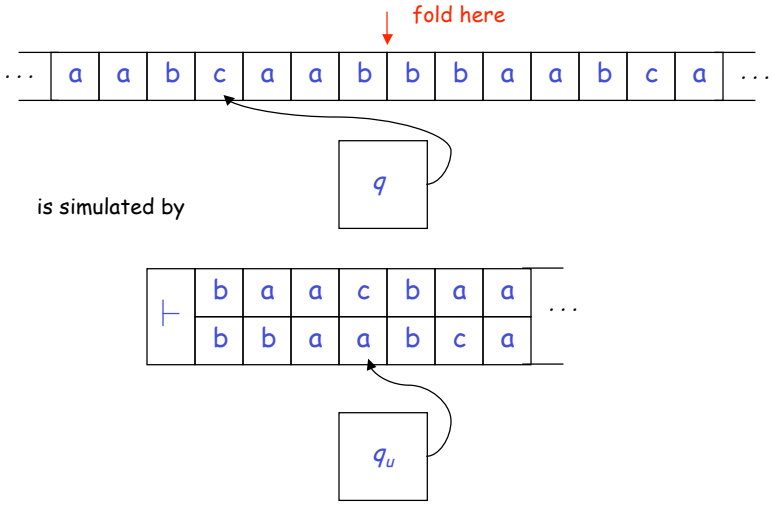
1. A TM with a head that can stay put (S) as well as move L or R.
2. A TM with a special left-marker symbol, $\vdash$ (Kozen)
3. A TM with a two-way infinite tape.
4. A TM with k tapes (k > 1).
5. A TM with a nondeterministic controller (δ is a transition relation rather than a transition function).

Remarkably, all of these have the same power (i.e., they describe the same language) as the simple deterministic one-way one-tape Turing Machine we've studied.

Two-way infinite tapes

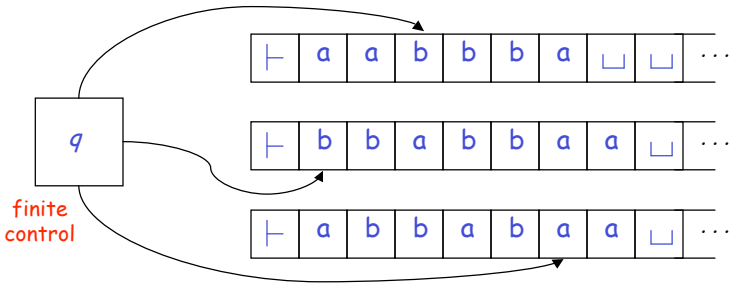


Just fold the tape somewhere



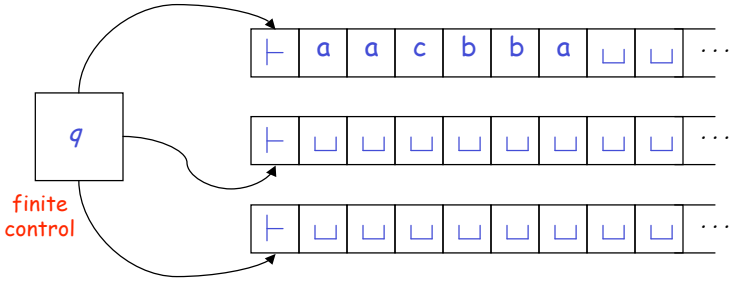
*Remember in state whether you are scanning the upper or lower track.

Multiple tapes

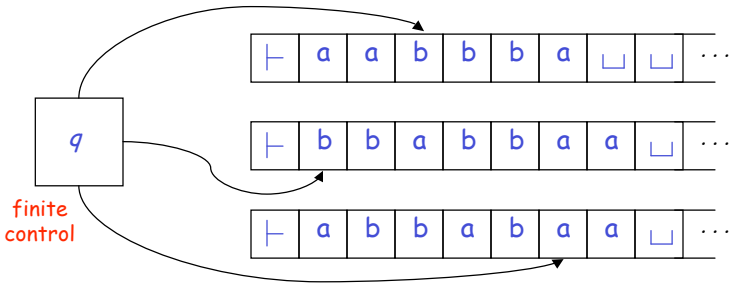


*Transition function has type $\delta : Q \times \Gamma^3 \rightarrow Q \times \Gamma^3 \times \{L, R\}^3$.

Start configuration of a multitape machine

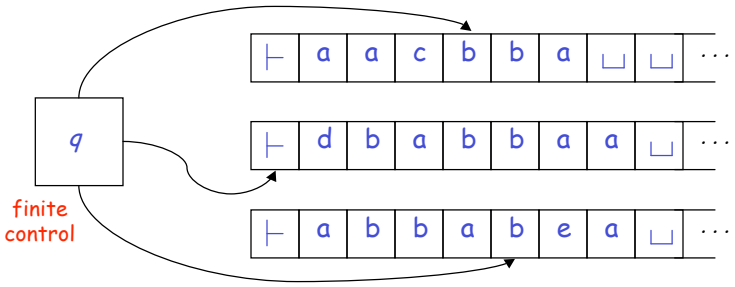


In one step



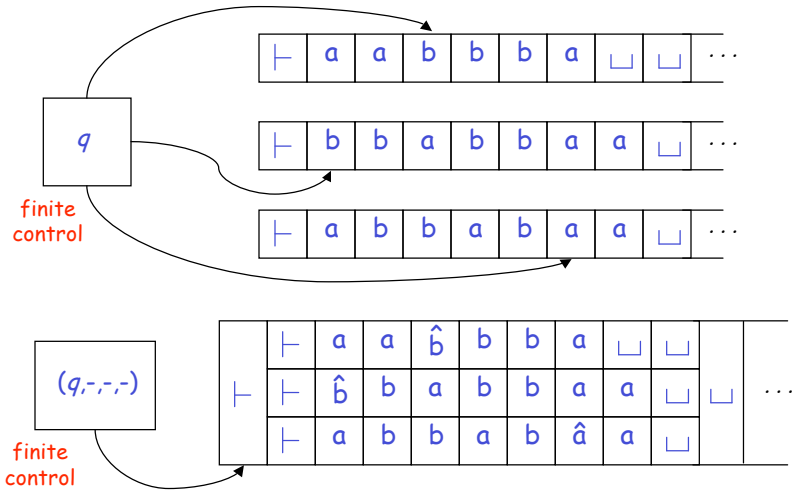
*M follows transition $\delta(q, b, b, a) = (p, c, d, e, R, L, L)$.

Multiple writes and moves



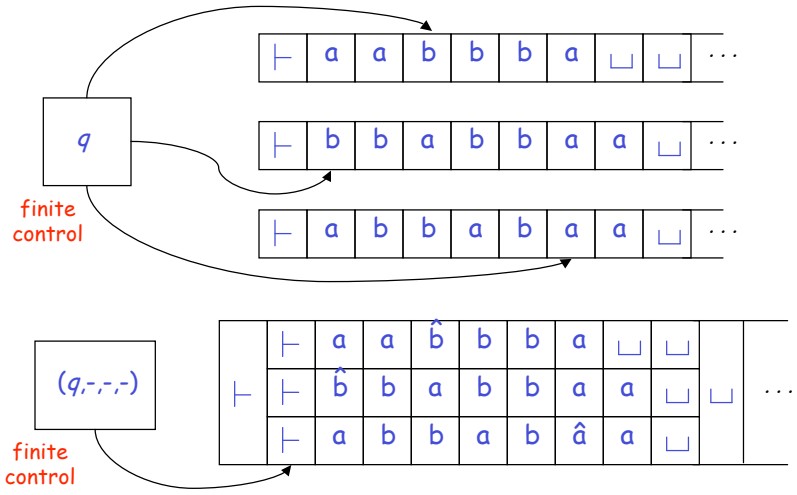
*M done followed transition $\delta(q, b, b, a) = (p, c, d, e, R, L, L)$.

Simulating multiple tapes



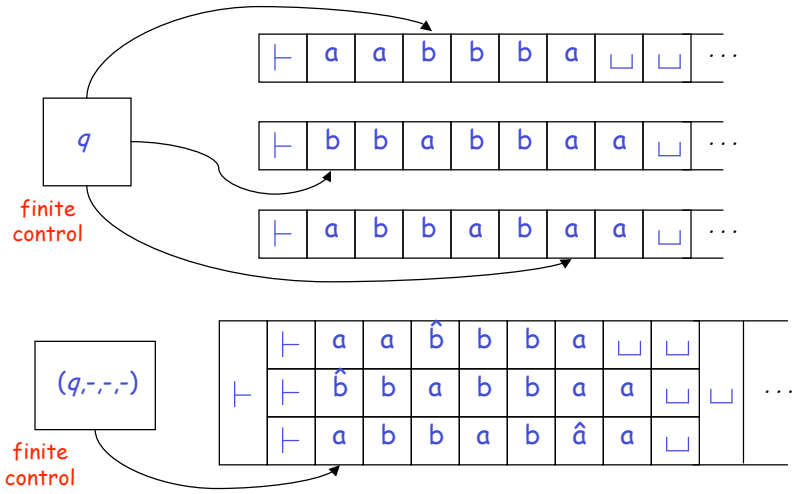
Simulate transition

$\delta(q, b, b, a) = (p, c, d, e, R, L, L)$

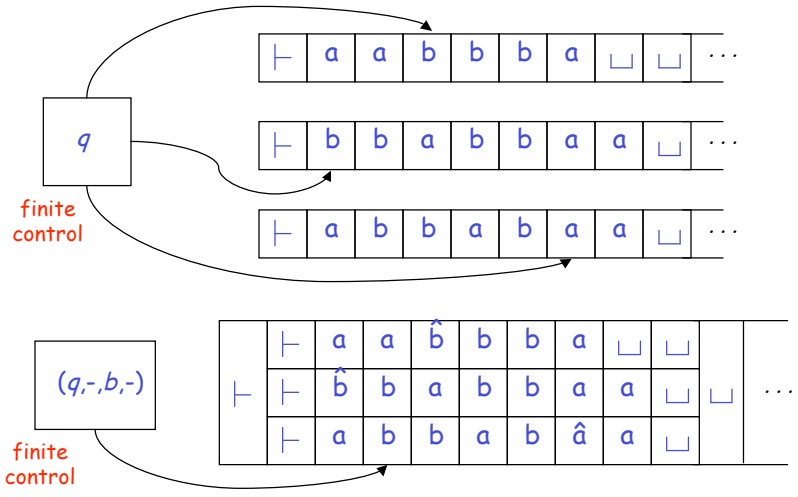


Found position of head on tape 2

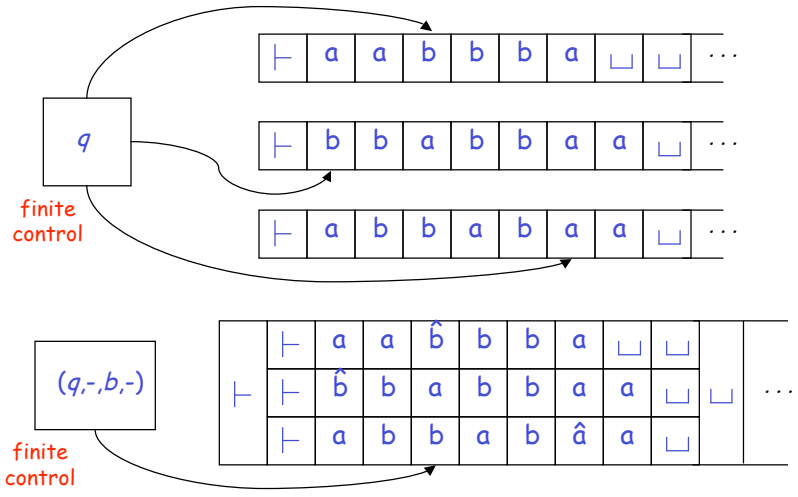
$\delta(q, b, b, a) = (p, c, d, e, R, L, L)$



Remember symbol under head on tape 2 in state $\delta(q, b, b, a) = (p, c, d, e, R, L, L)$

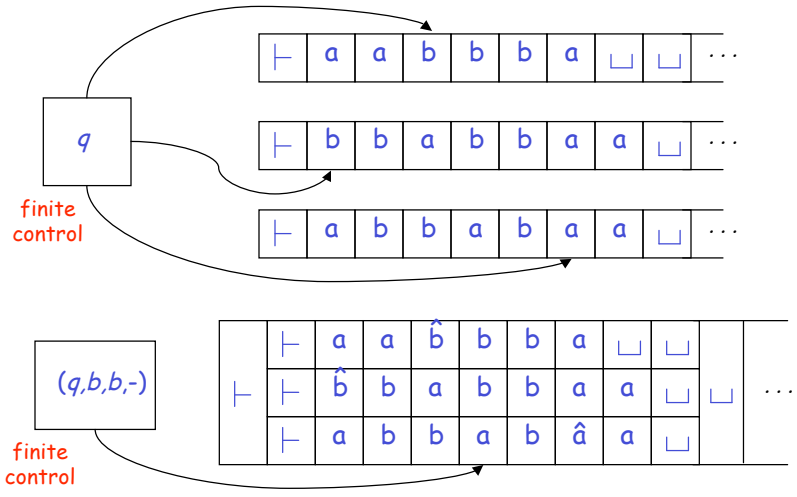


Found position of head on tape 1 $\delta(q, b, b, a) = (p, c, d, e, R, L, L)$



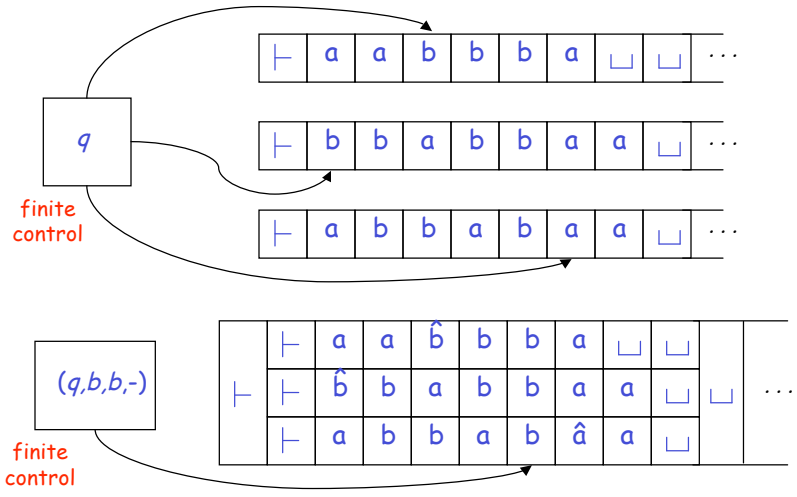
Remember symbol under head of tape 1 in state

$\delta(q, b, b, a) = (p, c, d, e, R, L, L)$



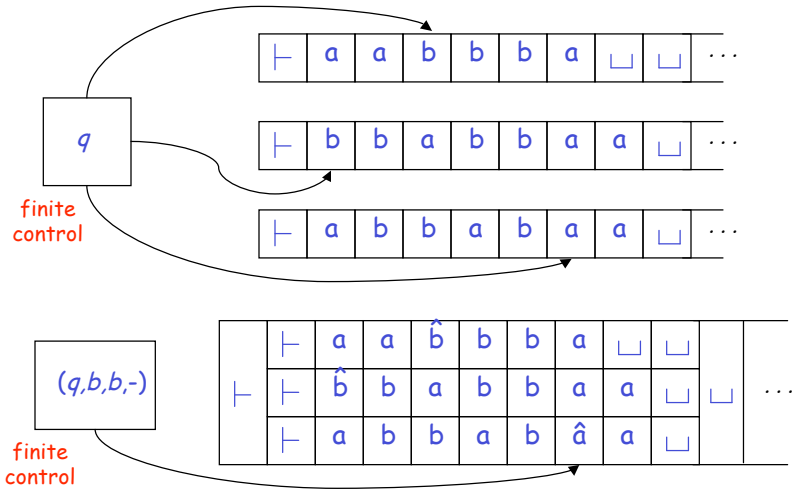
Still scanning

$\delta(q, b, b, a) = (p, c, d, e, R, L, L)$



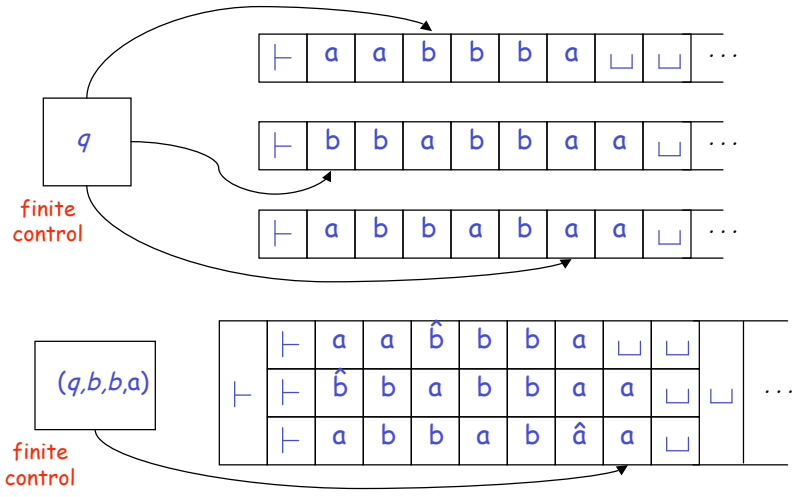
Eureka!

$\delta(q, b, b, a) = (p, c, d, e, R, L, L)$



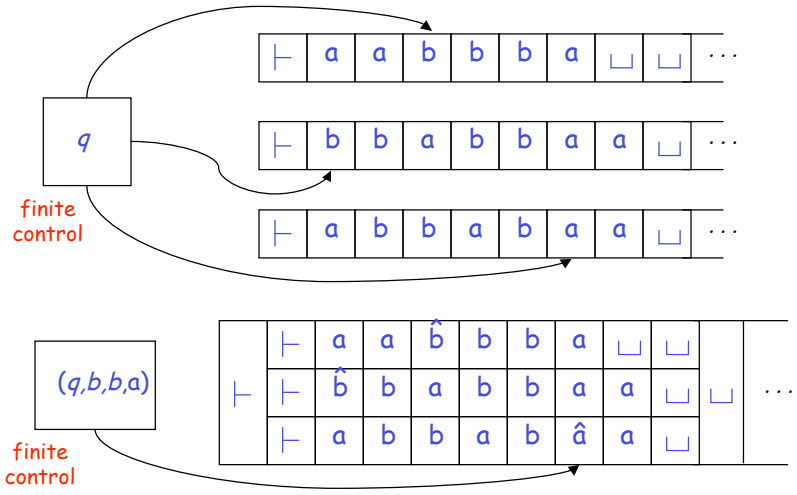
Remember symbol under head of tape 3

$\delta(q, b, b, a) = (p, c, d, e, R, L, L)$



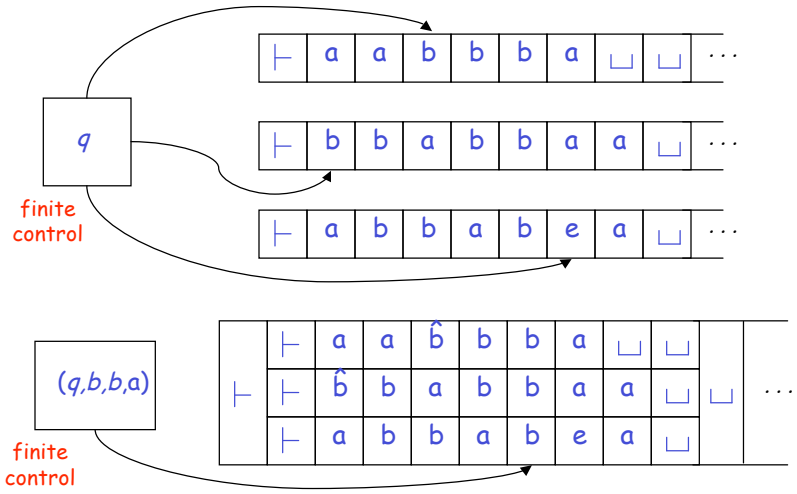
Back up & update tape

$\delta(q, b, b, a) = (p, c, d, e, R, L, L)$



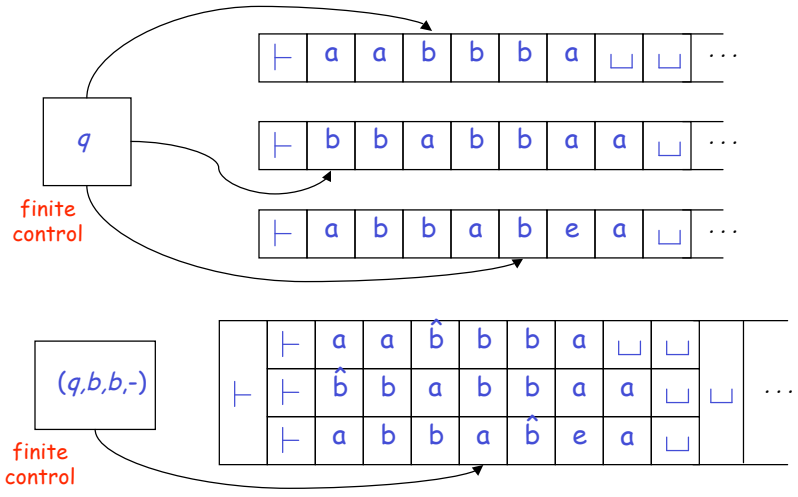
Write "e" into track 3

$\delta(q, b, b, a) = (p, c, d, e, R, L, L)$



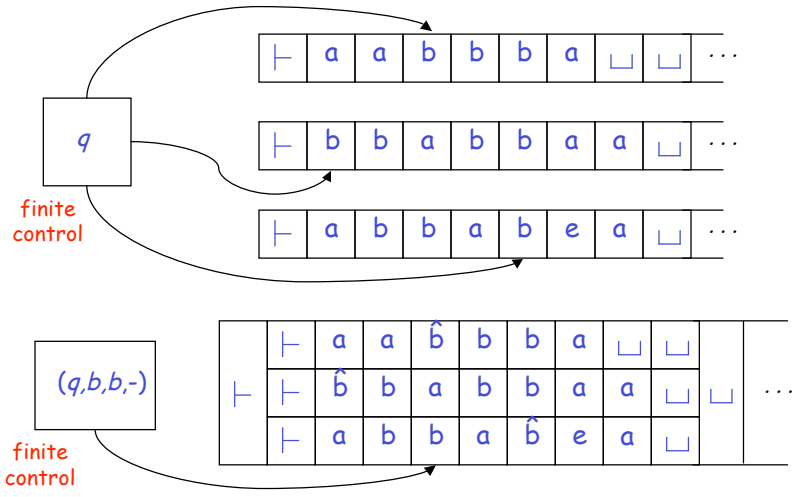
And update head position on tape 3

$\delta(q, b, b, a) = (p, c, d, e, R, L, L)$



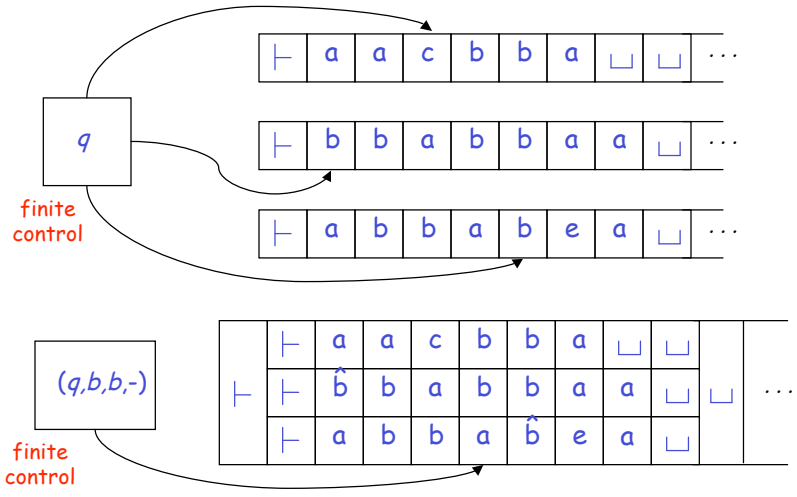
Prepare to update track 1

$\delta(q, b, b, a) = (p, c, d, e, R, L, L)$



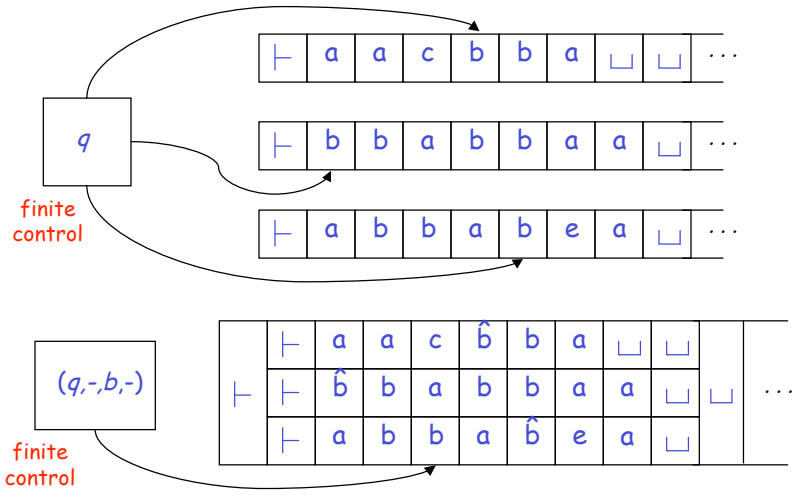
Write "c" into cell on track 1

$\delta(q, b, b, a) = (p, c, d, e, R, L, L)$



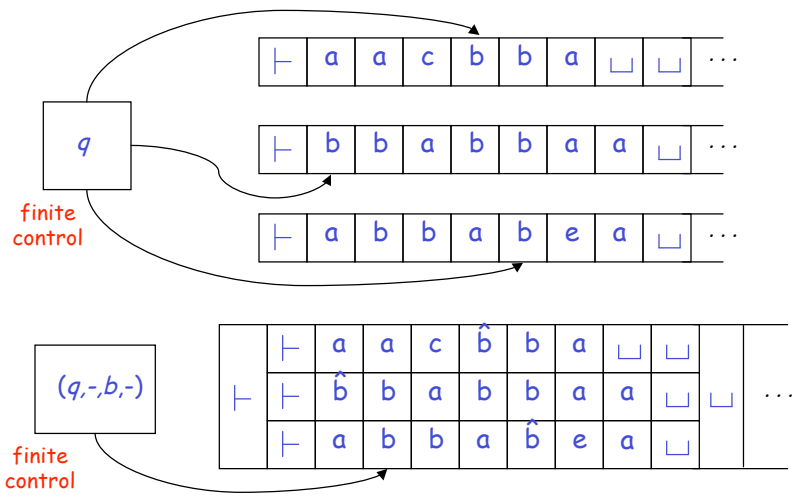
And update head position on track 1

$\delta(q, b, b, a) = (p, c, d, e, R, L, L)$



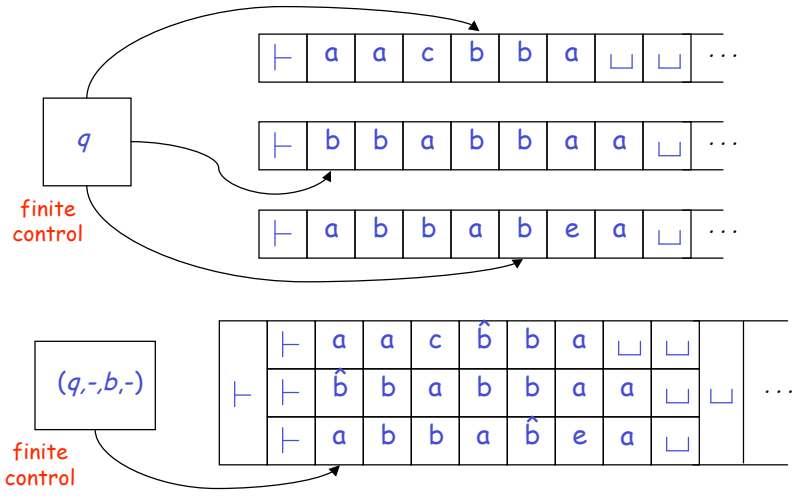
Continue scanning

$\delta(q, b, b, a) = (p, c, d, e, R, L, L)$



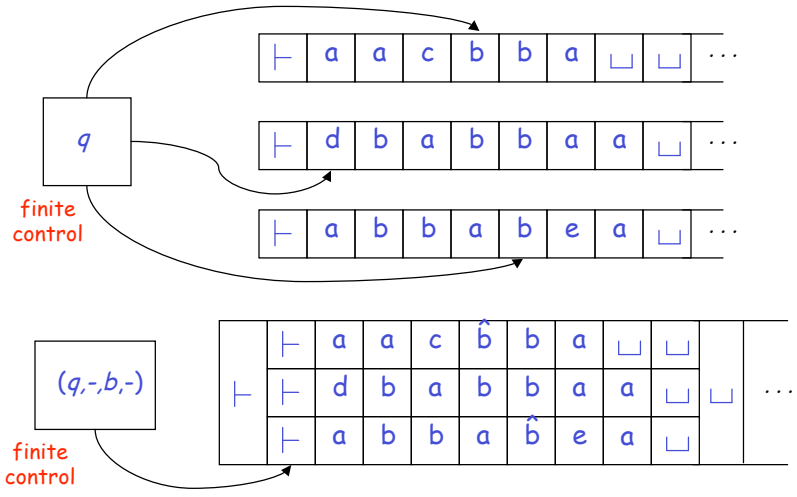
Found position of head on tape 2

$\delta(q, b, b, a) = (p, c, d, e, R, L, L)$



Write "d" into cell on track 2

$\delta(q, b, b, a) = (p, c, d, e, R, L, L)$



Update head position on track 2

$\delta(q, b, b, a) = (p, c, d, e, R, L, L)$