COMPSCI 250: Introduction to Computation

Lecture #37: Two-Way Automata and Turing Machines

David Mix Barrington and Ghazaleh Parvini

12 May 2023

2WDFA's and Turing Machines

- Enhancing a DFA's Abilities
- Definition and Semantics of 2WDFA's
- Why 2WDFA's Have Regular Languages (Sketch)
- Turing Machines
- The Formal Turing Machine Model
- A Turing Machine Example
- The Church-Turing Thesis

Enhancing a DFA's Abilities

- DFA's, and the other models we have now shown to be equivalent to them, model a particular kind of computation. A DFA:
- (1) can read its input only once, from left to right,
- (2) can only read, not write to, the memory holding the input, and
- (3) has only a bounded amount of memory apart from that input.

Enhancing a DFA's Abilities

- In our last week of lectures we will look at another model of computation called a **Turing machine**, which we can think of as an enhanced DFA. Turing machines:
- (1) can move both ways on the **tape** that contains their input,
- (2) can **write** new characters into the space that originally holds the input, and
- (3) can utilize **additional memory**, as much as they need, as well as the original space.

Enhancing a DFA's Abilities

- We'll begin today by looking at the effect of adding new ability (1) alone to a DFA, producing a new kind of machine called a **two-way DFA**.
- In COMPSCI 501 you'll also look at machines that have new abilities (1) and (2) but not (3) -- these are called **linear bounded automata**.

Two-Way Finite Automata

- Like a DFA, a 2WDFA has a state set Q, start state i, final state set F, input alphabet Σ, and transition function δ.
- The only difference is that δ goes from $Q \times \Sigma$ to $Q \times \{L, R\}$. Based on the current state and the letter it sees, the 2WDFA enters a new state and moves *either left or right* on its tape.
- It continues taking steps until or unless it moves off one end of the tape.

Semantics of 2WDFA’s

- We need to define the **semantics** of the 2WDFA M -- the meaning of each computation in terms of defining a language L(M).
- We start with the **read head** on the first letter of the input, and start the computation.
  If the machine moves off the left end of the tape, we say that it **hangs** and the input is not in L(M).

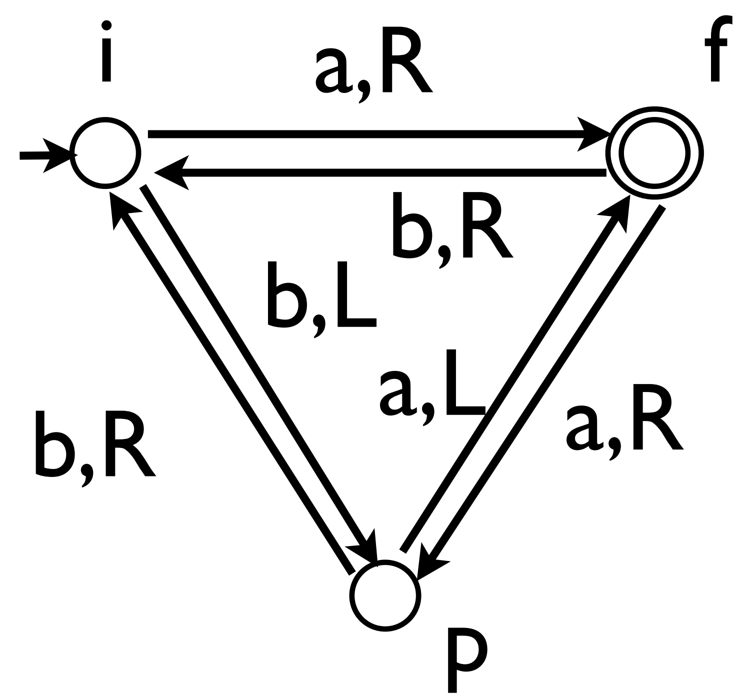
Semantics of 2WDFA's

- If it moves off the right end of the tape, we say that it **accepts** if it goes into a final state and that it **rejects** if it goes into a nonfinal state.
- There is a fourth possibility, that it **loops** or never terminates.
- The input is in L(M) if and only if M accepts.

A 2WDFA Example

- Let's look at the behavior of this 2WDFA on some strings:
- On a, it moves right off the input in state f and accepts.
- On b, it moves off the left end and hangs.
- On aaa, it moves right to state f, right again to state p, left to state f, right to p,..., and thus loops forever.

iRf

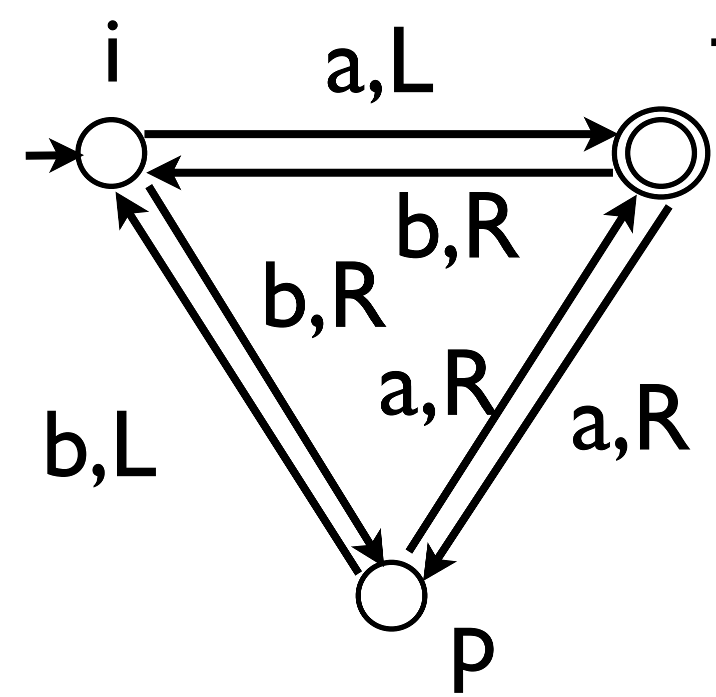


iLp hang

iRfRpLfRpLf…

Clicker Question #1

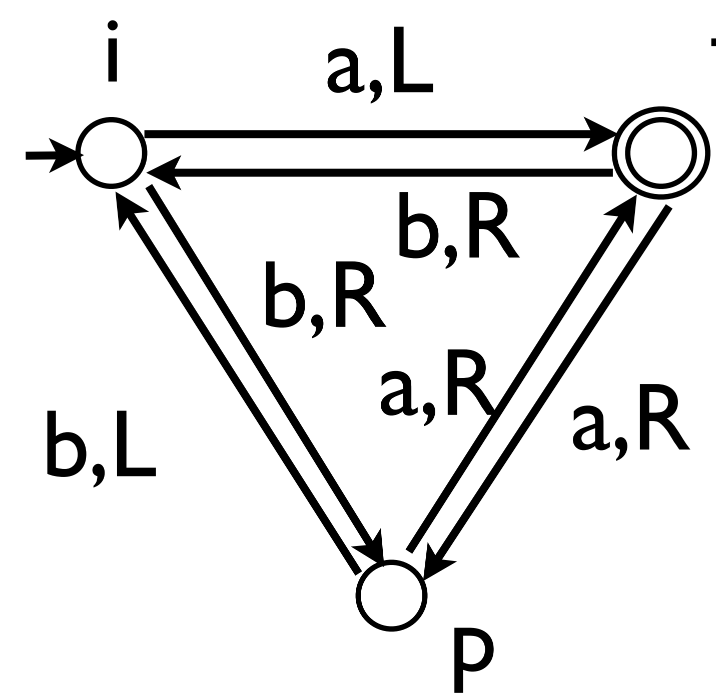
- What does this 2WDFA do on input string babab?
- (a) accepts by leaving to the right in a final state
- (b) enters an infinite loop
- (c) hangs by leaving to the left
- (d) rejects by leaving to the right in a nonfinal state



Not the Answer

Clicker Answer #1

- What does this 2WDFA do on input string babab?
- (a) accepts by leaving to the right in a final state
- *(b) enters an infinite loop*
- (c) hangs by leaving to the left
- (d) rejects by leaving to the right in a nonfinal state



iRpRfRiLfRiLfR...

$b_1a_2b_3a_4b_3a_4b_3\ldots$

2WDFA's and Regular Languages

- Could a 2WDFA have a non-regular language like $\{a^n b^n : n \geq 0\}$?
  For DFA's, we argued that after the a's have been read, the machine "must know" how many a's it saw (formally, each different number of a's was in a different equivalence class).
- But now, the machine could make multiple visits to the a's. Can it use this capability in any way to get more information about the a's?

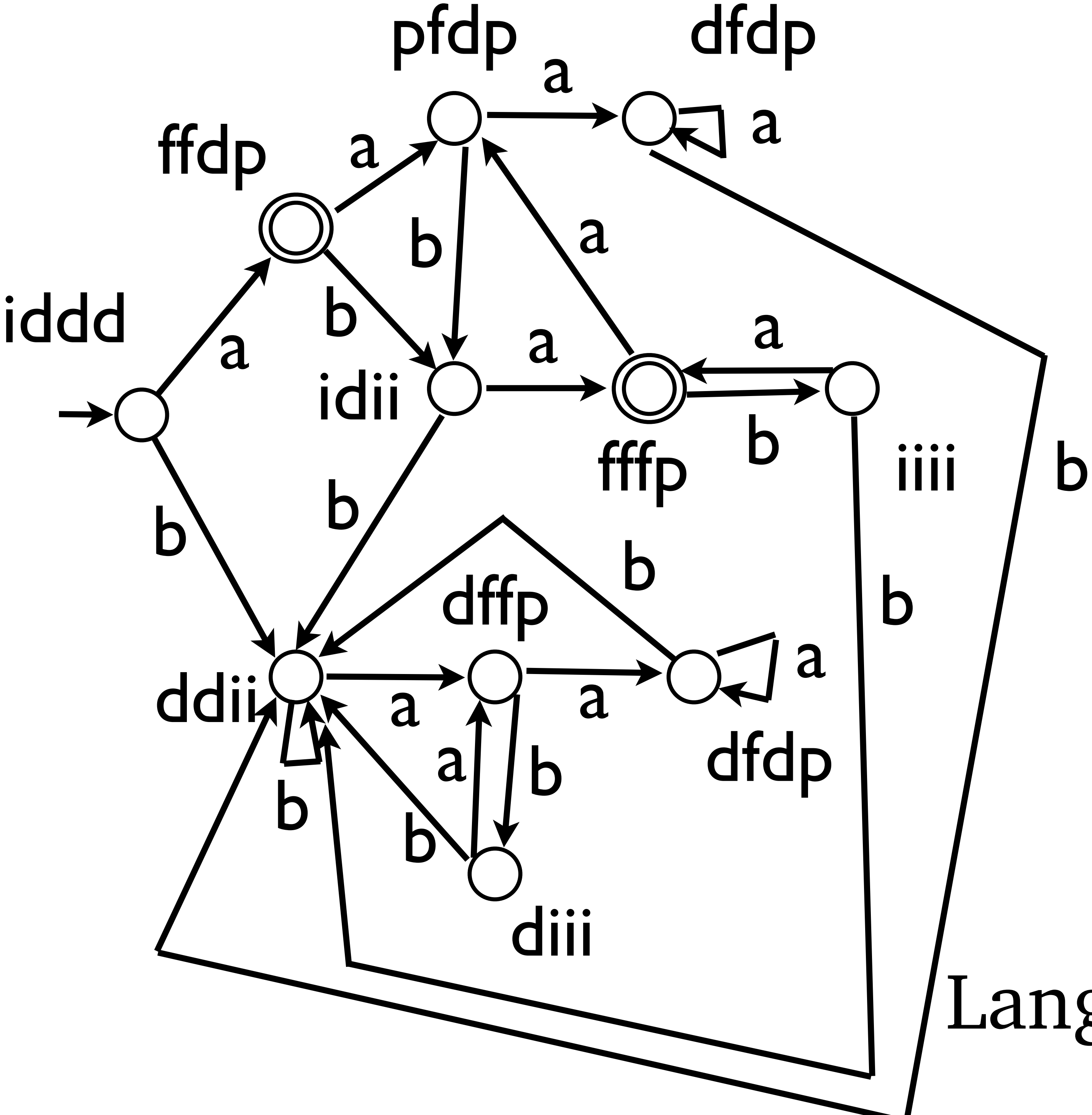
2WDFA's and Regular Languages

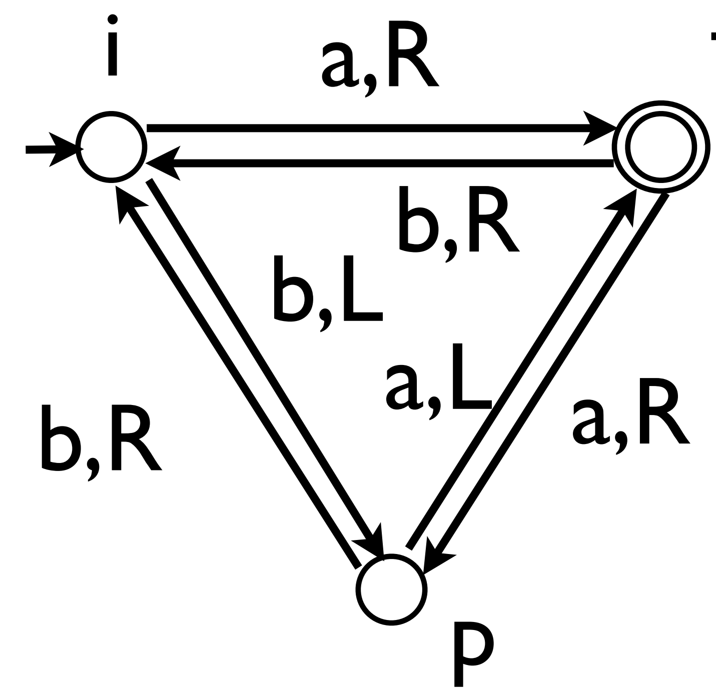
- In Section 15.1 of the text, we prove that the language of any 2WDFA is regular. Here is a sketch of the argument.
- Given a 2WDFA M and a string w, we define several functions of w based on M's behavior.
- If M exits w to the right in state q when started in state i on the left, we say that $f_0(w) = q$.
- If it hangs or loops in that situation, we say that $f_0(w) = d$.

2WDFA's and Regular Languages

- Similarly, we define a function f_p for each state p. Consider starting M on the *right* of w in state p.
- If it loops or hangs, we define $f_p(w) = d$.
- If it exits to the right in state q, we define $f_p(w) = q$.

Converting a 2WDFA Example





Language of DFA = a(ba+aba)*

2WDFA's = Regular

- Here's the crux of the argument. Suppose that for two strings v and w, the values of each of these functions are the same. That is, $f_0(v) = f_0(w)$ and for each state p, $f_p(v) = f_p(w)$.
- Then, we will argue, v and w are L(M)-equivalent in the sense of the Myhill-Nerode Theorem.
- Since there are only finitely many possible sequences of values for these functions, there are only finitely many equivalence classes, and the theorem tells us that L(M) is a regular language.

2WDFA's = Regular

- We need to show that for any string z, the strings vz and wz are either both in L(M) or both not in L(M).
- Let z be an arbitrary string, assume that the functions agree on v and w, and look at what happens when M starts computing on v and on w.
- If M hangs or loops on vz without leaving v, it must do the same on wz because $f_0(v) = f_0(w) = d$.

2WDFA's = Regular

- If it exits v to the right, then it also exits w to the right, and in the same state. From that point, the two computations in z proceed identically, until or unless they leave z.
- If they leave to the right, both computations accept or both reject. If they go back into v and w, they do so in the same state p. Then either both die, or both move back into z in the same state $f_p(v) = f_p(w)$, and so forth until eventually both accept, both reject, or both die. So $vz \in L(M) \leftrightarrow wz \in L(M)$.

Turing Machines

- In the 1930's, various researchers designed **systems of computation** in an attempt to create a simple mathematically precise model that could express any possible computation. The model that has become most widely used is the **Turing machine**, proposed by the English mathematician Alan Turing in 1936. (Another one of these models, the **lambda calculus** of Alonzo Church, developed into the Lisp family of programming languages.)

Turing Machines

- Turing and Church each convinced themselves that any clear, precise computational instructions could be translated (we might say “compiled”) into each of their systems.
- When each heard about the other’s system, they proved that any computation in one could be translated to the other.

Turing Machines

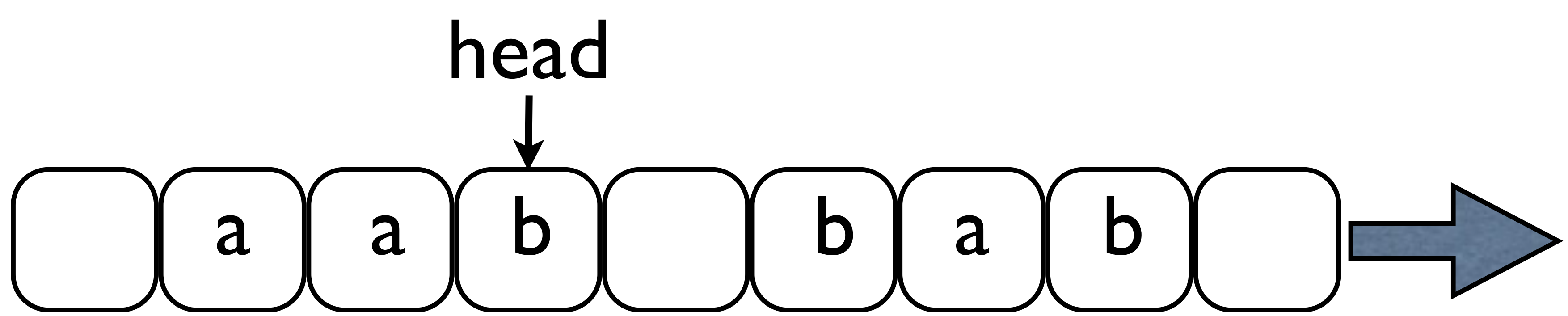
- Thus the two systems defined the same set of **computable functions** from strings to strings.
- Just as a language is either regular or not, a function is either computable or not.
- (Actually finite-state machines would not be formalized for another twenty years or so.)

The Turing Machine Model

- A **Turing machine** is formally defined by giving a state set Q, an input alphabet Σ, a start state i, and a final state set F, as we've seen already.
- But it also has a **tape alphabet** Γ with $\Sigma \subseteq \Gamma$, and a **blank symbol** □ that is an element of Γ and is the initial contents of every tape cell right of the input.

The Turing Machine Model

- The machine has a **tape** that is infinite to the right and finite to the left. Each cell of the tape holds a letter in Γ at any given time.
- There is a **head** that points to one cell of the tape at any given time.



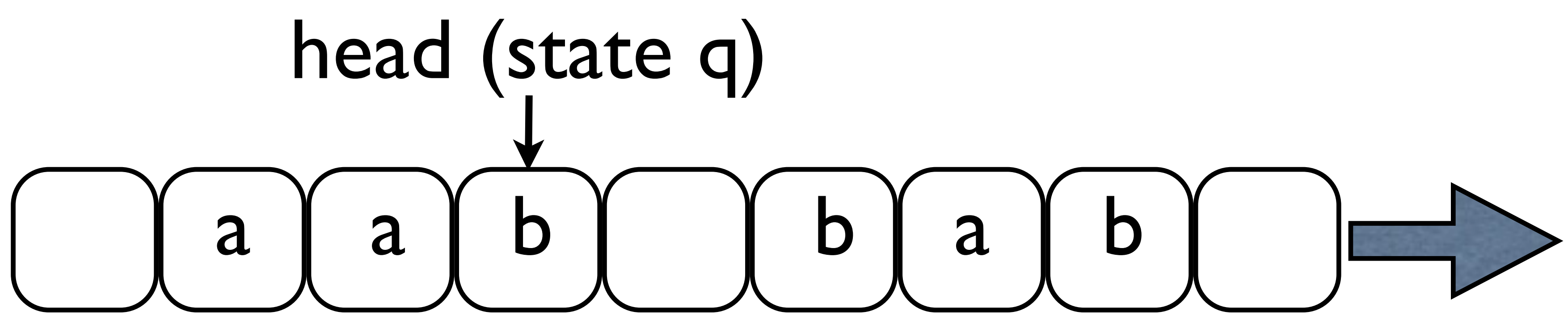
The Turing Machine Model

- The **transition function** δ is from $Q \times \Gamma$ to $Q \times \Gamma \times \{L, R\}$.
  A **step** of the computation consists of the machine looking at the letter at its head, applying δ to its current state and that letter to get a triple (q, a, L) or (q, a, R), then *changing its state* to q, *writing* an a in the current cell, and *moving* left or right.
- Actually δ is not defined for states in F -- the machine **halts** in those states.

Turing Machine Configurations

- At any given time, we can describe everything we would ever want to know about the Turing machine's computation by a string called a **configuration**.

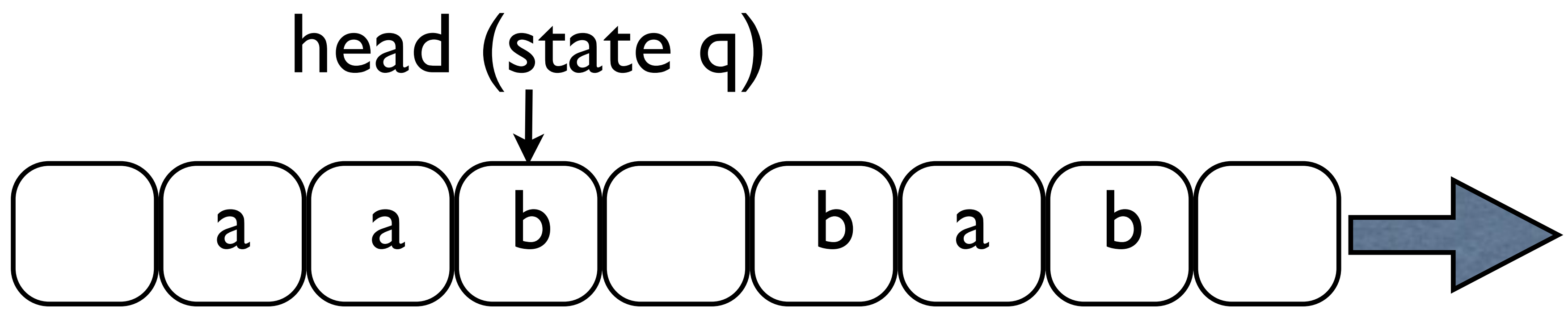
- What we need to record is the current state, the contents of the tape, and the position of the head.



Configuration: □aaqb□bab□

Turing Machine Configurations

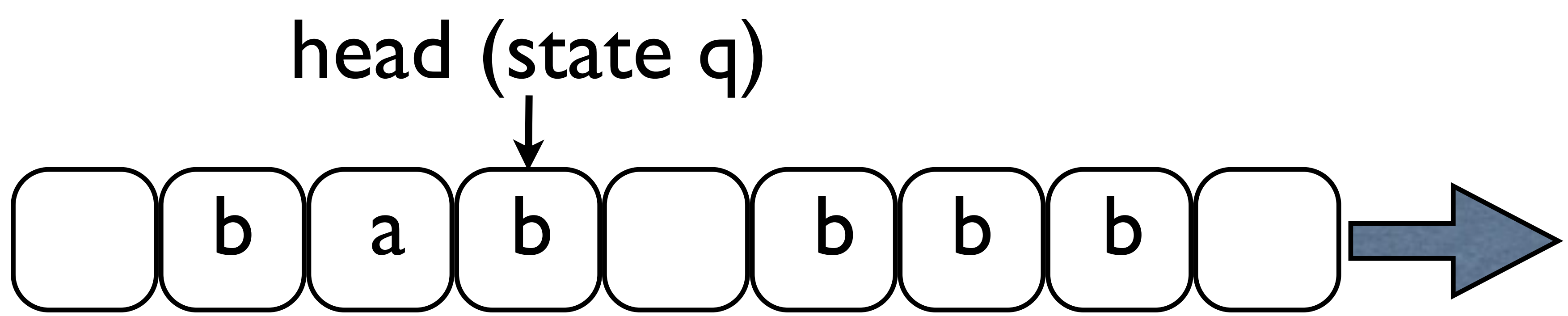
- We record the tape contents as a string of letters from Γ, starting at the left end of the tape and ending with the last non-blank letter.
- We record the state and head position by inserting a letter for the state into this string, just to the left of the head position.



Configuration: □aaqb□bab□

Clicker Question #2

- Suppose δ(q, b) = (r, □, L). What will be the new configuration of the Turing machine below?
- (a) □baa□bbb□ (b) □bar□□bbb□
- (c) □bra□□bbb□ (d) □babr□bbb□

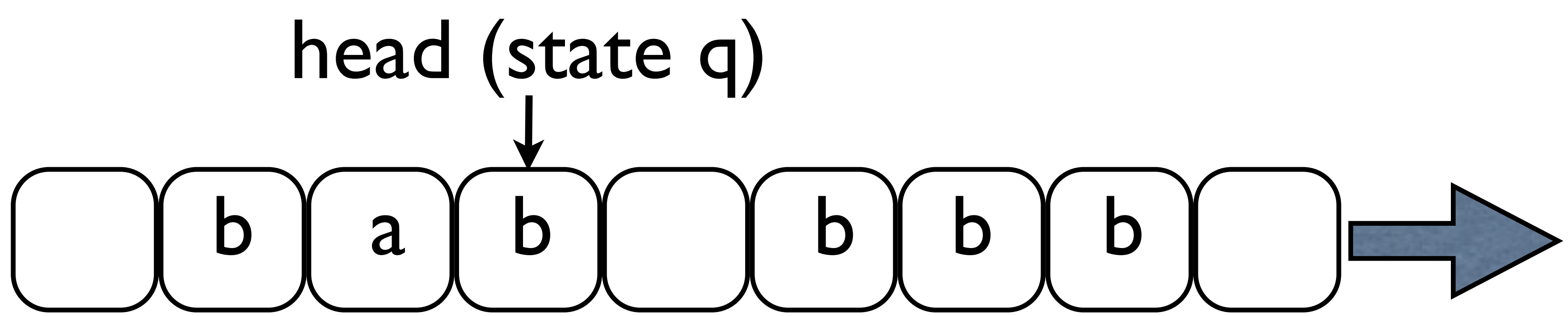


Configuration: □baqb□bbb□

Not the Answer

Clicker Answer #2

- Suppose δ(q, b) = (r, □, L). What will be the new configuration of the Turing machine below?
- (a) □baa□bbb□ (b) □bar□□bbb□
- *(c) □bra□□bbb□* (d) □babr□bbb□



Configuration: □baqb□bbb□

Turing Machine Configurations

- A Turing machine starts with only finitely many non-blank symbols on its tape.
- So in writing a configuration, we only need to go to the last non-blank symbol (unless we need to go further to indicate the head position).
- We can think of the computation then as a series of configurations, starting with i□$w_1w_2...w_n$ and continuing until or unless the machine halts or hangs.

A Turing Machine Example

- On the next slide is a machine that solves a problem that a DFA cannot. When started in configuration i□$w_1w_2...w_n$, it will halt if and only if w is in the language $\{a^nb^n: n \geq 0\}$ -- otherwise it will hang.
- With input aabb we get i□aabb, □paabb, □□qabb, □□aqbb, □□abqb, □□abbq□, □□abrb, □□asb, □□sab, □s□ab, □□pab, □□□qb, □□□bq□, □□□rb, □□s□, □□□p□, □□□h□. The string aabb is accepted.

A Turing Machine Example

```
In i: Move R and go to p.
In p: On □, go to h.
      On b, move L and go to z.
      On a, print □, move R, and go to q.
In q: On a or b, move R and stay in q.
      On □, move L and go to r.
In r: On a or □, move L and go to z.
      On b, print □, move L, and go to s.
In s: On a or b, move L and stay in s.
      On □, move R and go to p.
In h: Halt (final state).
In z: Move left and stay in z.
```

Clicker Question #3

- What does this TM do on inputs aa (starting i□aa□) and bb (starting i□ab□)?
- (a) accepts both (b) accepts aa, hangs on ab
- (c) hangs on both (d) hangs on aa, accepts ab

```
In i: Move R and go to p.
In p: On □, go to h.
      On b, move L and go to z.
      On a, print □, move R, and go to q.
In q: On a or b, move R and stay in q.
      On □, move L and go to r.
In r: On a or □, move L and go to z.
      On b, print □, move L, and go to s.
In s: On a or b, move L and stay in s.
      On □, move R and go to p.
In h: Halt (final state).
In z: Move left and stay in z.
```

Not the Answer

Clicker Answer #3

- What does this TM do on inputs aa (starting i□aa□) and bb (starting i□ab□)?
- (a) accepts both (b) accepts aa, hangs on ab
- (c) hangs on both *(d) hangs on aa, accepts ab*

```
In i: Move R and go to p.
In p: On □, go to h.
      On b, move L and go to z.
      On a, print □, move R, and go to q.
In q: On a or b, move R and stay in q.
      On □, move L and go to r.
In r: On a or □, move L and go to z.
      On b, print □, move L, and go to s.
In s: On a or b, move L and stay in s.
      On □, move R and go to p.
In h: Halt (final state).
In z: Move left and stay in z.
```

i□aa□	i□ab□
□paa□	□pab□
□□qa□	□□qb□
□□aq	□□bq□
□□ra□	□□rb□
□z□a□	□s□
z□□a□	□□p□
hang	□□h□□
	accept

The Church-Turing Thesis

- The **Church-Turing Thesis** says that any "reasonable" general-purpose model of computation will be able to compute exactly the same functions from strings to strings as Turing machines or the lambda calculus.
- (More precisely, they compute the same set of **partial functions**, because a general computation always has the possibility of not returning an output.)

The Church-Turing Thesis

- We can’t mathematically prove this thesis, only amass evidence for it. In fact it actually serves as an implicit definition of “reasonable”.
- Serious people have argued against the thesis -- for example physicist Roger Penrose argues that quantum effects in the brain compute in ways that a Turing machine could not. (Dave doesn’t buy it.)
- For more on this see Turing’s article *On Minds and Machines* or almost anything by Douglas Hofstadter.

The Church-Turing Thesis

- You probably believe that we could simulate a Turing machine in Java, given unlimited memory. Could a Turing machine simulate any Java program?
- We know that Java can be compiled into machine language, so we would have to believe that any machine language program could be simulated by a TM.