Turing Machines

Chapter 17

Alan Turing

- One of the 100 Most Important People of the 20th Century
  - For his role in the creation of the modern computer
  - "The fact remains that everyone who taps at a keyboard, opening a spreadsheet or a word-processing program, is working on an incarnation of a Turing machine."
- Turing machine, influential formalization of the concept of the algorithm and computation
- Turing test, influential in AI

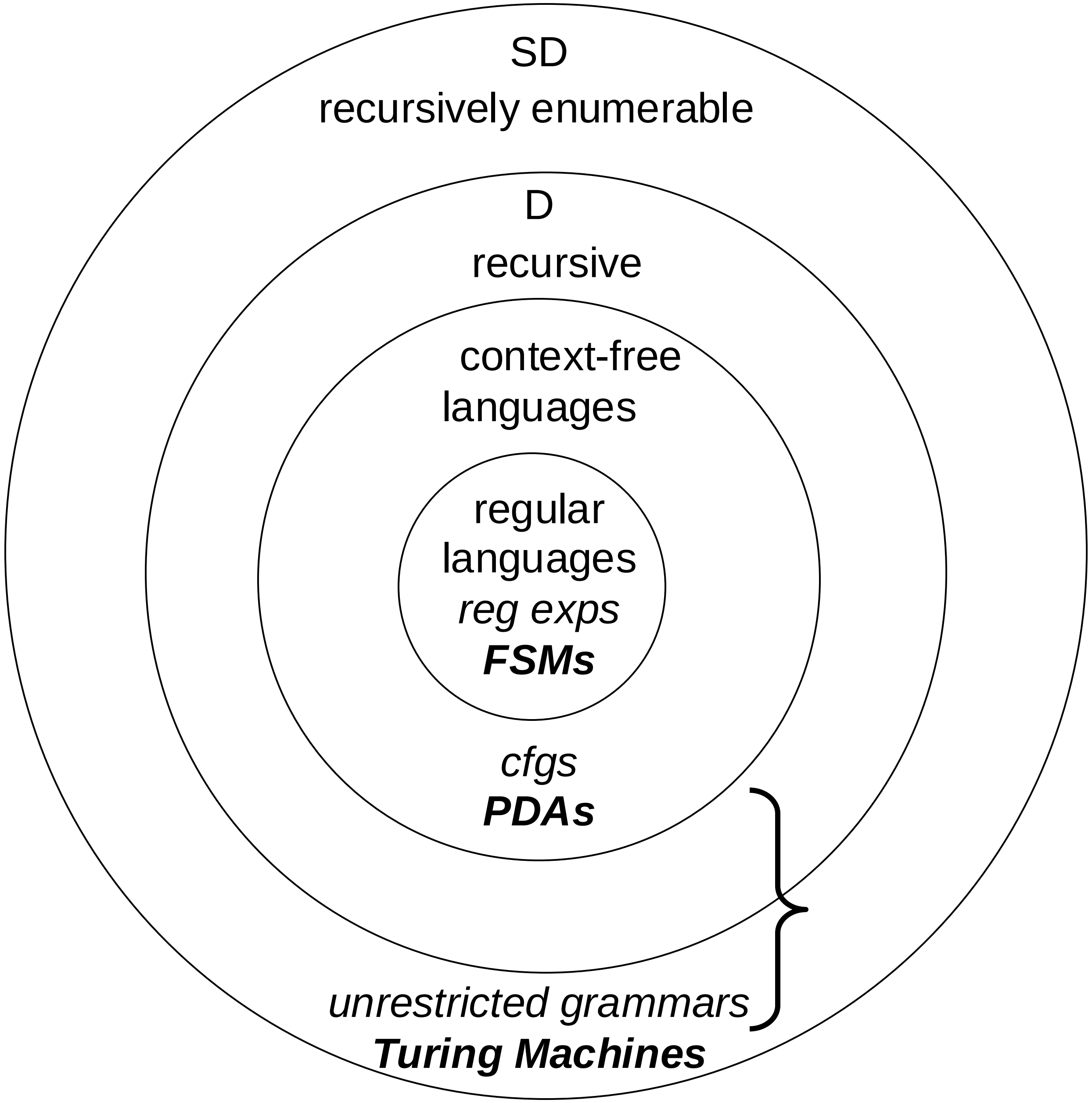
1912 – 1954

- 1936 – 1938, PhD, Princeton, under Alonzo Church
- Then, back to Cambridge, attended lectures by Ludwig Wittgenstein about the foundations of mathematics.
  - *Ludwig Wittgenstein*, student of *Bertrand Russell* at Cambridge, the two are widely referred to as the greatest philosophers of last century

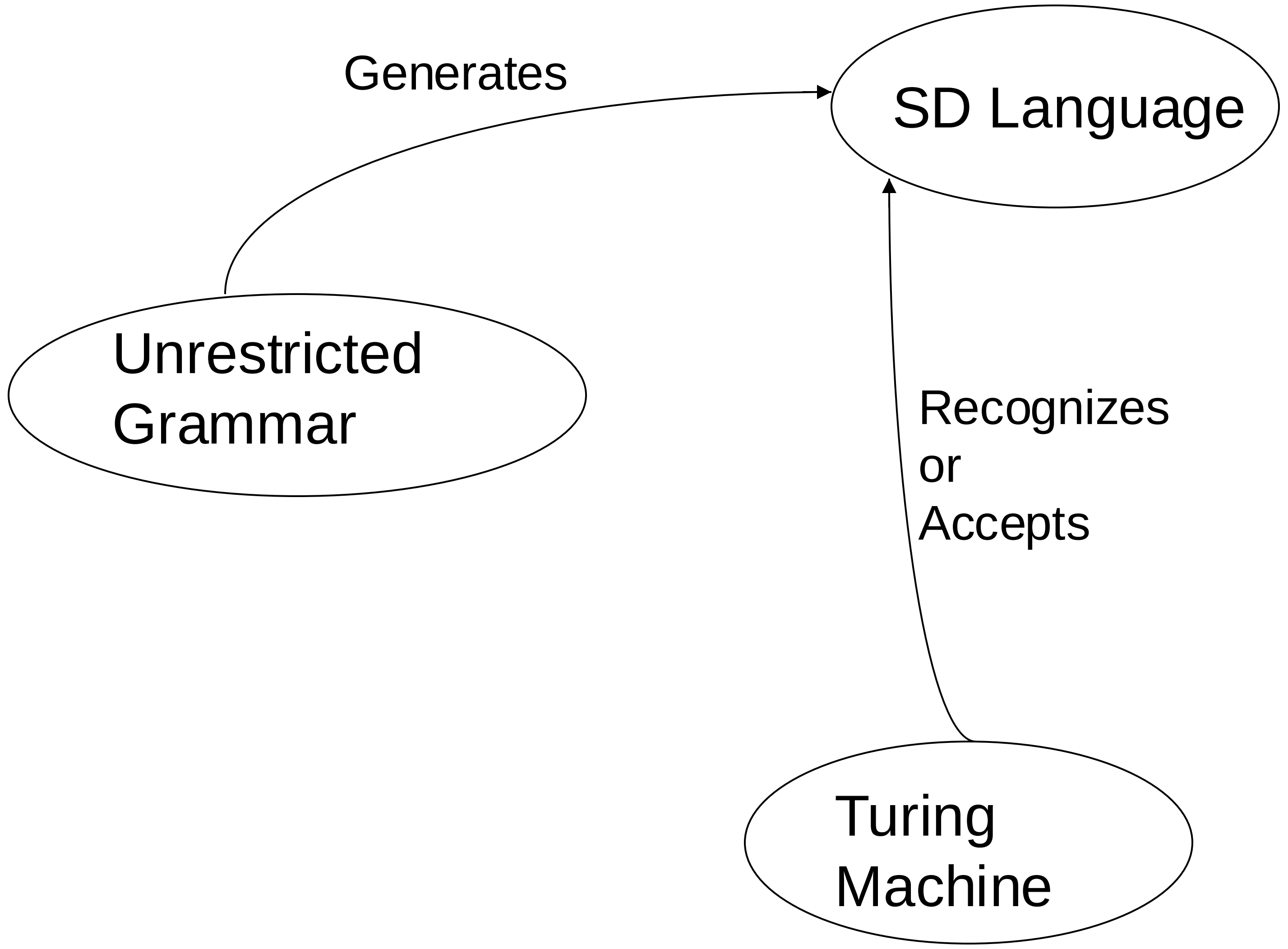
Alan Turing Memorial

Sackville Gardens in Manchester, England

Languages and Machines



Grammars, SD Languages, and Turing Machines



Turing Machines

Can we come up with a new kind of automaton that has two properties:

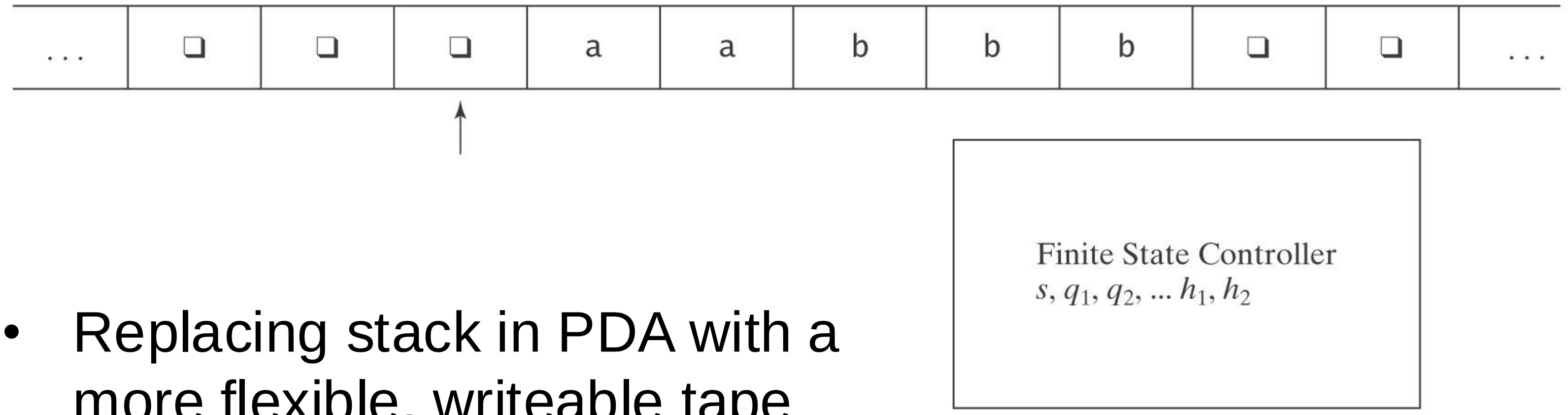
- powerful enough to describe all computable things

  unlike FSMs and PDAs

- **simple enough that we can reason formally about it**

  like FSMs and PDAs
  unlike real computers

Turing Machines



- Replacing stack in PDA with a more flexible, writeable tape
- Single read/write head
- Input on the tape, no longer “consumed” as in FSM/PDA
- By convention, start state to the left of the leftmost symbol

At each step, the machine must:

- choose its next state,
- write on the current square, and
- move left or right

A Formal Definition

A Turing machine *M* is a sixtuple (K, Σ, Γ, δ, s, A):

- *K* is a finite set of states;
- Σ is the input alphabet, which does not contain ❑;
- Γ is the tape alphabet, which must contain ❑ and have Σ as a subset.
- $s \in K$ is the initial state;
- $A \subseteq K$ is the set of accepting states;
- δ is the transition function:

K	$\times$	Γ	to	K	$\times$	Γ	$\times$	$\{\rightarrow, \leftarrow\}$
state	$\times$	tape char		state	$\times$	tape char	$\times$	action (R or L)

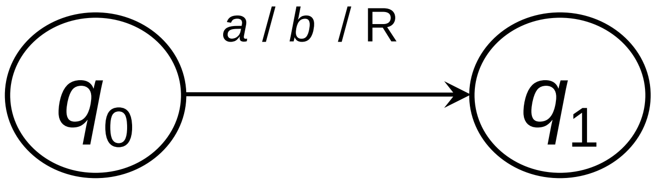
❑ is the tape symbol (blank symbol):

- the input string does not contain ❑
- initially, all tape squares except those containing the input contain ❑
- helps in recognizing ends of input
- some definitions do not use it

Transition Function

K	$\times$	Γ	to	K	$\times$	Γ	$\times$ $\{\rightarrow, \leftarrow\}$
state	$\times$	tape char		state	$\times$	tape char	$\times$ action (R or L)

- If δ contains $((q_0, a), (q_1, b, R))$
  - when M is in state q_0, char under read/write head is a
  - M will go to q_1, write b, and move to right))



Notes on the Definition

1. The input tape is infinite in both directions.

2. δ is a function, not a relation. So this is a definition for deterministic Turing machines.

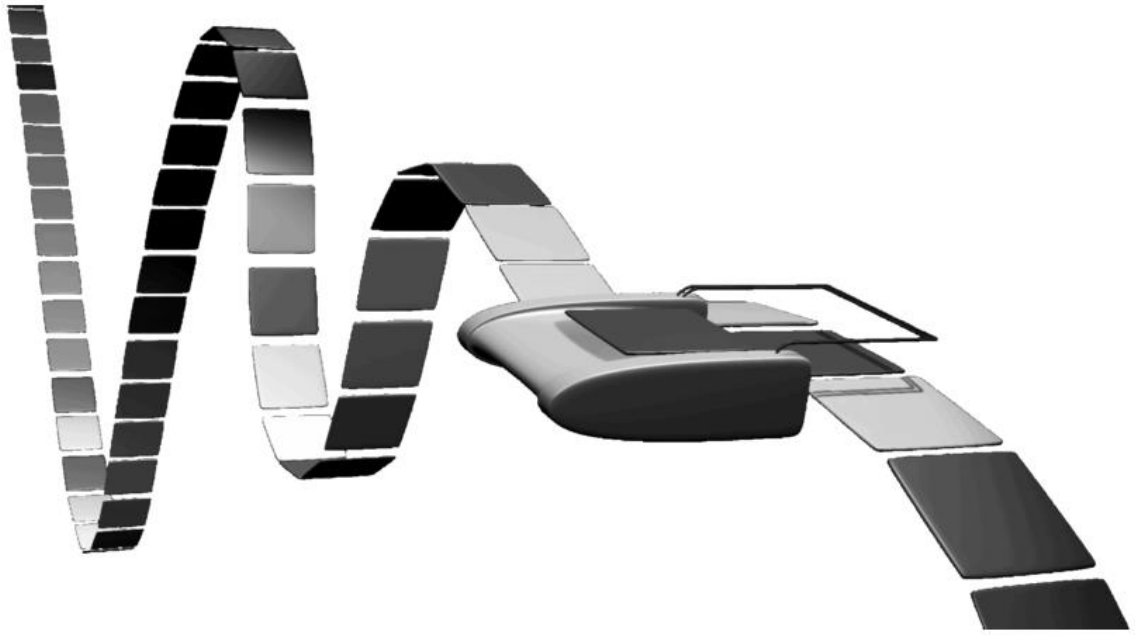
3. Turing machines do not necessarily halt (unlike DFSMs and DPDAs). Why?

4. The above is the same to say, the path (since there's only one) may or may not end.

5. Turing machines generate output so they can compute functions.

More on Turing Machines

Artistic representation:



On sale

http://www.youtube.com/watch?v=cYw2ewoO6c4

Simulator: many online

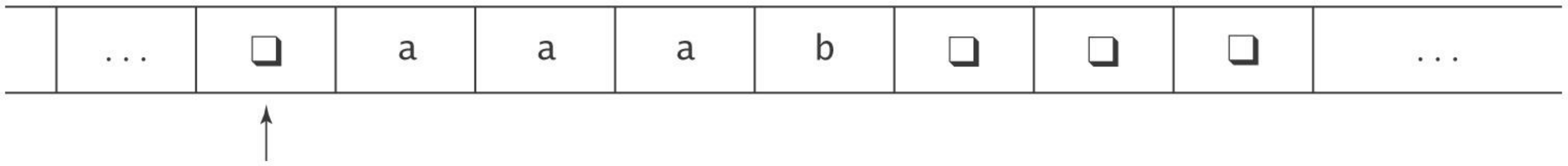
Add b's to Make them match a's

M takes as input a string in the language:

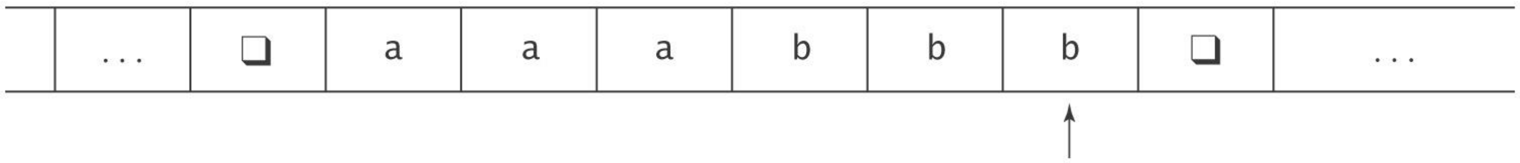
$\{\mathtt{a}^i\mathtt{b}^j, 0 \le j \le i\}$,

and adds `b`'s to make the number of `b`'s equal the number of `a`'s.

The input to *M* will look like this:



The output should be:



Describing *M* in English

1. Move one square to the right. If the character under the read/write head is ❑, halt. Otherwise, continue.
2. Loop:
   - 2.1. Mark off an a with a $.
   - 2.2. Scan rightward to the first b or ❑.
     - If b, mark it off with a # and get ready to go back and find the next matching a,b pair.
     - If ❑, then there are no more b's but there are still a's that need matches. So it is necessary to write another b on the tape. But that b must be marked so that it cannot match another a. So write a #. Then get ready to go back and look for remaining unmarked a's.
   - 2.3. Scan back leftward looking for a or ❑. If a, then go back to the top of the loop and repeat. If ❑, then all a's have been handled. Exit the loop. (Notice that the specification for *M* guarantees that there will not be more b's than a's).
3. Make one last pass all the way through the nonblank area of the tape, from left to right, changing each $ to an a and each # to a b.
4. Halt.

Defining *M* with Transition Table

K = {1, 2, 3, 4, 5, 6}, Σ = {a, b}, Γ = {a, b, ❑, $, #},
s = 1, A = {6}, δ =

(((1, ❑), (2, ❑, ->)),
… …
((2, ❑), (6, ❑, ->)),
((2, a), (3, $, ->)),
… …

((3, #), (4, $, <-)),
… …

((4, ❑), (5, ❑, ->)),
… …

((5, ❑), (6, ❑, <-)),
((5, $), (5, a, ->)),
((5, #), (5, b, ->)))

The table can also look like this:

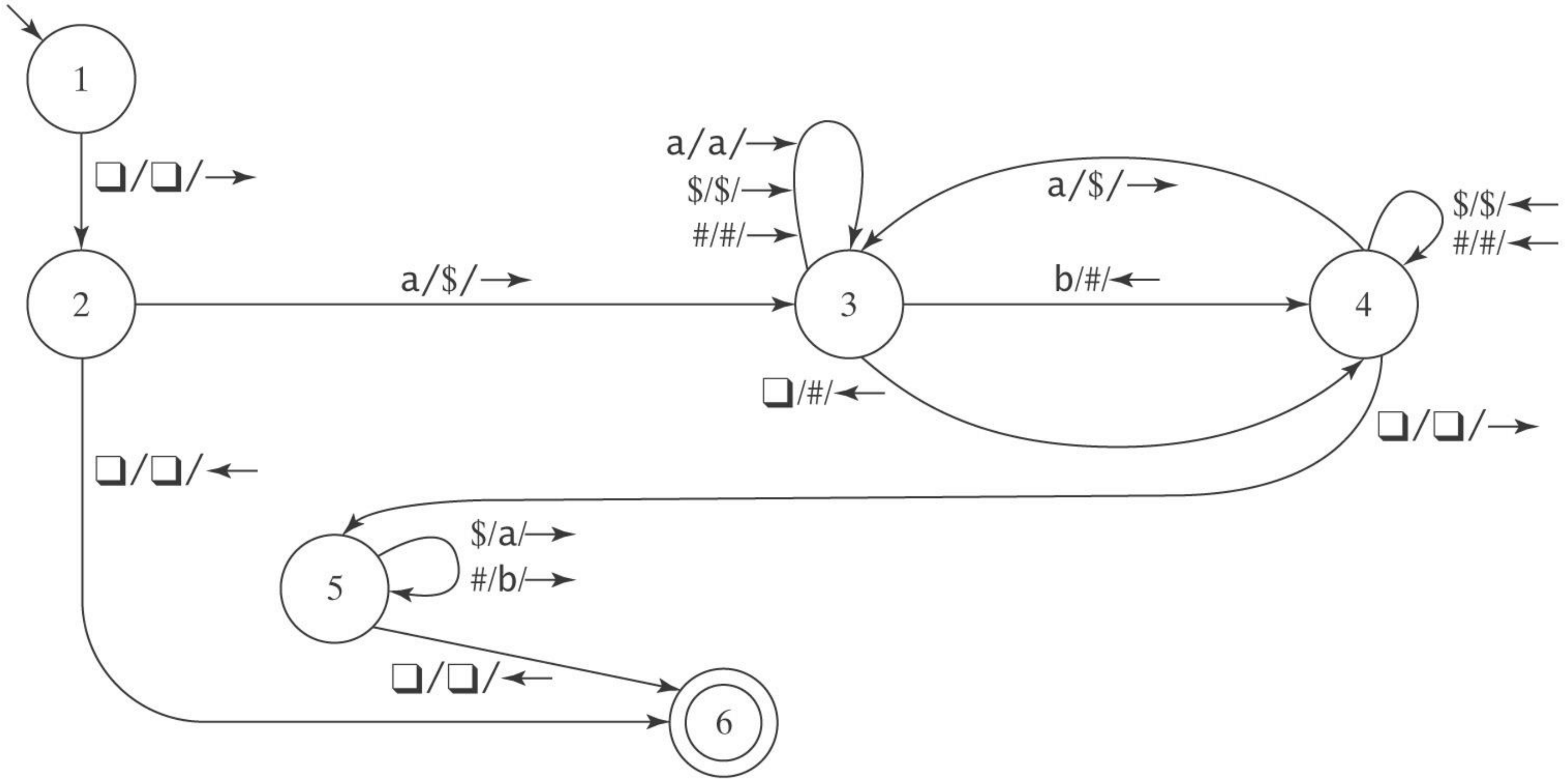
	symbol				
state	**❑**	**a**	**b**	**$**	**#**
1	(2, ❑, ->)				
2					
3					
4					
5					

Note: 6 is a halting state, no transition out of it

transition table is hard to read, can use graphical representation.

Defining *M* with Transition Diagram

K = {1, 2, 3, 4, 5, 6}, Σ = {a, b}, Γ = {a, b, ❑, \$, #},
s = 1, A = {6}, δ =



Notations may vary from book to book …

Notes on Programming

- The machine has a strong procedural feel, with one phase coming after another.
- There are common idioms, like scan left until you find a ❑
- There are two common ways to scan back and forth marking things off.
  - Scan left to the first `a` and process the rest of `a`'s right to left, as in last example
  - Scan all the way left until we find the first unmarked `a`, process all `a`'s left to right
- If we care about output (function, not decision problem)
  - often there is a final phase that makes one last pass over the tape and converts the marked characters back to proper form.
- Even a very simple machine is a nuisance to write.

Halting

- A DFSM or DPDA, on input w, is guaranteed to halt in at most |w| steps.

- A NDFSM or NDPDA without ε-transitions, on input w, is guaranteed to halt in at most |w| steps.

- A NDFSM *M*, on input *w*, is not guaranteed to halt. But there is a DFSM M' such that L(M) = L(M'). (in this case, M and M' are said to be equivalent)

- A NDPDA *M*, on input *w*, is not guaranteed to halt. But there is an equivalent NDPDA M' that halts.

- A deterministic TM *M*, on input *w*, is not guaranteed to halt. And, there may not be an equivalent one that halts.
  - What if L(M) is in D?

Formalizing the Operation

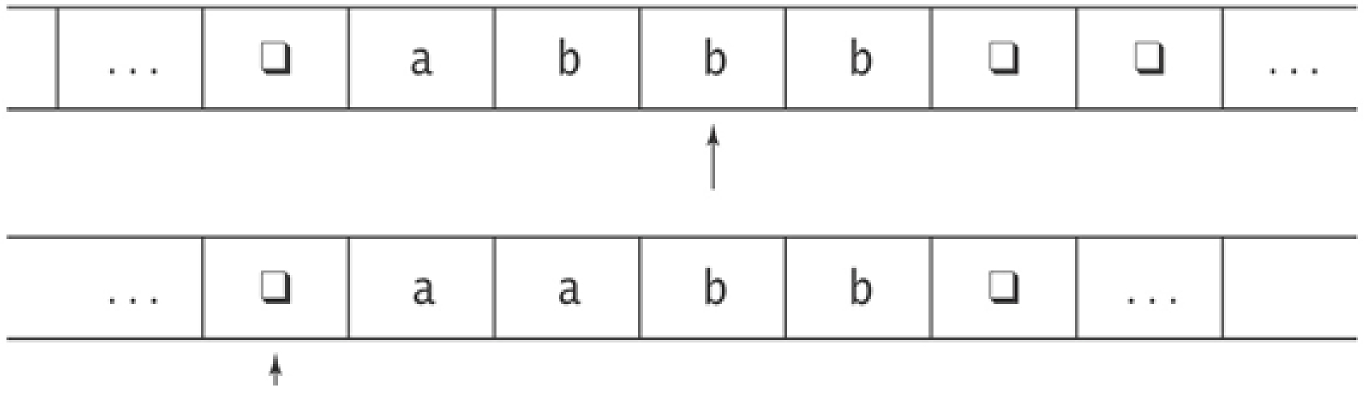
As in FSM and PDA, we use configuration to describe a particular execution/processing step of the system.

A *configuration* of a Turing machine

$M = (K, \Sigma, \Gamma, s, A)$ is an element of:

K	$\times$	$((\Gamma - \{\square\})\, \Gamma^*) \cup \{\varepsilon\}$	$\times$	Γ	$\times$	$(\Gamma^* (\Gamma - \{\square\})) \cup \{\varepsilon\}$
state		Active tape to the left of read/write head		Square under read/write head		Active tape to the right of read/write head

Example Configurations



	as a 4-tuple		Shorthand
(1)	(q, ab, b, b)	=	(q, ab$\underline{\text{b}}$b)
(2)	(q, ε, ❑, aabb)	=	(q, $\underline{❑}$aabb)

Initial configuration is (*s*, $\underline{❑}$*w*).

Yields

(q_1, w_1) |- (q_2, w_2) iff (q_2, w_2) is derivable, via δ, in one step.

After the transition, state q_1 -> q_2 , active tape w_1 -> w_{12}

For any TM M, let **|-*** be the reflexive, transitive closure of $|\!-_M$.
Configuration C_1 ***yields*** configuration C_2 if: C_1 **|-**$_M$***** C_2.

Recall: a path is a maximal sequence of execution / processing steps (described by configurations) from the start

Even for a deterministic TM, a path may end, or may not

Turing Machines as Language Recognizers

The initial configuration of *M*: $(s, \underline{\square}w)$

Let $M = (K, \Sigma, \Gamma, \delta, s, \{y\})$.

- *M* ***accepts*** a string *w* iff the path accepts it.
    - For TM, ***(y, w′)*** is an accepting configuration
    - We do not care the tape content when it halts

- *M* ***rejects*** a string *w* iff the path rejects it.

- Possible that on input *w*, *M* neither accepts nor rejects. It may loop.

- Again, this definition is a bit different from our text and Sipser, but similar to Ullman. This definition is more consistent to FSM and DPDA, i.e., all the non-accepting states are rejecting states.
- This way, we do not need to draw the rejecting states and the many transitions to them.
- It works just fine with JFLAP.

Turing Machines as Language Recognizers

M ***decides*** a language $L \subseteq \Sigma^*$ iff:

For any string $w \in \Sigma^*$ it is true that:

if $w \in L$ then *M* accepts *w*, and

if $w \notin L$ then *M* rejects *w*.

A language *L* is ***decidable*** iff there is a Turing machine *M* that decides it. In this case, we will say that *L* is in ***D***.

In some books, *D* is called the set of recursive languages.

- Computability theory was recursion theory, originated with work of Kurt Gödel, Alonzo Church, Alan Turing, Stephen Kleene …

A Deciding Example: $A^nB^nC^n$

$A^nB^nC^n = \{a^n b^n c^n : n \geq 0\}$

Example : $\underline{❑}$aaabbbccc ❑❑❑❑❑❑❑❑❑

Example : $\underline{❑}$aaccb ❑❑❑❑❑❑❑❑❑

- Not context-free, not recognizable by a PDA

```
1. Move right.
2. If the current symbol is the blank symbol, halt and accept.
3. Loop:
  3.1. If the current symbol is a, change it to X and move right. Otherwise exit loop.
  3.2. Scan rightwards, pass a's and Y's to find b.
  3.3. If b found, change it to Y and move right. Otherwise reject.
  3.4. Scan rightwards, pass b's and Z's to find c.
  3.5. If c found, change it to Z and move left. Otherwise reject.
  3.6. Scan leftwards, pass a's, b's, Y's and Z's to find X.
  3.7. When finding the first X, move right, and go back to 3.1.
4. If the current symbol is Y, move right. Otherwise reject.
5. Scan rightwards, pass Y's and Z's, to find the blank symbol.
6. When finding the first blank symbol, move right and accept.
```

Another Deciding Example: wcw

WcW = $\{w\mathtt{c}w : w \in \{\mathtt{a}, \mathtt{b}\}^*\}$

Example: ❑abbcabb❑❑❑

Example: ❑acabb❑❑❑

Describing *M* in English:

1. Loop:

   1.1. Move right to the first character. If it is c, exit the loop. Otherwise, overwrite it with # and **remember** what it is.

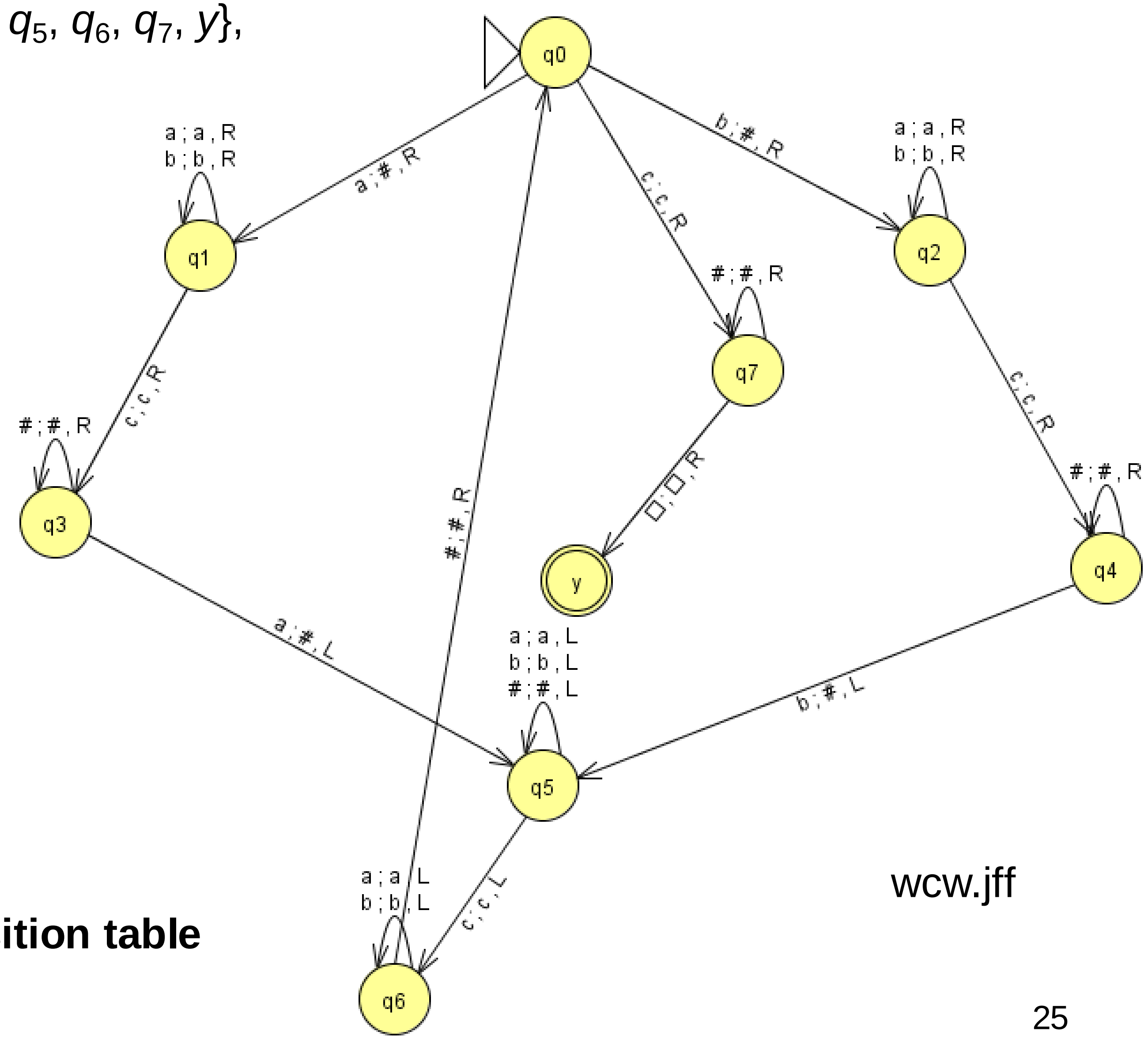
   1.2. Move right to find c. Then continue right to the first unmarked character. If it is ❑, halt and reject. (This will happen if the string to the right of c is shorter than the string to the left.) If it is anything else, check to see whether it matches the remembered character from the previous step. If it does not, halt and reject. If it does, mark it off with #.

   1.3. Move leftward to find the next unprocessed character (pass c to find first # then right).

3. There are no characters remaining before the c. Make one last sweep left to right checking that there are no unmarked characters after the c and before the first ❑. If there are, halt and reject. Otherwise, halt and accept.

wcw: define *M* with transition diagram

$K = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, y\}$,
$\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$,
$\Gamma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \#, ❑\}$,
$s = q_0$,
$A = \{y\}$,
$\delta =$



Exercise:
define *M* with transition table

wcw.jff

Semideciding a Language

Let Σ be the input alphabet to a TM *M*. Let $L \subseteq \Sigma^*$.

M ***semidecides*** *L* iff, for any string $w \in \Sigma_M^*$:

- $w \in L \rightarrow M$ accepts *w*
- $w \notin L \rightarrow M$ does not accept *w*.

M may either:
reject or
fail to halt.

A language *L* is ***semidecidable*** iff there is a Turing machine that semidecides it. We define the set ***SD*** to be the set of all semidecidable languages.

In some books, *SD* is called the set of recursively enumerable languages.

Example of Semideciding

Let L = `b`*`a`(`a` ∪ `b`)*

We can build M to semidecide L:

```
1. Loop
        1.1 Move one square to the right.  If the character under
            the read head is an a, halt and accept.
```

Can we build M to decide L?

Example of Semideciding

L = `b`*`a`(`a` ∪ `b`)*. We can also decide *L*:

```
Loop:
    1.1 Move one square to the right.
    1.2 If the character under the read/write head is
        an a, halt and accept.
    1.3 If it is ❑, halt and reject.
```

However, as we will prove later, there are *L* in *SD* but not *D*.

Computing Functions

When a Turing machine halts, there is a value on its tape.

So, it can be used to compute functions.

Deciding (semideciding) a language *L* is computing the characteristic function (partial characteristic function) of *L*

Part of Chapter 25, use ch25.ppt

Why Are We Working with Our Hands Tied Behind Our Backs?

Turing machines	Are more powerful than any of the other formalisms we have studied so far. ☺
Turing machines	Are a **lot** harder to work with than all the real computers we have available. ☹

Why bother?

The very simplicity that makes it hard to program Turing machines makes it possible to reason formally about what they can do. If we can, once, show that anything a real computer can do can be done (albeit clumsily) on a Turing machine, then we have a way to reason about what real computers can do.

Turing Machines

Sections 17.3 – 17.5

Turing Machine Extensions

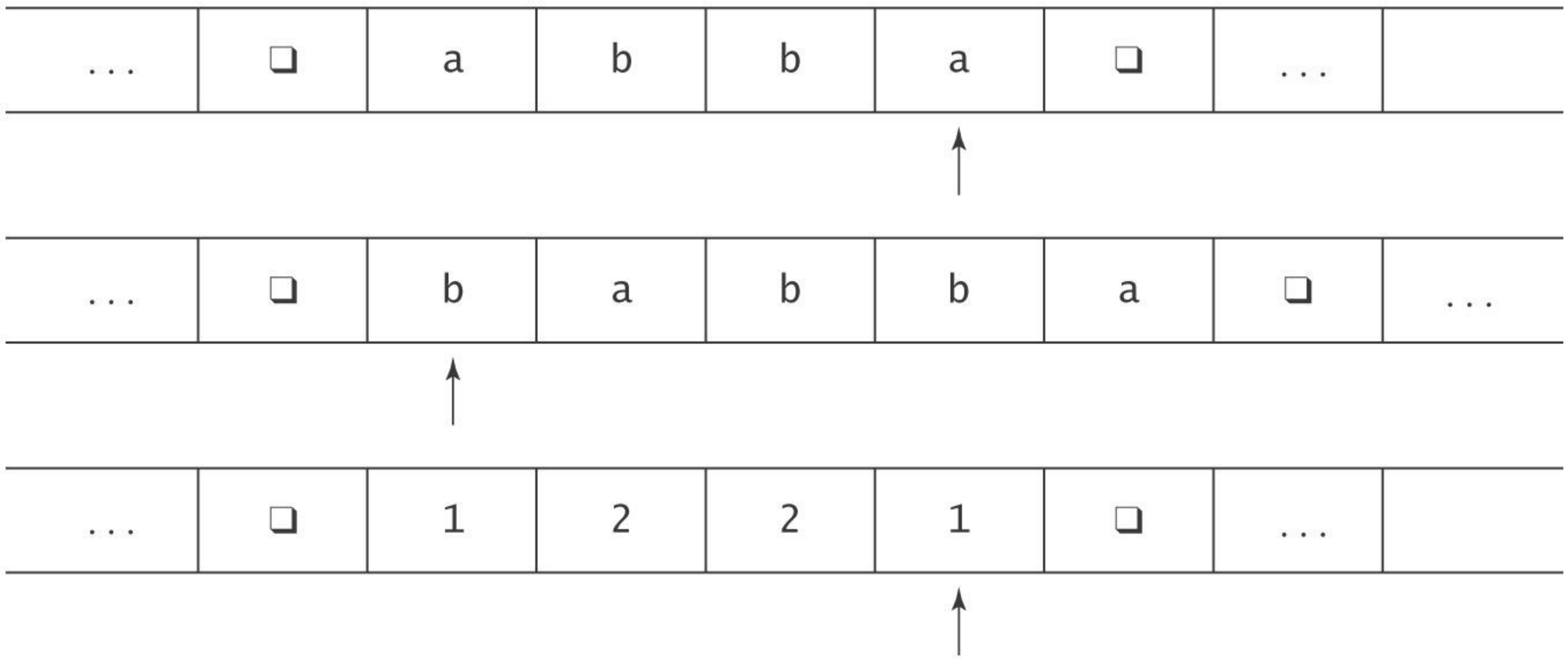
There are many extensions we might like to make to our basic Turing machine model. But:

We can show that every extended machine has an equivalent basic machine.

Some possible extensions:

- Multiple tape TMs
- Nondeterministic TMs

Multiple Tapes



Adding Tapes Adds No Power

Impact of Nondeterminism

Computability (decidability, solvability)

• FSMs	NO
• PDAs	YES
• Turing machines	NO

Complexity

Yes, adds power to TM.
Allows “lucky guesses”. In this sense, Nondeterminism adds “superpower”.
It may take exponentially more steps to solve a problem using a deterministic TM.

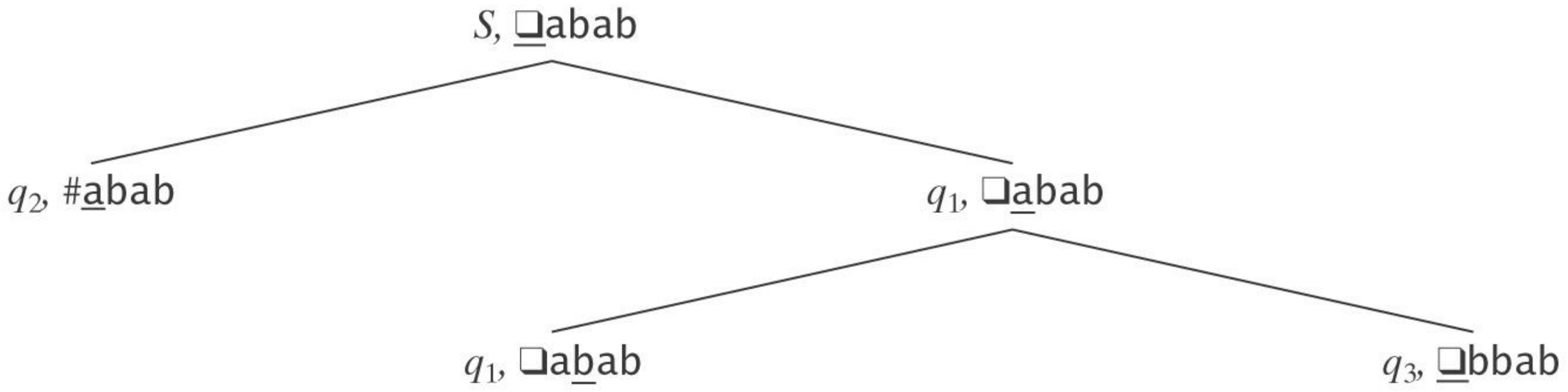
New player: Quantum Turing machine

Nondeterministic Turing Machines

A **nondeterministic** TM is a sixtuple (K, Σ, Γ, Δ, s, A).

Δ is a ***subset*** of:

$$(K \times \Gamma) \times (K \times \Gamma \times \{\leftarrow, \rightarrow\})$$



Nondeterministic Turing Machines

What does it mean for a nondeterministic Turing machine to:

- Decide a language
- Semidecide a language

Similarly,

M ***accepts*** *w* iff there exists *some path* that accepts it.
M ***rejects*** *w* iff all paths reject it.

- It is possible that, on input $w \notin L(M)$, *M* neither accepts nor rejects. In that case, no path accepts and some path does not reject.

Equivalence of Deterministic and Nondeterministic Turing Machines

Theorem: If a nondeterministic TM *M* decides or semidecides a language, or computes a function, then there is a standard TM *M'* semideciding or deciding the same language or computing the same function.

Proof: (by construction). We must do separate constructions for deciding/semideciding and for function computation.

Simulating a Real Computer

Theorem: A random-access, stored program computer can be simulated by a Turing Machine. If the computer requires n steps to perform some operation, the Turing Machine simulation will require $\mathcal{O}(n^6)$ steps.

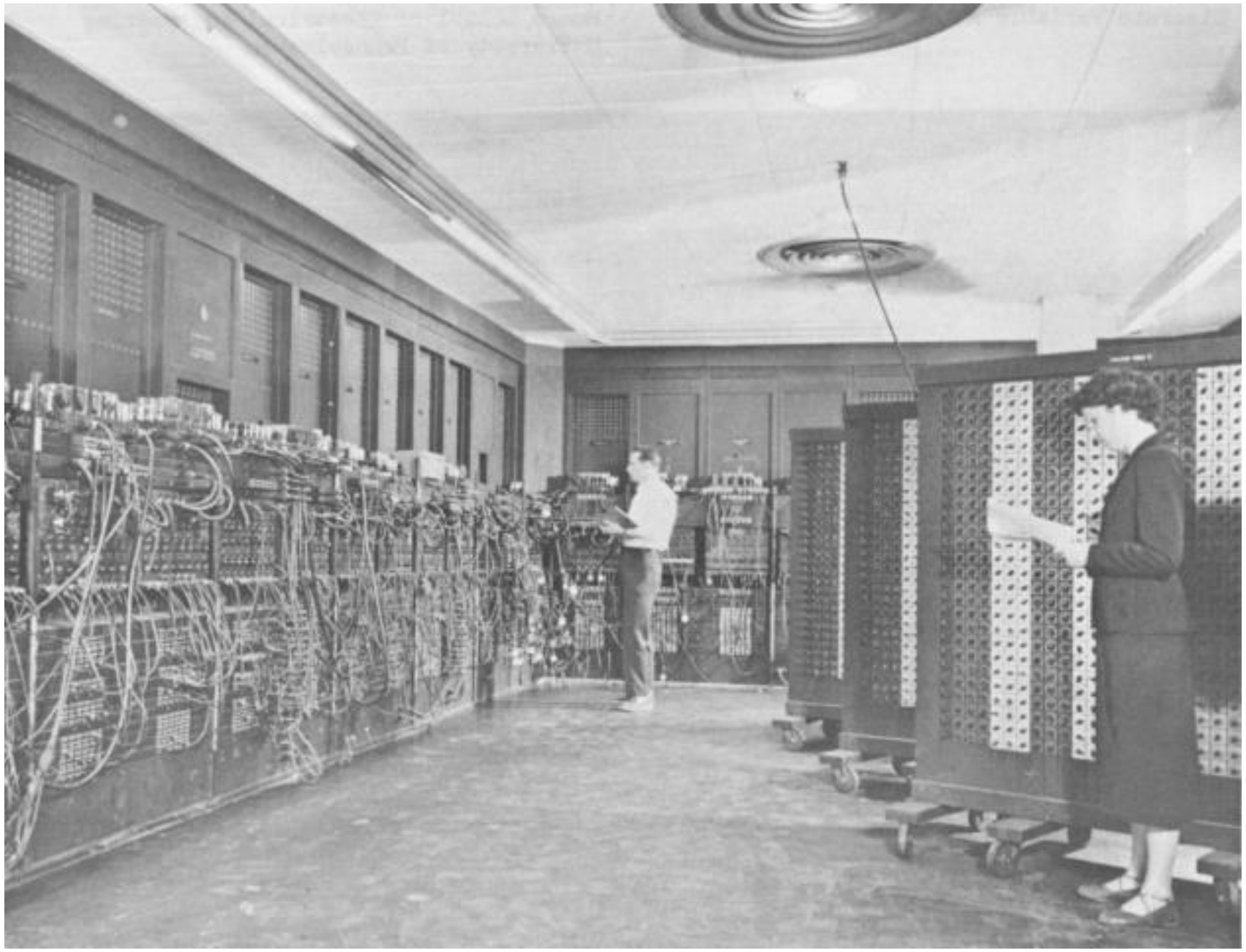
On the other hand, there are also lots of simulation of TMs by real computers, check it out on the web.

Turing Machines

Sections 17.6 – 17.7

The Universal Turing Machine

All our machines so far are hardwired.


ENIAC – 1945 (first electronic general-purpose computer)

The Universal Turing Machine

All our machines so far are hardwired.

Question: Can we build a programmable TM that accepts as input a (*M*: Turing machine, *w*: input string) pair and outputs whatever *M* would output when started up on *w*?

Answer: Yes, it's called the ***Universal Turing Machine***.

To define the Universal Turing Machine *U* we need to:

1. Define an encoding scheme that can be used to describe to *U* a (*M*, *w*) pair
   - <*M*, *w*>, many ways of encoding …
2. Describe the operation of *U* given input <*M*, *w*>

Specification of *U*

On input <*M*, *w*>, *U* must:

- Halt iff *M* halts on *w*.
- If *M* is a deciding or semideciding machine, then:
    - If *M* accepts, accept.
    - If *M* rejects, reject.
- If *M* computes a function, then *U*(<*M*, *w*>) must equal *M*(*w*).

Another Benefit of Encoding

Another benefit of defining a way to encode any Turing machine *M*:

- We can talk about operations on TMs, as they are just input strings.

In the following, *T* (a Turing machine) takes one TM as input and creates another as its output.

This idea of transforming one TM to another has extensive use in reduction, to show that various problems are undecidable.