Introduction to Turing Machines

Lecture 8

February 13, 2025

"Most General" computer?

1. DFAs are simple machine: accept only the regular languages.
2. CFLs (pushdown automata) more general class of languages/machines but also limited. Do not capture the language $\{a^n b^n c^n \mid n \geq 0\}$.
3. Is there a kind of computer that can accept any language, or compute any function?
4. Recall counting argument. Set of all languages is uncountably infinite. Set of all programs countably infinite. Hence there are languages for which there are no programs.

What can be computed?

Most General Computer:

1. If not all functions are computable, which are?
2. Is there a "most general" model of computer?
3. What languages can they recognize?

History: Formalizing mathematics

1. 19th century: *Ooops*. Math is a mess.
   Fix calculus, invented set theory (Cantor), etc.

History: Formalizing mathematics

1. 19th century: *Ooops*. Math is a mess.
   Fix calculus, invented set theory (Cantor), etc.
2. David Hilbert (1862–1943)
   1. 1900: The list of 23 problems.
   2. Early 1900s – crisis in math foundations
      attempts to formalize resulted in paradoxes, etc.
   3. 1920: Hilbert's Program: "mechanize" mathematics.
   4. Finite axioms, inference rules turn crank, determine truth
      needed: axioms consistent & complete

History: Formalizing mathematics

1. 19th century: *Ooops*. Math is a mess.
   Fix calculus, invented set theory (Cantor), etc.
2. David Hilbert (1862–1943)
   1. 1900: The list of 23 problems.
   2. Early 1900s – crisis in math foundations
      attempts to formalize resulted in paradoxes, etc.
   3. 1920: Hilbert's Program: "mechanize" mathematics.
   4. Finite axioms, inference rules turn crank, determine truth
      needed: axioms consistent & complete
3. Kurt Gödel (1906–1978)
   German logician, at age 25 (1931) proved: "There are true statements that cannot be proved or disproved". (i.e., "no" to Hilbert)
   Shook the foundations of mathematics/philosophy.

More history: Turing...

Alan Turing (1912–1954):

1. British mathematician
2. cryptoanalysis during WW II (enigma project)
3. Credited with AI/Theory of Computing
4. Defined a computing model (Turing machine). In 1936 (age 23) provided foundations for investigating fundamental question of what is computable, what is not computable via machines
5. Proved the halting theorem: Deciding if a computer program stops on a given input cannot be decided by a program.
6. Gay, convicted, commited suicide. Movies, UK apology.
7. Turing award named in his honor: highest prize in CS

Turing original paper...

Is quite readable. Available here:
https://www.cs.virginia.edu/~robins/Turing_Paper_1936.pdf

More history: Church ...

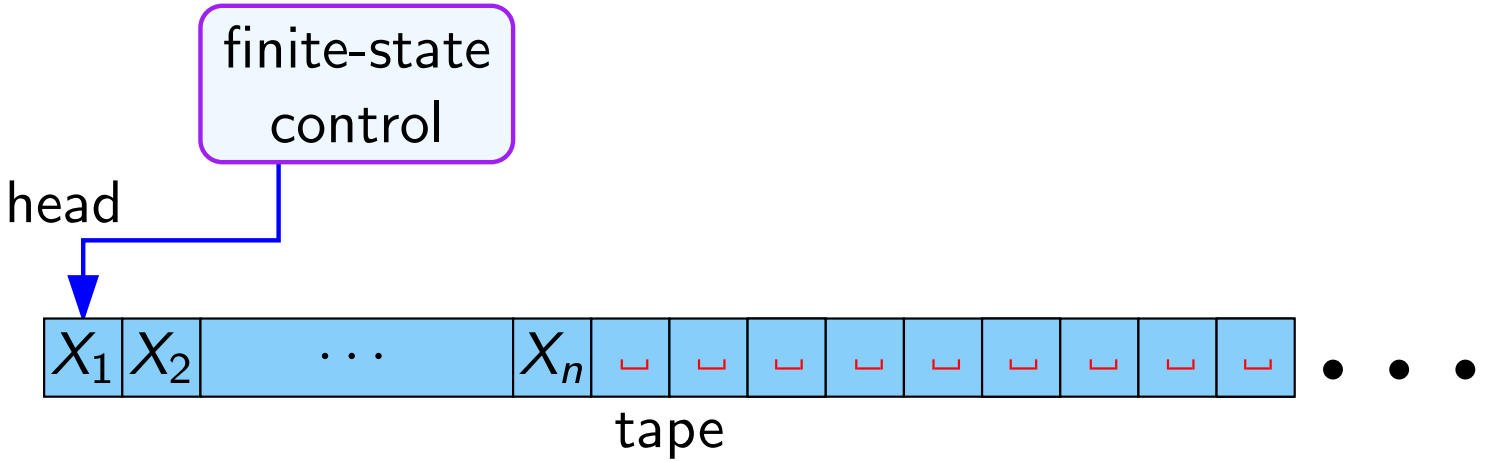
Alonzo Church (1903–1995):

1. American mathematician/logician/philospher/computer scientist
2. Defined λ-calculus before TMs
3. PhD advisor or Turing! Proved undecidablity before Turing but via logic. Turing came to the US to work with Church
4. Church-Turing thesis following proof of equivalence of TMs and λ-calculus by Church and Turing together
5. Many results in logic
6. Alonzo Church award in Logic and Computation

High level goals on TMs

1. What is the formal definition of a TM?
2. Church-Turing thesis: TMs are the most general computing devices. So far no counter example.
3. Every TM can be represented as a string.
4. Existence of **Universal** Turing Machine (UTM) which is the model/inspiration for stored program computing. UTM can simulate any TM.
5. Implications for what can be computed and what cannot be computed
6. Formal definition of time and space usage of algorithms, complexity classes, *P*, *NP* and others.
7. Connection to modern RAM model that underlies our machines/programming. the

Turing machine



1. Input written on (infinite) one sided tape.
2. Special blank characters.
3. Finite state control (similar to DFA).
4. Each step: Read character under head, **write** character out, move the head right or left or stay.

Turing machine: Formal definition

A **Turing machine** is a **7**-tuple

$$(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$$

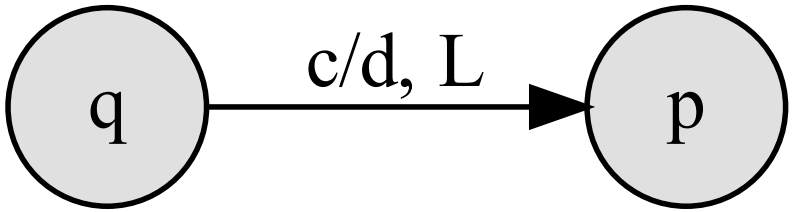
- Q: finite set of states.
- Σ: finite input alphabet.
- Γ: finite tape alphabet. $\Sigma \subset \Gamma$
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\mathtt{L}, \mathtt{R}, \mathtt{S}\}$: Transition function.
- $q_0 \in Q$ is the initial state.
- $q_{\text{acc}} \in Q$ is the **accepting**/**final** state.
- $q_{\text{rej}} \in Q$ is the **rejecting** state.
- $\sqcup$ or $\textvisiblespace$: Special blank symbol on the tape. We assume that $\sqcup \in \Gamma$ the tape alphabet

Turing machine: Transition function

$$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{\mathtt{L}, \mathtt{R}, \mathtt{S}\}$$

As such, the transition

$$\delta(q, c) = (p, d, \mathtt{L})$$



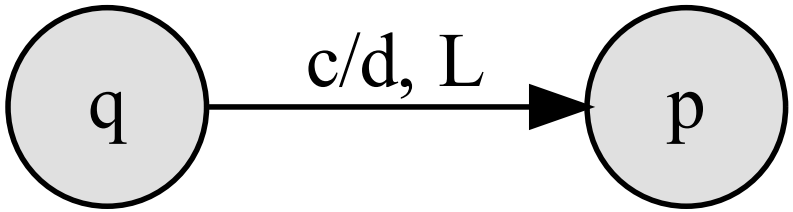
1. q: current state.
2. c: character **read** under tape head
3. p: new state.
4. d: character to **write** under tape head
5. L: Move tape head left.

Turing machine: Transition function

$$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{\mathtt{L}, \mathtt{R}, \mathtt{S}\}$$

As such, the transition

$$\delta(q, c) = (p, d, \mathtt{L})$$



1. q: current state.
2. c: character **read** under tape head
3. p: new state.
4. d: character to **write** under tape head
5. L: Move tape head left.

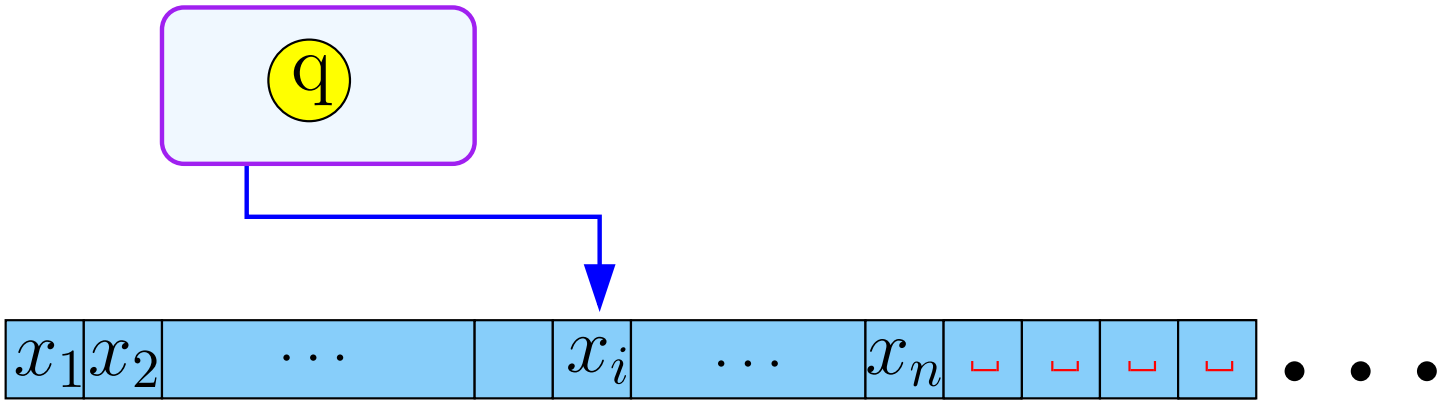
Missing transitions lead to hell state.
"Blue screen of death."
"Machine crashes."

Snapshot = ID: Instantaneous Description

1. Contains all necessary information to capture "state of the computation".
2. Includes
   1. state q of M
   2. location of read/write head
   3. contents of tape from left edge to rightmost non-blank (or to head, whichever is rightmost).

Snapshot = ID: Instantaneous Description

As a string



ID: $x_1 x_2 \ldots x_{i-1} q x_i x_{i+1} \ldots x_n$

$x_1, \ldots, x_n \in \Gamma$, $q \in Q$.

A step in computation as rewriting strings

$x_1 x_2 \ldots x_{i-1} q x_i x_{i+1} \ldots x_n$
If transition is $\delta(q, X_i) = (p, Y, \mathrm{L})$, new ID is:

$$\begin{array}{rl} \text{current ID :} & x_1 x_2 \ldots x_{i-2} x_{i-1} q x_i x_{i+1} \ldots x_n \\ \delta(q, X_i) = (p, y, \mathrm{L}) \implies & x_1 x_2 \ldots x_{i-2} p x_{i-1} y x_{i+1} \ldots x_n \end{array}$$

If no transition defined, or illegal transition, then no next ID (crash).

A step in computation as rewriting strings

$x_1 x_2 \ldots x_{i-1} q x_i x_{i+1} \ldots x_n$
If transition is $\delta(q, X_i) = (p, Y, \mathrm{L})$, new ID is:

$$\begin{array}{rl} \text{current ID :} & x_1 x_2 \ldots x_{i-2} x_{i-1} q x_i x_{i+1} \ldots x_n \\ \delta(q, X_i) = (p, y, \mathrm{L}) \implies & x_1 x_2 \ldots x_{i-2} p x_{i-1} y x_{i+1} \ldots x_n \end{array}$$

If no transition defined, or illegal transition, then no next ID (crash).

Shockingly: Computation is just a string rewriting system.

A step in computation as rewriting strings

1. Initial ID: $q_0 w$:
2. Accepting ID: $\alpha q_{\text{acc}} \alpha'$, for some $\alpha, \alpha' \in \Gamma^*$.
3. Rejecting ID: $\alpha q_{\text{rej}} \alpha'$, for some $\alpha, \alpha' \in \Gamma^*$.
4. $\mathcal{I} \rightsquigarrow \mathcal{J}$:Denotes that if we start execution of TM with configuration/ID encoded by $\mathcal{I}$, leads TM after *one time step* to ID $\mathcal{J}$
5. $\mathcal{I} \rightsquigarrow^* \mathcal{J}$:Denotes that if we start execution of TM with configuration/ID encoded by $\mathcal{I}$, leads TM to ID $\mathcal{J}$ after a finite number of time steps.

A step in computation as rewriting strings

1. Initial ID: $q_0 w$:
2. Accepting ID: $\alpha q_{\text{acc}} \alpha'$, for some $\alpha, \alpha' \in \Gamma^*$.
3. Rejecting ID: $\alpha q_{\text{rej}} \alpha'$, for some $\alpha, \alpha' \in \Gamma^*$.
4. $\mathcal{I} \rightsquigarrow \mathcal{J}$:Denotes that if we start execution of TM with configuration/ID encoded by $\mathcal{I}$, leads TM after *one time step* to ID $\mathcal{J}$
5. $\mathcal{I} \rightsquigarrow^* \mathcal{J}$:Denotes that if we start execution of TM with configuration/ID encoded by $\mathcal{I}$, leads TM to ID $\mathcal{J}$ after a finite number of time steps.
6. ***M* accepts *w***: If for some $\alpha, \alpha' \in \Gamma^*$, we have

$$q_0 w \rightsquigarrow^* \alpha q_{\text{acc}} \alpha'.$$

Acceptance happens as soon as TM enters accept state.

A step in computation as rewriting strings

1. Initial ID: $q_0 w$:
2. Accepting ID: $\alpha q_{\text{acc}} \alpha'$, for some $\alpha, \alpha' \in \Gamma^*$.
3. Rejecting ID: $\alpha q_{\text{rej}} \alpha'$, for some $\alpha, \alpha' \in \Gamma^*$.
4. $\mathcal{I} \rightsquigarrow \mathcal{J}$:Denotes that if we start execution of TM with configuration/ID encoded by $\mathcal{I}$, leads TM after *one time step* to ID $\mathcal{J}$
5. $\mathcal{I} \rightsquigarrow^* \mathcal{J}$:Denotes that if we start execution of TM with configuration/ID encoded by $\mathcal{I}$, leads TM to ID $\mathcal{J}$ after a finite number of time steps.
6. **M accepts w**: If for some $\alpha, \alpha' \in \Gamma^*$, we have

$$q_0 w \rightsquigarrow^* \alpha q_{\text{acc}} \alpha'.$$

   Acceptance happens as soon as TM enters accept state.
7. Language of TM M: $L(M) = \{w \in \Sigma^* \mid M \text{ accepts } w\}$

Non-accepting computation

M **does not accept** *w* if:

1. *M* enters q_{rej} (i.e., *M* **rejects** *w*)
2. *M* crashes (moves to left of tape, no transition available, etc).
3. *M* runs forever.

Non-accepting computation

M does not accept w if:

1. M enters q_{rej} (i.e., **M rejects w**)
2. M crashes (moves to left of tape, no transition available, etc).
3. M runs forever.

If the TM keeps running, should we wait, or is it rejection?

Recursive vs. Recursively Enumerable

1. **Recursively enumerable** (aka **RE**) languages

$$L = \{L(M) \mid M \text{ some Turing machine}\}.$$

2. **Recursive** / **decidable** languages

$$L = \{L(M) \mid M \text{ some Turing machine that halts on all inputs}\}.$$

Fundamental questions:

1. What languages are RE?
2. Which are recursive?
3. What is the difference?
4. What makes a language decidable?
5. Are there languages that are not even RE? What are those?

Recursive vs. Recursively Enumerable

1. **Recursively enumerable** (aka **RE**) languages (ok)

   $$L = \{L(M) \mid M \text{ some Turing machine}\}.$$

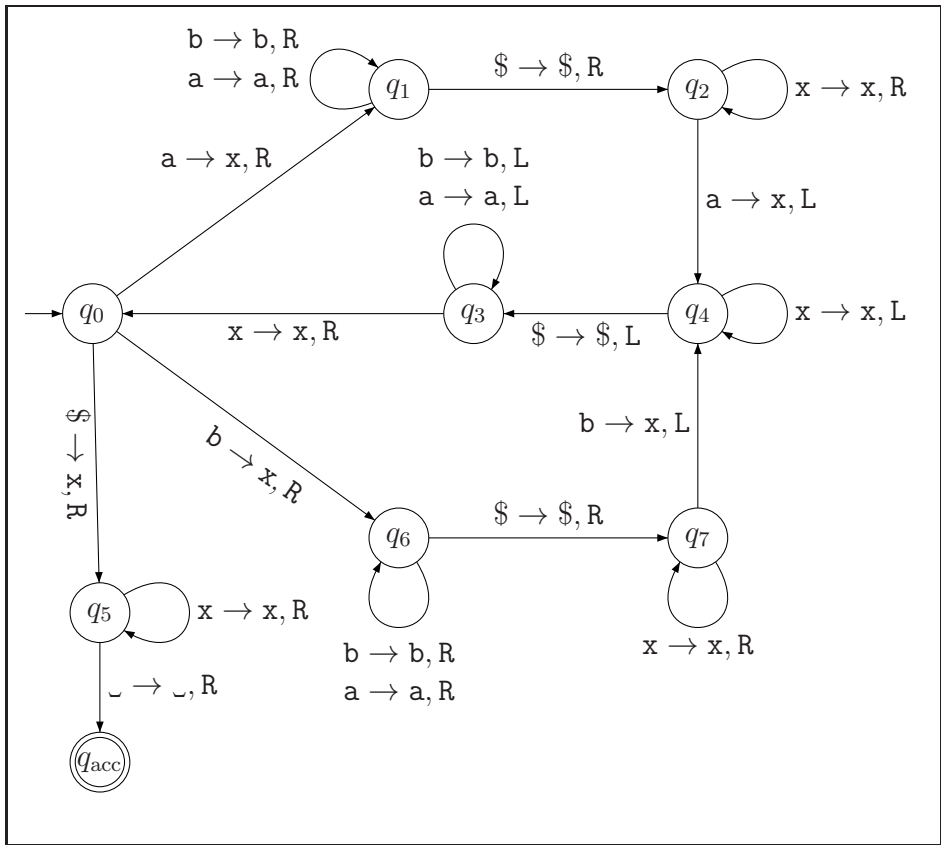
2. **Recursive** / **decidable** languages (good)

   $$L = \{L(M) \mid M \text{ some Turing machine that halts on all inputs}\}.$$

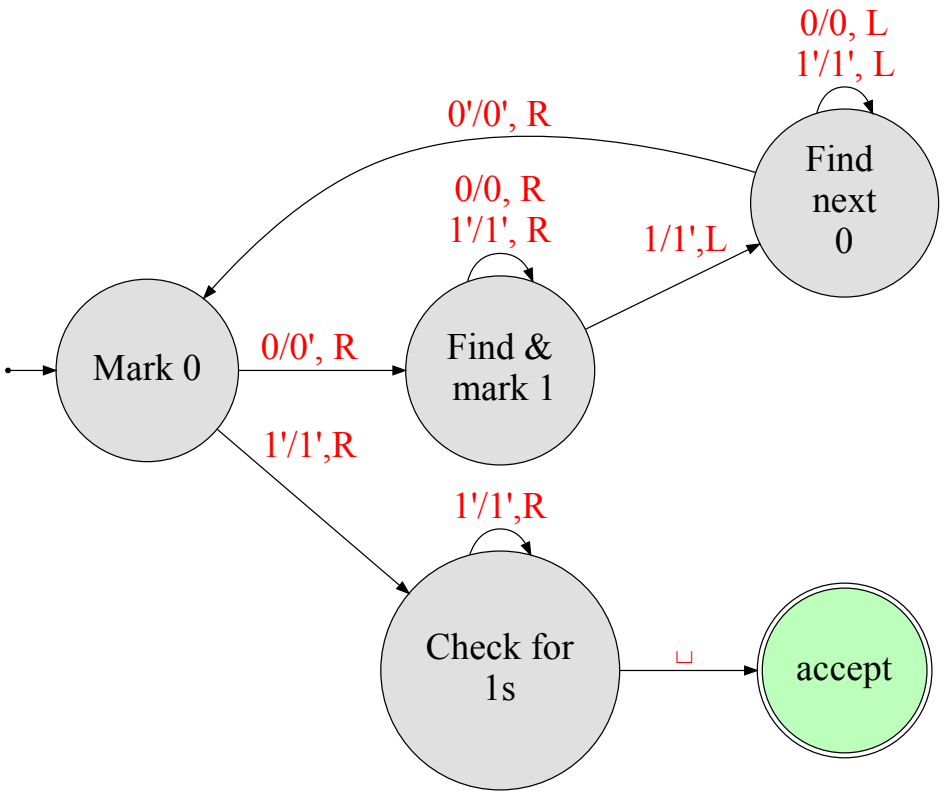
Fundamental questions:

1. What languages are RE?
2. Which are recursive?
3. What is the difference?
4. What makes a language decidable?
5. Are there languages that are not even RE? What are those?

Example: Turing machine for w$w

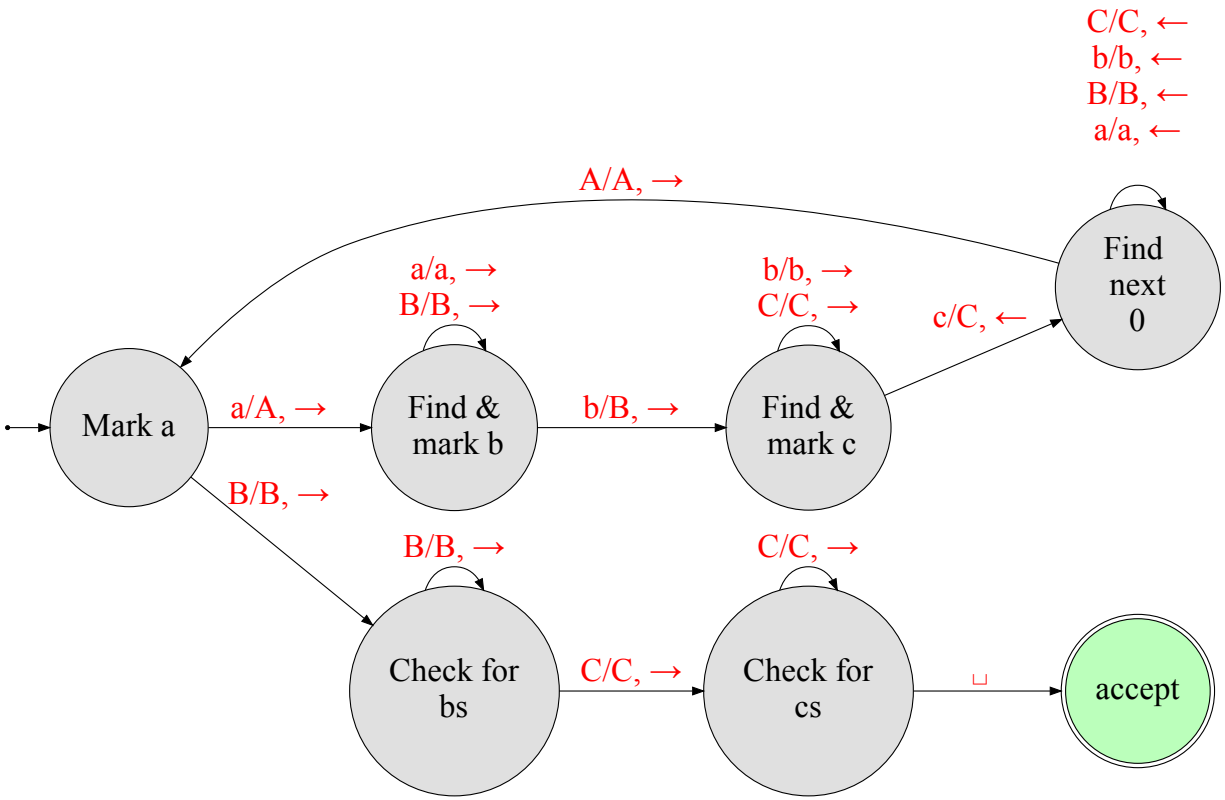


Example: Turing machine for 0^n1^n



Example: Turing machine for $a^n b^n c^n$

A language that is not context free...



Function Computation

So far we focused recognizing languages or equivalently computing Boolean functions (those that have a 0/1 or yes/no answer.

We also want machines to compute functions.

Focus on computing integer functions of the form $f : \mathbb{N} \rightarrow \mathbb{N}$. Functions on rationals, reals etc can be handled or approximated arbitrarily closely.

Informal: we say that M computes f if M started on input n written in binary stops and leaves $f(n)$ in binary on the tape (nothing else).

Multi-valued functions that take several inputs can also be defined similarly (addition of two numbers, other arithmetic operations, matrix operations, etc)

Extensions to TMs

TM can compute anything that a real computer can but very very very tediously. For this, various tricks and extensions

Why bother? Why is a TM defined in such a primite fashion?

Several reasons including:

- simplicity lays bare the essence of computation
- allows one to define certain aspects very formally without "cheating" inherent in most "practical" models
- allows one to prove equivalence with other models

Multi-tape/head TMs

k-tape TM

- k different infinite tapes
- k different *independently* controllable heads
- input initially on tape 1; other tapes have blank symbols
- single move of the macine
  - read symbols under all heads
  - print (possibly different) symbols under heads
  - move all heads (possibly different directions)
  - go to new state

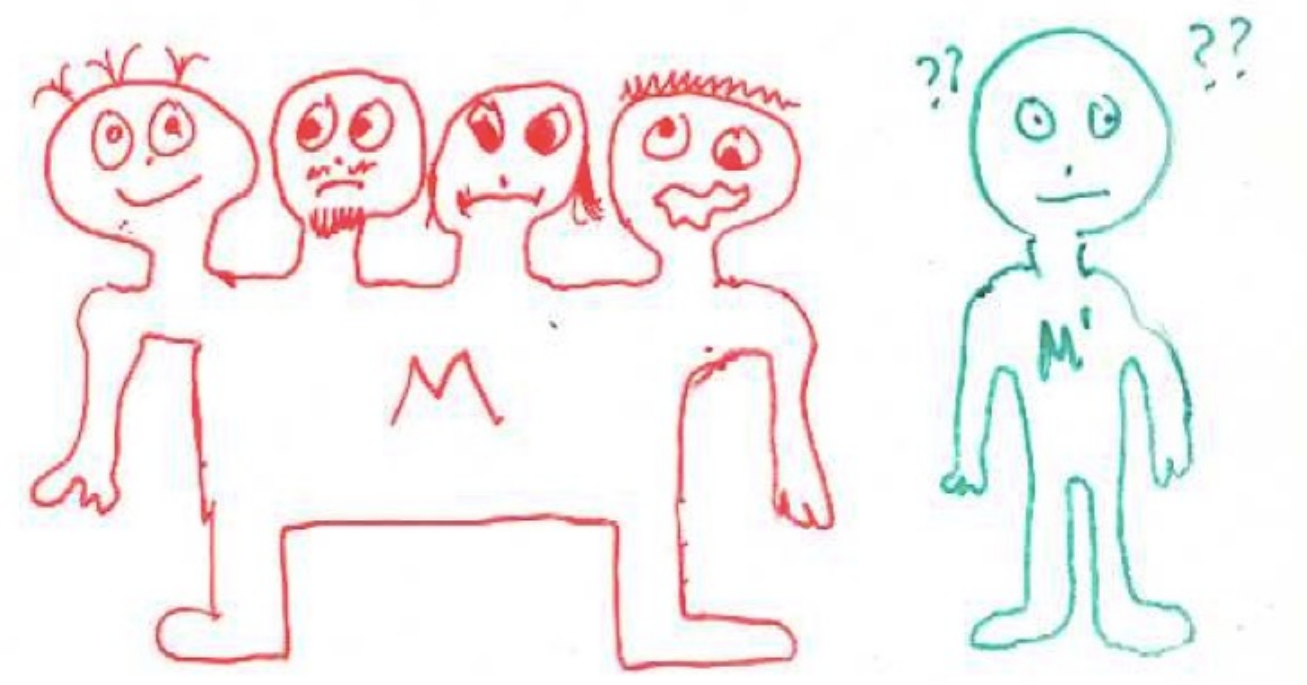
$$\delta : Q \times (\Sigma \cup \Gamma)^k \rightarrow Q \times (\Sigma \cup \Gamma)^k \times (\{L, R, S\})^k$$

$$\delta(q, a_1, a_2, \ldots, a_k) = (q', b_1, b_2, \ldots, b_k, D_1, D_2, \ldots, D_k)$$

Multi-tape/head TMs

Theorem

*If **L** is accepted by a **k**-tape TM **M** then **L** is accepted by a some **1**-tape TM **M′***



But then why?

Multi-tape/head TMs

Any advantage of k-tape TM?

Multi-tape/head TMs

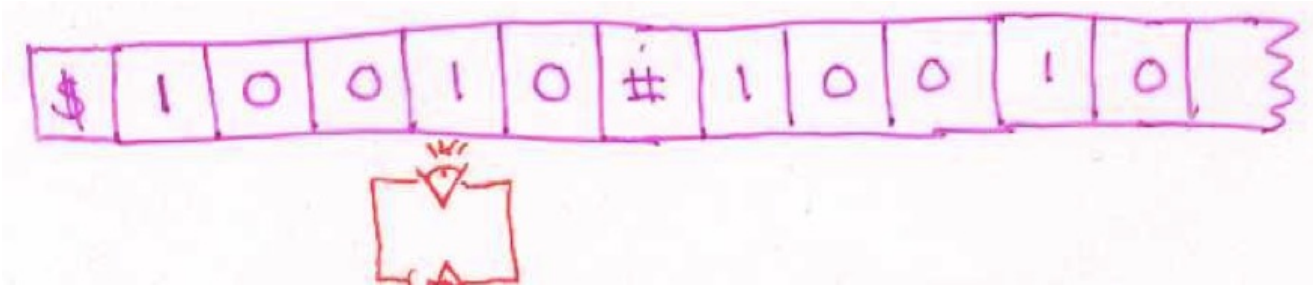
Any advantage of k-tape TM? First, it is much easier to "program" with k-tape machines.

Multi-tape/head TMs

Any advantage of k-tape TM? First, it is much easier to "program" with k-tape machines. More fundamental reason below.

Multi-tape/head TMs

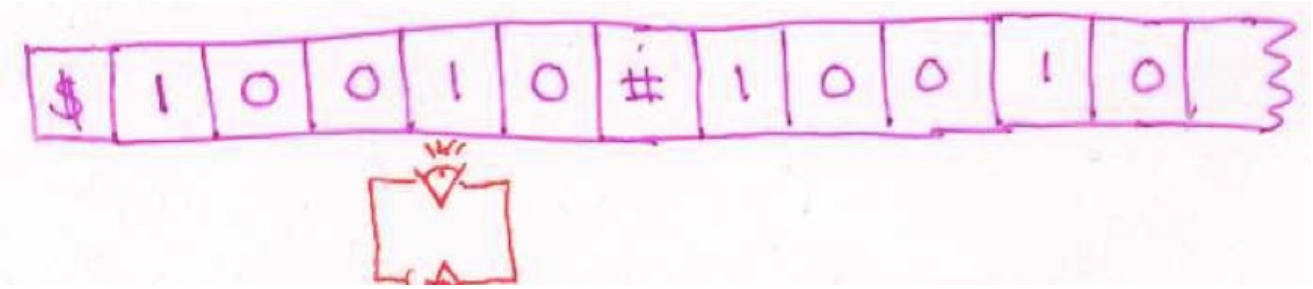
Any advantage of k-tape TM? First, it is much easier to "program" with k-tape machines. More fundamental reason below. Consider $L = \{w\$w \mid w \in \{0,1\}^*\}$



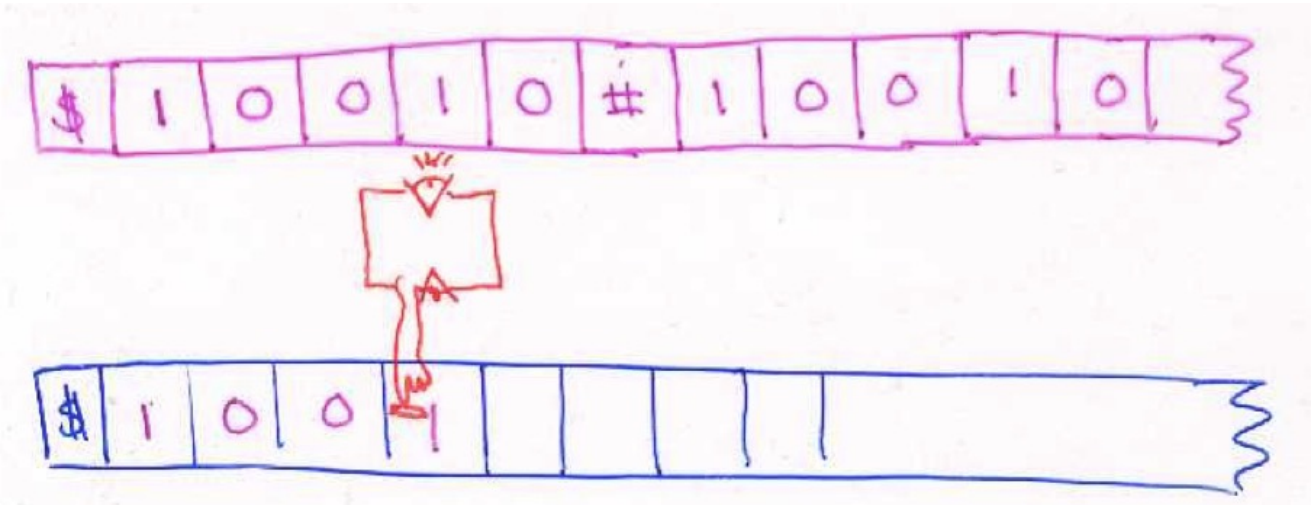
Any single tape TM takes $\Omega(n^2)$ steps for L.

Multi-tape/head TMs

Any advantage of k-tape TM? First, it is much easier to "program" with k-tape machines. More fundamental reason below. Consider $L = \{w\$w \mid w \in \{0,1\}^*$



Any single tape TM takes $\Omega(n^2)$ steps for L. Easy to design a 2-tape TM that takes $3n/2$ steps



Two tapes vs more

Any advantage of k-tape TM?

We saw that **2**-tapes provide substantial speed up over **1**-tape

Two tapes vs more

Any advantage of k-tape TM?

We saw that **2**-tapes provide substantial speed up over **1**-tape

Theorem

*Any k-tape TM that runs in T steps can be simulated by a **2**-tape TM in $O(T \log T)$ steps.*

Thus, **2**-tape TM is a more **robust** model to define time complexity etc.

Encoding TMs as strings and numbers

A **Turing machine** is a **7**-tuple

$$(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$$

Encoding TMs as strings and numbers

A **Turing machine** is a **7**-tuple

$$(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$$

Simple but powerful idea

Encoding TMs as strings and numbers

A **Turing machine** is a **7**-tuple

$$(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$$

Simple but powerful idea

Every TM M can be described as unique string

Encoding TMs as strings and numbers

A **Turing machine** is a **7**-tuple

$$(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$$

Simple but powerful idea

Every TM M can be described as unique string

Every TM M can be described as a unique binary string, and hence every TM M can be mapped to a unique natural number

Encoding TMs as strings and numbers

A **Turing machine** is a **7**-tuple

$$(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$$

Simple but powerful idea

Every TM M can be described as unique string

Every TM M can be described as a unique binary string, and hence every TM M can be mapped to a unique natural number

Integers that do not correspond to a valid TM can be thought of as encoding a junk TM

Encoding TMs as strings and numbers

Lemma

If $L \subseteq \{0,1\}^*$ is accepted by a TM M then there is a one-tape TM M' that accepts L such that:

- $\Gamma = \{0, 1, B\}$
- *states numbered* 1 *to* k *for some finite integer* k
- q_1 *is the unique start state*
- q_2 *is the unique accept state*
- q_3 *is the unique halt/reject state*

Encoding TMs as strings and numbers

Lemma

If $L \subseteq \{0,1\}^*$ is accepted by a TM M then there is a one-tape TM M' that accepts L such that:

- $\Gamma = \{0, 1, B\}$
- *states numbered 1 to k for some finite integer k*
- q_1 *is the unique start state*
- q_2 *is the unique accept state*
- q_3 *is the unique halt/reject state*

Thus, to represent a TM we only need to specify k and the transition function δ. Everything else is implicit.

Listing transitions

Use the following order for specifying δ

$$\delta(q_1, 0), \delta(q_1, 1), \delta(q_1, B), \ldots, \delta(q_k, 0), \delta(q_k, 1), \delta(q_k, B)$$

Use the following encoding

$$111 t_1 11 t_2 11 t_3 11 \ldots 11 t_{3k} 111$$

Encoding a transition

Recall transition looks like $\delta(q, a) = (p, b, L)$

Encode as follows

$\langle state \rangle 1 \langle input \rangle 1 \langle new - state \rangle 1 \langle new - symbol \rangle 1 \langle direction \rangle$

where

- q_i is represented by 0^i
- $0, 1, B$ reprsented by $0, 00, 000$
- L, R, S represented by $0, 00, 000$

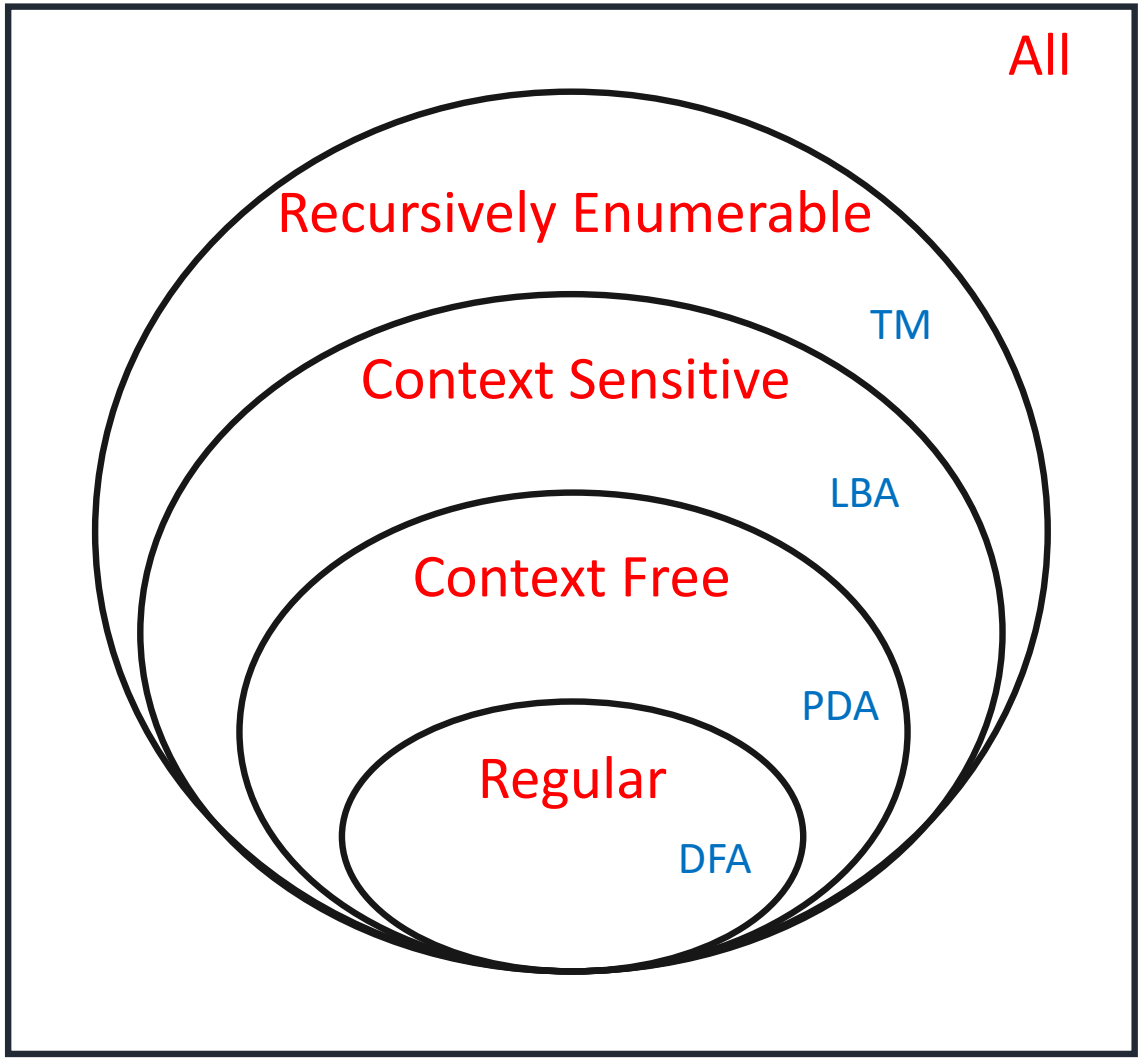
$\delta(q_3, 1) = (q_4, 0, R)$ is encoded as 0001001000010100

Typical TM Code

111010100001001001101001000001010 11 ..... 11 ..... 11 111

- Begins, ends with **111**
- Only thing to specify is δ. Can keep k implicit in description of δ or explicit at the start.
- Transistions separated by **11**
- Fields within transitions separated by **1**
- Individual fields represented by **0**s

TMs and Grammars



Are there grammars corresponding to TMs?

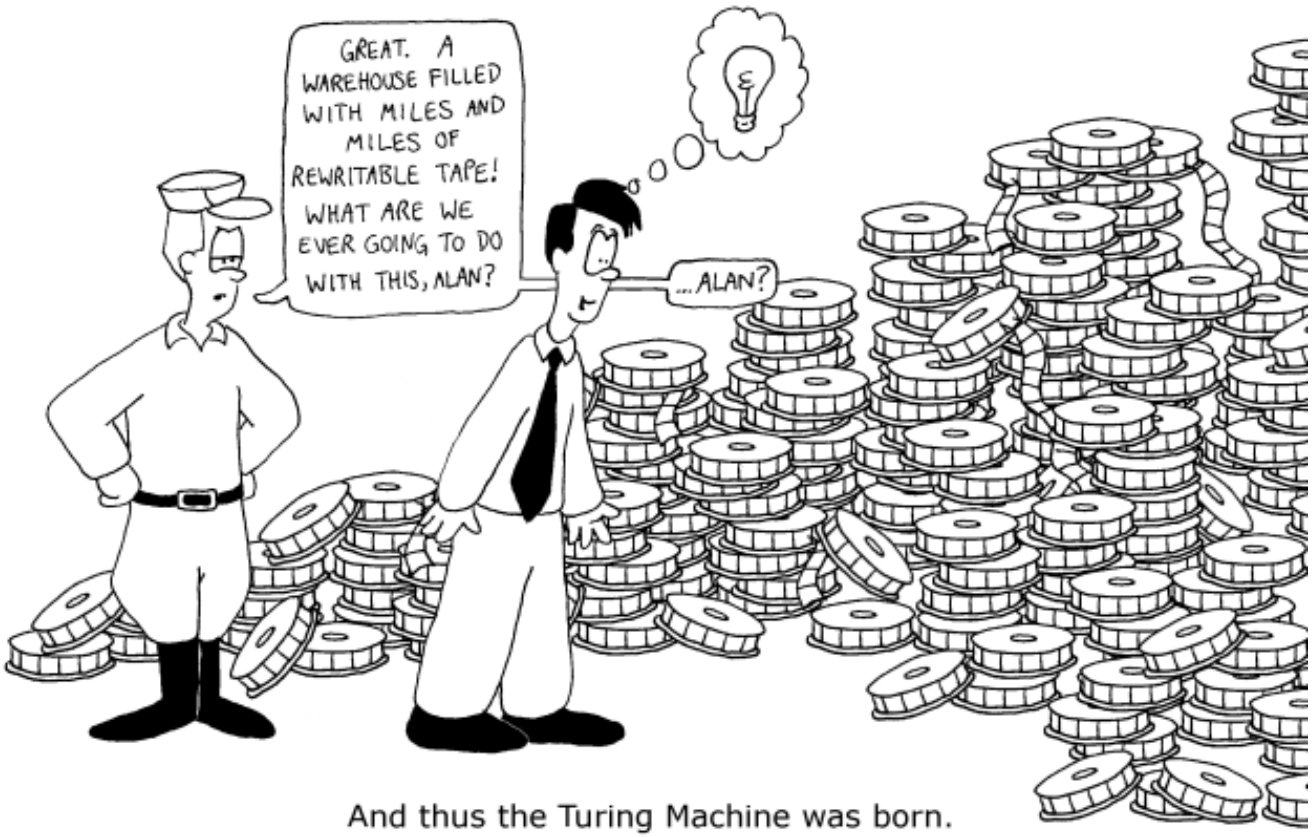
TMs and Grammars

Grammars: $G = (V, T, P, S)$ like in CFLs with non-terminals V, terminals/alphabet T, production rules P and start symbol S.

- **Regular:** Only rules of the form $A \to a$, $A \to \varepsilon$, $A \to aB$ (right-regular to be precise)
- **Context-free:** Only rules of the form $A \to \gamma$ where A is a non-terminal in V and $\gamma \in (V \cup T)^*$
- **Context-sensitive:** rules of the form $\alpha A \beta \to \alpha \gamma \beta$ which allow expansion of a *single* non-terminal in the presence of the context α and β.
- **Unstructured:** rules of the form $\alpha \to \beta$, no restrictions essentially. String rewriting.

Languages generated by unrestricted grammars are equivalent to languages accepted by TMs (recursively enumerable)

How was the Turing Machine invented...



DFAs and TMs

Manuel Blum: Turing award winner, eminent professor

From his advice to graduate students
https://www.cs.cmu.edu/~mblum/research/pdf/grad.html

STUDYING: You are all computer scientists. You know what FINITE AUTOMATA can do. You know what TURING MACHINES can do. For example, Finite Automata can add but not multiply. Turing Machines can compute any computable function. Turing machines are incredibly more powerful than Finite Automata. Yet the only difference between a FA and a TM is that the TM, unlike the FA, has **paper and pencil**. Think about it. It tells you something about the power of writing. Without writing, you are reduced to a finite automaton. With writing you have the extraordinary power of a Turing machine.