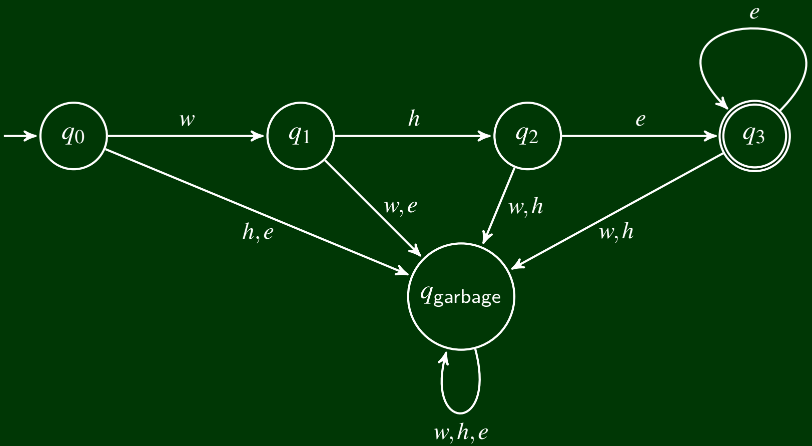
15
251

Great Theoretical Ideas in Computer Science

Finite State Machines I



Outline

A Language is a set of strings.
We can list out all programs. We use the notation $\langle P \rangle$ to mean "the number representing the program P".

There are many models of computation:

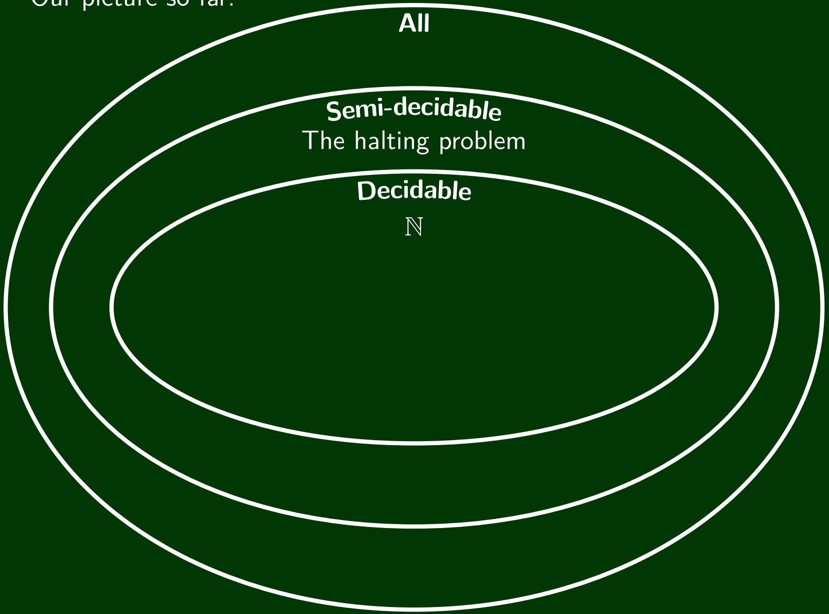
Models of Computation

You've already seen **register machines**.
You will see **Lambda Calculus** next week.
We'll discuss several more today, including **Turing Machines**!

The main question we've discussed so far is:

For a particular language L, is L decidable?

Our picture so far:



A Simple Program

Let's consider the following code:

```
# Input: b_n b_{n-1} b_{n-2} ··· b_2 b_1 b_0
  low = 0
  hi = n
  while low < hi:
    if b_low != b_hi:
      return false
    low++
    hi--
  return true
```

Okay, now let's pretend that our input is given as a stream. So, we can only read from left to right, and once we've consumed a bit, it's gone:

Input:	0	1	1	0	1	0	1	1	0	1		···
	$\rightarrow$											

We're also explicitly given memory to work with. Think of it as a linked list, where each node has a bit or is blank. It starts out empty.

Work:												···
	$\uparrow$											

New Machine, new program

Input:	0	1	1	0	1	0	1	1	0	1		⋯
	→											

Work:												⋯
	↑											

- Copy the input to the work tape:

Work:	0	1	1	0	1	0	1	1	0	1		⋯
										↑		

- Erase the last bit, go to the front, and check that it's the same:

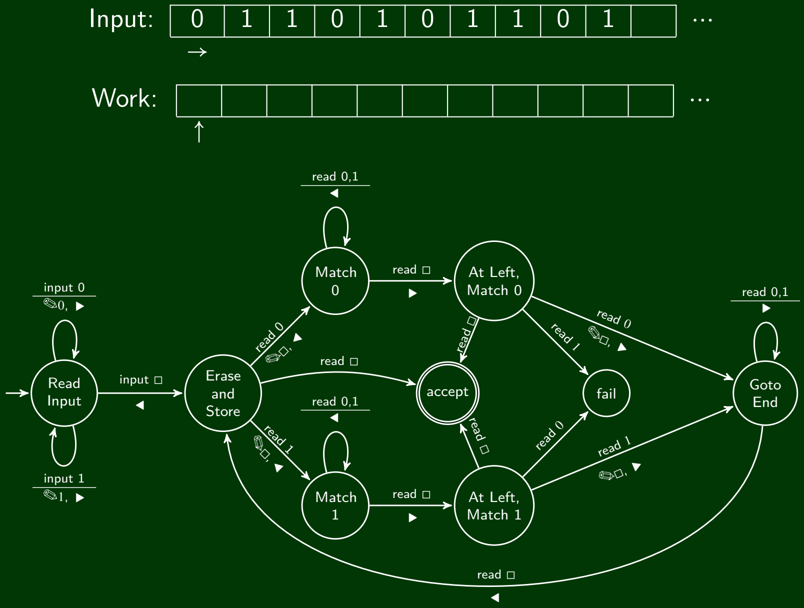
Work:	0	1	1	0	1	0	1	1	0			⋯
	↑											

- If it isn't, return false. If it is, go back to the end and repeat step (2):

Work:		1	1	0	1	0	1	1	0			⋯
									↑			

We could just write another program to do this, but let's write a flow chart instead.

New Machine, new ~~program~~ flowchart



Some infinite tapes: (how many doesn't matter; one tape for input and work, etc.)

Input:

0	1	1	0	1	0	1	1	0	1		⋯

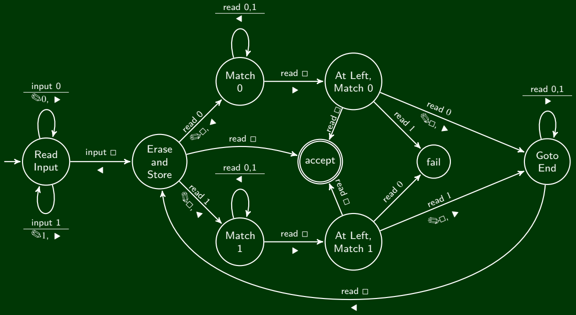
→

Work:

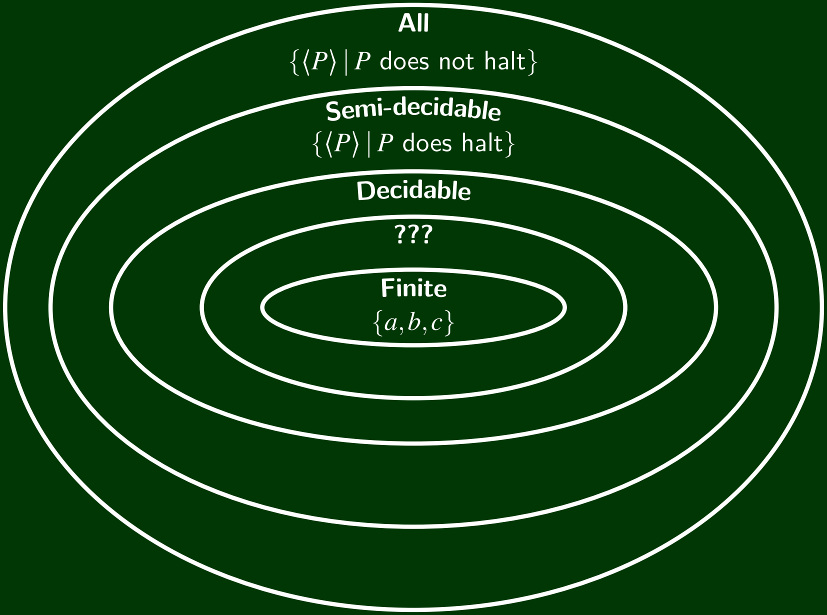
											⋯

↑

A finite-state controller:

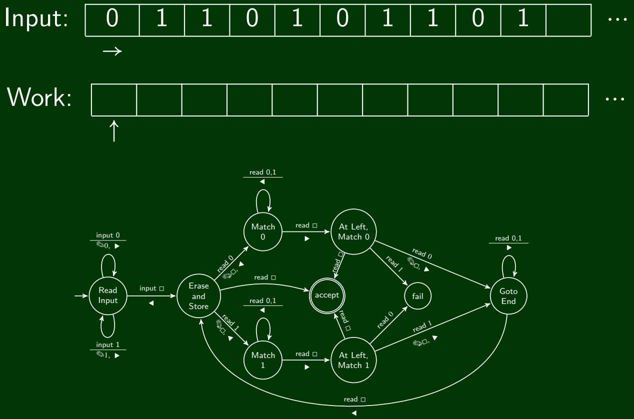


That's it. These things can decide exactly the same languages as register machines, and lambda calculus, and...LaTeX.


All
$\{\langle P\rangle \mid P \text{ does not halt}\}$
Semi-decidable
$\{\langle P\rangle \mid P \text{ does halt}\}$
Decidable
???
Finite
$\{a,b,c\}$

Like a TM, but Stupider

Remember, a Turing Machine has three pieces: an input tape, a work tape, and a controller:



If we wanted something dumber than a TM, but not quite as dumb as a decider for finite sets, what could we do?

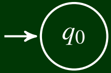
Kill the “work” tape!


All
$\{\langle P \rangle \mid P \text{ does not halt}\}$
Semi-decidable
$\{\langle P \rangle \mid P \text{ does halt}\}$
Decidable
Regular
Finite
$\{a, b, c\}$
???
???

Deterministic Finite Automata

A Deterministic Finite Automaton (or DFA) is a TM which reads exactly one character of the input on each transition. It has no work tape; so, it can't write anything down; so, this definition makes sense.

Like before, we denote the start state with a lone arrow:

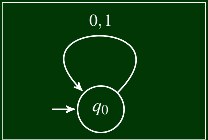


and accept states are double circles:

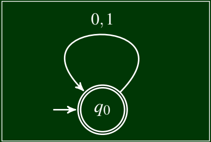
We also specify an alphabet that strings may range over: $\Sigma = \{0, 1\}$ (before we had $\square$ as an additional symbol).
Here's the simplest two DFAs:

We say that the language of a machine M, written $\mathcal{L}(M)$ is the set of strings it accepts.

M_1



M_2



$$\mathcal{L}(M_1) = \varnothing \qquad \mathcal{L}(M_2) = \Sigma^*$$

BTW, if A is a set, A^* is called the Kleene Closure of A.

$$A^* = A^0 \cup A^1 \cup A^2 \cup \cdots$$

Definition (DFA)

A DFA M is a 5-tuple:

$$M = (Q, \Sigma, \delta, q_0, F)$$

where $\delta : Q \times \Sigma \to Q$, $q_0 \in Q$, $F \subseteq Q$

- Q is a finite set of states
- Σ is a finite alphabet
- δ is a transition function between states
- q_0 is the start state
- F is a set of final states

And you don't have to draw them manually:

`https://whiteboard.ddt.cs.cmu.edu/dfas/latex`

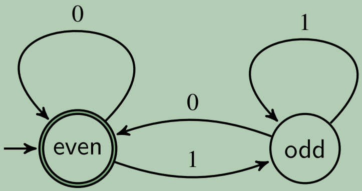
Note that $\delta : Q \times \Sigma \to Q$ takes a character as its second argument. It would be nicer if it took in a string. We will assume that $\delta : Q \times \Sigma^* \to Q$ does the right thing. That is,

$$\delta(q, x_0 x_1 \cdots x_n) = \delta(\delta(\cdots\delta(\delta(q, x_0), x_1)\cdots, x_{n-1}), x_n)$$

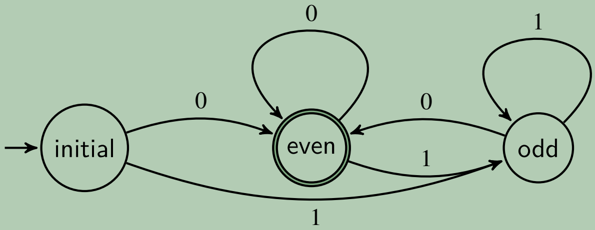
Let $L_{\text{even}} = \{x \in \{0,1\}^* \mid x, \text{ interpreted as binary, is even}\}$.

Find a DFA M_{even}, such that $\mathcal{L}(M_{\text{even}}) = L_{\text{even}}$.

How about this one?



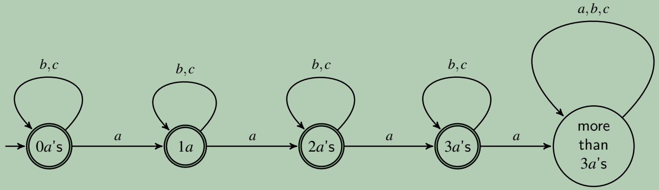
Okay, better.



Let $L_{3a\text{'s}} = \{x \in \{a,b,c\}^* \mid x$ has no more than 3 a's$\}$.

Find a DFA $M_{3a\text{'s}}$, such that $\mathcal{L}(M_{3a\text{'s}}) = L_{3a\text{'s}}$.

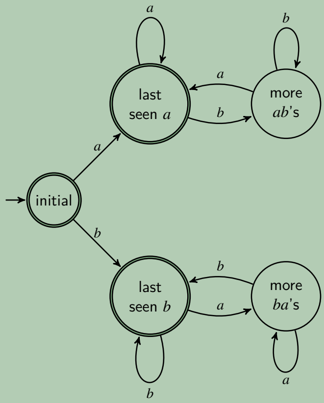
Here we go!



Let $L_{ab=ba} = \{x \in \{a,b\}^* \mid x$ has an equal $\#$ of substrings "ab", "ba"$\}$.

Find a DFA $M_{ab=ba}$, such that $\mathcal{L}(M_{ab=ba}) = L_{ab=ba}$.

Here it is!



Let $L'_{ab=ba} = \{x \in \{a,b,c\}^* \mid x$ has an equal # of substrings "ab", "ba"$\}$.

Find a DFA $M'_{ab=ba}$, such that $\mathcal{L}(M'_{ab=ba}) = L'_{ab=ba}$.

Uh oh. Well, if you can't find something, maybe it doesn't exist...

Proving a Language L is not Regular

- Assume it is regular. Therefore, there exists some machine M such that $\mathcal{L}(M) = L$.
- It's a **finite** state machine...so, let's say it has n states.
- Our goal is to show that this machine is broken. What does it mean for a DFA to be broken?

Well, we can basically attack the states or the transition function. Which seems more useful?
Insight: The transition function is complicated. But the states have one bit of information. Either they accept, or not.

Well...what if some state s did both!

Proving a Language L is not Regular

- Assume it is regular. Therefore, there exists some machine M such that $\mathcal{L}(M) = L$.
- It's a **finite** state machine...so, let's say it has n states.
- We want to make the machine tell us that some state s both **accepts** and **rejects**.
- Feed the machine a **ton of strings**. How many? $n+1$, because then **two of them must end in the same state**, by pigeonhole.
- Now, we have S_1 and S_2, where $\delta(q_0, S_1) = \delta(q_0, S_2)$. So, what?
- Choose **one** string X so that S_1X should be accepted, but S_2X shouldn't be.
- Then we get a contradiction, because

$$\delta(q_0, S_1X) = \delta(\delta(q_0, S_1), X) = \delta(\delta(q_0, S_2), X) = \delta(q_0, S_2X)$$

Let $L'_{ab=ba} = \{x \in \{a,b,c\}^* \mid x$ has an equal # of substrings "ab", "ba"$\}$.

Proving $L'_{ab=ba}$ is not Regular

- Assume it is regular. Therefore, there exists some machine M such that $\mathcal{L}(M) = L$. And M has n states.
- Feed the machine $n+1$ strings. The string we **add onto the end** can only have one number of ab's and one number of ba's.

Now, we prove it!

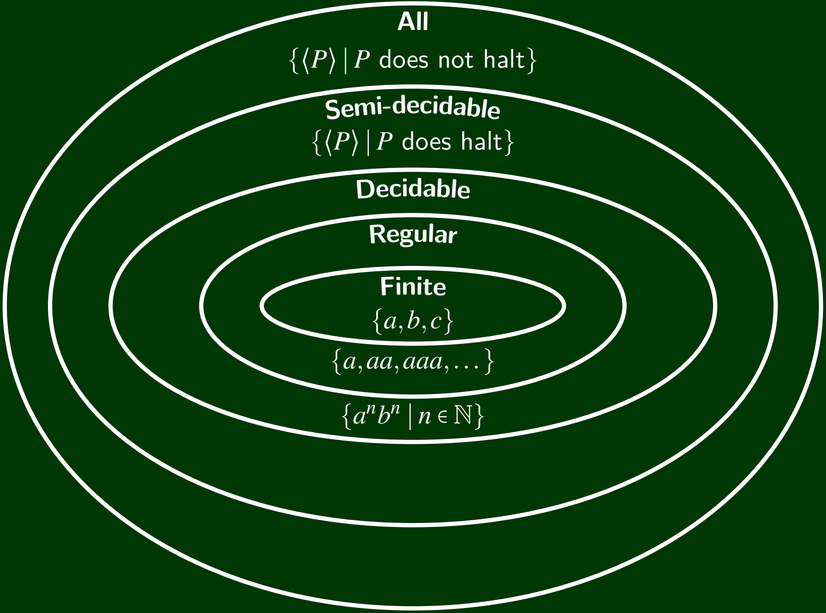
Let $L'_{ab=ba} = \{x \in \{a,b,c\}^* \mid x$ has an equal # of substrings "ab", "ba"$\}$.

Proving $L'_{ab=ba}$ is not Regular

- Assume it is regular. Therefore, there exists some machine M such that $\mathcal{L}(M) = L$. And M has n states.
- Feed the machine $n+1$ strings. Consider $\{(abc)^k \mid k \in \mathbb{N}\}$. Also, $\infty > n$.
- Now, we have $(abc)^x$ and $(abc)^y$, where $\delta(q_0, (abc)^x) = \delta(q_0, (abc)^y)$, and $x \neq y$ by pigeonhole.
- Choose **one** string X so that $(abc)^x X$ should be accepted, but $(abc)^y X$ shouldn't be. Let's try $X = (bac)^x$.
- Then we get a contradiction, because

$$\delta(q_0, (abc)^x (bac)^x) = \delta(q_0, (abc)^y (bac)^x)$$

- That is, $(abc)^x (bac)^x \in L$, and $(abc)^y (bac)^x \notin L$. So, $\delta(q_0, (abc)^x (bac)^x)$ must be accepting and rejecting! That's obviously not possible.

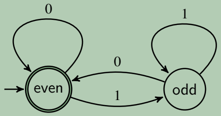

All
$\{\langle P\rangle \mid P \text{ does not halt}\}$
Semi-decidable
$\{\langle P\rangle \mid P \text{ does halt}\}$
Decidable
Regular
Finite
$\{a,b,c\}$
$\{a,aa,aaa,\dots\}$
$\{a^nb^n \mid n\in\mathbb{N}\}$

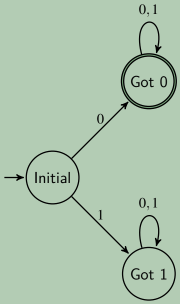
Suppose we have a regular language L_1 and another regular language L_2. How do we construct a machine M such that $\mathcal{L}(M) = L_1 \cup L_2$?

Idea!

We have DFAs for L_1 and L_2; call them M_1 and M_2. We can run both machines at the same time.

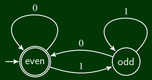
L_1 and L_2



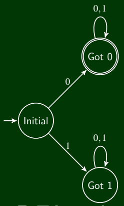


To run these machines at the same time, we "keep a finger" on a state in each machine. If either one accepts, our new machine should too.

L_1:

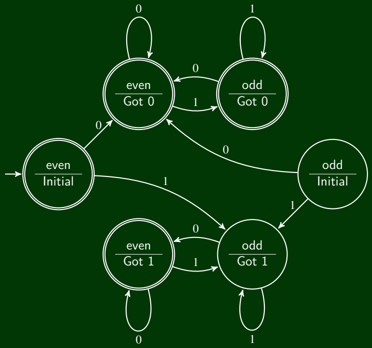


L_2:



How can we make a DFA out of this idea? Make a new DFA, where each state is made of one state from the left and one state from the right.

$L_1 \cup L_2$:



If we have two DFAs

$$M_1 = (Q_1, \Sigma, \delta_1, q_0, F_1) \qquad M_2 = (Q_2, \Sigma, \delta_2, q'_0, F_2)$$

then we can construct a DFA $M_1 \cup M_2$ such that

$$\mathcal{L}(M_1 \cup M_2) = \mathcal{L}(M_1) \cup \mathcal{L}(M_2)$$

as follows:

$$M_1 \cup M_2 = (Q_1 \times Q_2, \Sigma, \delta_\cup, (q_0, q'_0), \{(q_1, q_2) \in Q_1 \times Q_2 \mid q_1 \in F_1 \vee q_2 \in F_2\})$$

where

$$\delta_\cup((q_1, q_2), \sigma) = (\delta_1(q_1, \sigma), \delta_2(q_2, \sigma))$$

Can we do intersection?

1. DFAs can match any set of finite strings, S.
2. DFAs can match the Kleene Closure of a set of characters, Σ^*
3. DFAs can match the union of two sets of strings, $S_1 \cup S_2$.
4. DFAs can match one arbitrary character, $?$
5. DFAs can match the concatenation of two sets of strings, $S_1 \cdot S_2$.

Suppose I have a long piece of text, say

The History of Twitch Plays Pokemon: Generation 1

Let's use the notation `[a-z]` $= \{a, b, \cdots, y, z\}$, etc.

1. `{p,P}` $\cdot$ `ok` $\cdot$ `?` $\cdot$ `[a-z]` $\cdot$ `[a-z]*`
2. `pik` $\cdot$ `{a}*`
3. `DUX`
4. `{A,B,C,D}` $\cdot$ `{A,B,C,D}*`

`({p,P}` $\cdot$ `ok` $\cdot$ `?` $\cdot$ `[a-z]` $\cdot$ `[a-z]*)` $\cup$ `(pik` $\cdot$ `{a}*)` $\cup$ `(DUX)` $\cup$ `({A,B,C,D}` $\cdot$ `{A,B,C,D}*)`

`[pP]ok.[a-z][a-z]*\|pika*\|DUX\|[ABCD][ABCD]*\)[,.:]*`

What we're doing is actually grep!

The Smaller the Better?

$\vee$

SFMs