Intro to Foundations of CS; slides 1, 2017-18

Prof. Paul W. Goldberg (Dept. of Computer Science, University of Oxford)

2017-18

General information

Paul Goldberg

email: Paul.Goldberg@cs.ox.ac.uk

administrative info, e.g. problem class times, office hours, at:

`http://www.cs.ox.ac.uk/people/paul.goldberg/FCS/index.html`

notes and slides to appear on my FCS web pages (above)

problem class sign up at

`https://www.cs.ox.ac.uk/minerva/student_signup.pl`

General information

- Text: Introduction to the Theory of Computation 2nd (latest uk) Edition by Mike Sipser
- 6 problem sheets
- Slides will be available online shortly before lectures
- but don't just read slides instead of lecture...
- Some material covered in the exercises!
- Read the book (the relevant parts).

Others: *Intro to Automata Theory, Languages, and Computation* by Hopcroft, Motwani, and Ullman; *Computational Complexity* by Papadimitriou; *Models of Computation: Exploring the Power of Computing* by Savage; . . .

Ideally you are familiar with:

- programming and basic algorithms
- asymptotic notation "big-oh"
- sets, graphs
- proofs, especially induction proofs and proof by contradiction

Chapter 0 of text

What it's about

- "Problem" — basic notion in CS. A (mathematically well-defined computational challenge
- e.g. travelling salesman problem; search for max element of a list; primality testing; testing equivalence of boolean circuits
- A problem can be expressed as a *language recognition* challenge: specify a syntax that represents instances (or inputs), then want to recognise certain words in that syntax (and, compute associated outputs). Usually enough to think in terms of recognising subsets of words.
- we look at mechanisms for recognising (formal) languages, also logic (propositional, first-order), and its expressive power. Properties of these mechanisms (e.g. Turing machines; equivalence to generic programming). Classify languages according to how powerful a mechanism is needed
- We obtain (mathematical) positive/negative results (some language can/cannot be recognised using some mechanism)

Topics

Part I: Begin with computation models having limited power

- Finite Automata and Regular Languages
- Pushdown Automata and Context Free Grammars

Where these are usable, problem has very efficient solution; these models are captured by specialised programming languages

Part II: models of general computing (corresponding to the question: is this problem solvable by a computer at all)

Part III: modelling efficient computation.

Part IV: we look at formal logic, and apply previous parts to look at computational problems related to logic.

Provides essential background for other courses, e.g. complexity, logic automata and games

Computational problems as language recognition

- problem: compute function $f : \mathit{inputs} \longrightarrow \mathit{outputs}$
- Simple. Can we make it simpler?
- Yes. Decision problems:

$$f : \mathit{inputs} \longrightarrow \{\mathit{accept}, \mathit{reject}\}$$

- Does this still capture our notion of problem, or is it too restrictive?
- Example: factoring:
  - given an integer m, find its prime factors

$$f_{\mathit{factor}} : \{0, 1\}^* \longrightarrow \{0, 1\}^*$$

- Decision version:
  - given 2 integers m,k, accept iff m has a prime factor $p < k$

Can certainly use the latter to solve the former.

Terminological Summary

- finite **alphabet** Σ : a set of symbols
- **language** $L \subseteq \Sigma^*$: subset of strings over Σ
- a **machine** takes an input string and either
  - accepts, rejects, or
  - loops forever
- a machine **recognizes** the set of strings that lead to accept
- a machine **decides** a language L if it accepts $x \in L$ and rejects $x \notin L$

"machine"...?

$$\textit{input} \xrightarrow{\textbf{machine}} \begin{cases} \bullet \text{ accept} \\ \bullet \text{ reject} \\ \bullet \text{ loop forever.} \end{cases}$$

- We define **simple** mathematical formalisations of computation
- Strategy:
  - endow box with a feature of computation
  - try to **characterize** the languages decided
  - if we see that box is too weak for real-life tasks, add new feature to overcome limits

Finite Automata

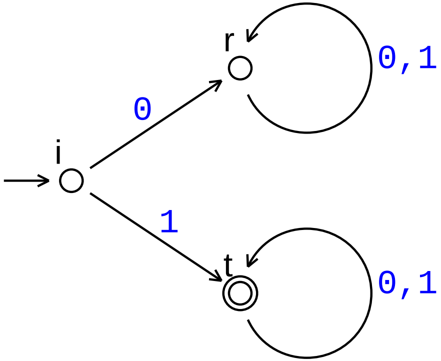
- simple model of computation
- reads input from left to right, one symbol at a time
- maintains **state**: information about what seen so far ("memory")
  - finite automaton has finite number of states: cannot remember more things for longer inputs
- start with *deterministic finite automata* (DFA)
- 2 ways to describe: by diagram, or formally

Software

Rogers and Finleys JFlap is software for creating state machines and seeing their properties. You can get a free copy at www.jflap.org

If you find any nice-looking interactive web sites, let me know...

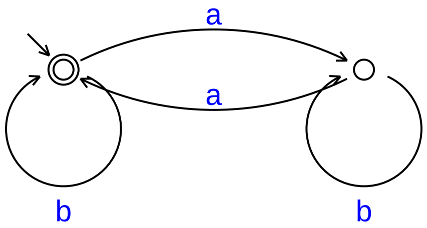
DFA diagrams



3 states, *i*, *r* and *t*. State *i* is *initial* or *starting* state. Alphabet $\Sigma = \{0, 1\}$. Read input one letter at a time; follow arrows. Accepting state *t* denoted by outgoing arrow/double circle. In general, multiple accept states are allowed.

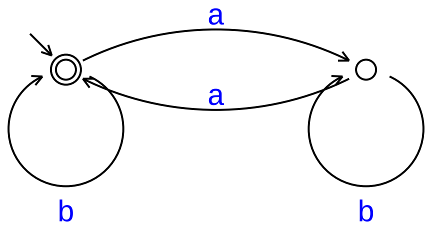
Let's consider what language **L** is accepted. Note the way DFA provides an unambiguous description of **L**.

Another example DFA



- What language does this DFA recognize?

Another example DFA



- What language does this DFA recognize?

$$L = \{x : x \in \{\mathtt{a}, \mathtt{b}\}^*, x \text{ has even number of } \mathtt{a}s\}$$

- illustrates fundamental feature/limitation of FA:
  - “tiny” memory
  - in this example only “remembers” 1 bit of info.

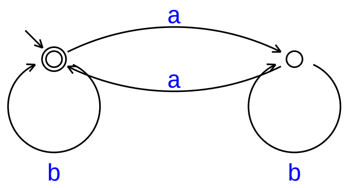
DFA formal definition

A deterministic finite automaton is a 5-tuple

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

- Q is a finite set called the **states**
- Σ is a finite set called the **alphabet**
- $\delta : Q \times \Sigma \longrightarrow Q$ is a function called the **transition function**
- q_0 is an element of Q called the **start state**
- F is a subset of Q called the accept states

DFA formal definition: example



Specification of this FA in formal terms: Let **even** and **odd** denote the states

- $Q = \{\textbf{even}, \textbf{odd}\}$
- $\Sigma = \{\texttt{a}, \texttt{b}\}$
- $q_0 = \textbf{even}$
- $F = \{\textbf{even}\}$
- $\delta(\textbf{even}, \texttt{b}) = \textbf{even}$; $\delta(\textbf{even}, \texttt{a}) = \textbf{odd}$; $\delta(\textbf{odd}, \texttt{b}) = \textbf{odd}$; $\delta(\textbf{odd}, \texttt{a}) = \textbf{even}$;

Formal description, continued

Operation

A deterministic finite automaton

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

accepts a string

$$w = \mathbf{w_1 w_2 w_3} \ldots \mathbf{w_n} \in \Sigma^*$$

if there is a sequence $r_0, r_1, r_2, \ldots, r_n$ of states for which

- $r_0 = q_0$
- $\delta(r_i, w_{i+1}) = r_{i+1}$ for $i = 0, 1, 2, \ldots, n-1$
- $r_n \in F$

Characterizing DFA languages

DFAs are a *model of computation*. Q: what kinds of languages can DFAs accept?
(formally: *characterise* the languages they can accept; some well-defined languages can't be accepted by DFAs.)

Important main result on DFA languages: DFA languages can be built up from other DFA languages by combining them via certain operations: union, concatenation, and closure (star, Kleene closure). Equivalent to languages described using *regular expressions* (details coming up). This is *Kleene's theorem*.

regular expressions building blocks

A regular expression can be a simple finite list of words, or can be built up from other r.e.'s using:

- union "$C = (A \cup B)$" or "$C = A + B$" or "$C = A|B$"

$$C = \{x \ : \ x \in A \text{ or } x \in B \text{ or both}\}$$

- concatenation "$C = (A \circ B)$", or just "C=AB"

$$C = \{xy \ : \ x \in A \text{ and } y \in B\}$$

- star (Kleene star, or Kleene closure) "$C = A^*$" (note: ϵ always in A^*)

$$C = \{x_1 x_2 x_3 \ldots x_k \ : \ k \geq 0 \text{ and each } x_i \in A\}$$

Combine DFAs to simulate regular expressions

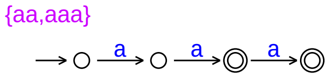
As noted, *regular expressions* are built up from alphabet letters, and operations union, concatenation and star.

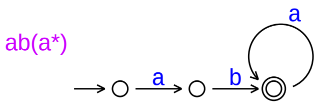
To show that regular expressions have the same expressive power as DFAs, we need to be able to simulate the building-blocks of reg. exps as operations on DFAs.

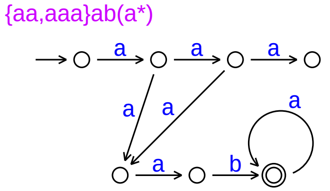
For example, given 2 DFAs, how do we build a new one that accepts the concatenation of the languages accepted by the 2 DFAs?

(It would be convenient to allow transitions labelled by empty string...)

Example: trying to simulate concatenation





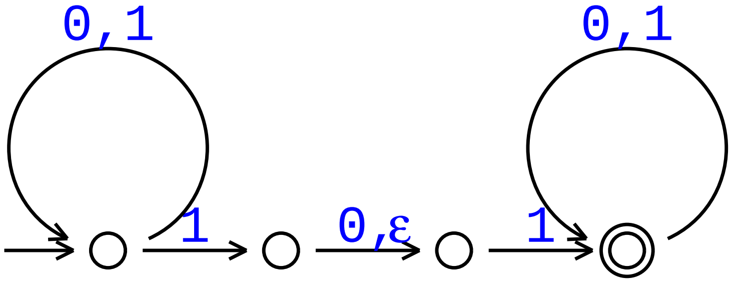


Nondeterministic FA (NFA)

- We will make life easier by describing an additional feature (**nondeterminism**) that helps us to “program” DFAs
- We will **prove** that FAs with this new feature can be **simulated** by ordinary DFA
- The concept of **nondeterminism** has a significant role in TCS and this course.

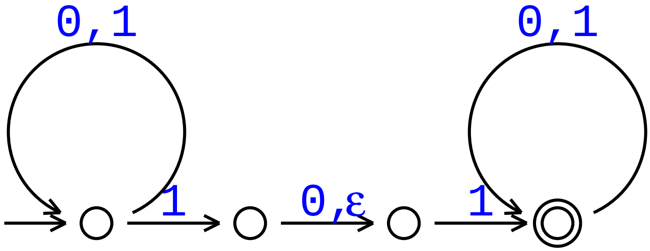
NFA diagrams

- single start state
- transitions may:
  - have several with a given label out of the same state
  - be labelled with ϵ



NFA operation

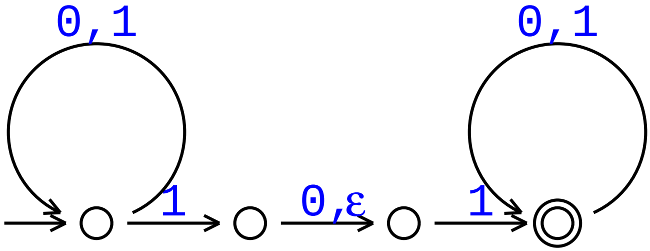
Example of NFA operation (alphabet $\Sigma = \{0, 1\}$)



input: 0 1 0
not accepted

NFA operation

Example of NFA operation (alphabet $\Sigma = \{0, 1\}$)



input: 1 1 0
accepted

NFA operation

- One way to think of NFA operation:
- string $x = x_1x_2x_3 \ldots x_n$ accepted if and only if
    - there exists a way of inserting ϵ's into x

$$x_1\epsilon\epsilon x_2x_3 \ldots \epsilon x_n$$

    - so that **there exists** a path of transitions from the start state to an accept state

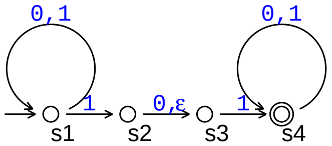
NFA formal definition

Let $\mathcal{P}(S)$ denote the set of all subsets of a set S. A *nondeterministic finite automaton* (or NFA) is a quintuple $(Q, \Sigma, \delta, q_0, F)$ where

1. Q is a finite nonempty set whose members are called **states**;
2. Σ is a finite nonempty set called the **alphabet**;
3. δ is a map from $Q \times (\Sigma \cup \{\epsilon\})$ to $\mathcal{P}(Q)$ called the **transition function** of the automaton;
4. q_0 is a member of Q and is called the **start state**;
5. F is a subset of Q whose members are called **accepting states**.

Other equivalent definitions exist; there's other standard notation...

NFA formal definition: example



Specification of this DFA in formal terms:

- $Q = \{s1, s2, s3, s4\}$
- $\Sigma = \{0, 1\}$
- $q_0 = s1$
- $F = \{s4\}$
- $\delta(s1, 0) = \{s1\}$; $\delta(s1, 1) = \{s1, s2\}$; $\delta(s1, \epsilon) = \{\}$; $\delta(s2, 0) = \{s3\}$; $\delta(s2, 1) = \{\}$; $\delta(s2, \epsilon) = \{s3\}$; $\delta(s3, 0) = \{\}$; $\delta(s3, 1) = \{s4\}$; $\delta(s3, \epsilon) = \{\}$; $\delta(s4, 0) = \{s4\}$; $\delta(s4, 1) = \{s4\}$; $\delta(s4, \epsilon) = \{\}$

Formal description of NFA operation

NFA $M = (\mathbf{Q}, \mathbf{\Sigma}, \delta, \mathbf{q_0}, \mathbf{F})$
accepts a string $w = \mathbf{w_1 w_2 w_3 \ldots w_n} \in \Sigma^*$ if w can be written (by inserting ϵ's) as:

$$y = \mathbf{y_1 y_2 y_3 \ldots y_m} \in (\Sigma \cup \{\epsilon\})^*$$

and $\exists$ sequence $r_0, r_1, \ldots, r_m$ of states for which

- $r_0 = q_0$
- $r_{i+1} \in \delta(r_i, y_{i+1})$ for $i = 0, 1, 2, \ldots, m-1$
- $r_m \in F$

Closures

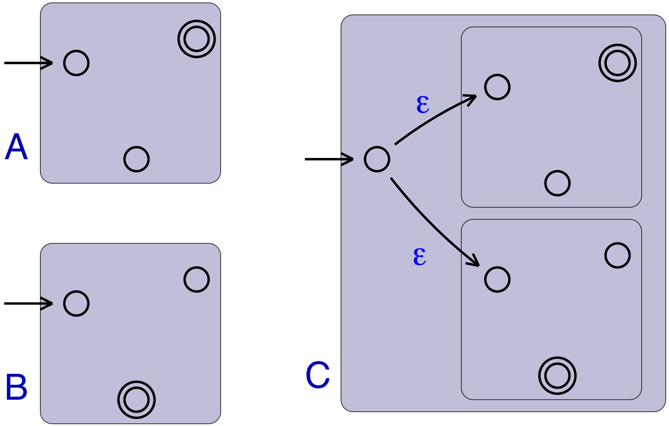
Next we show: the set of languages recognized by **NFA** is closed under:

- **union** "$C = (A \cup B)$"
- **concatenation** "$C = (A \circ B)$"
- **star** "$C = A^*$"

(this is more easily done for NFAs than for DFAs. We then show how to convert NFA to equivalent DFA)

Closure under union

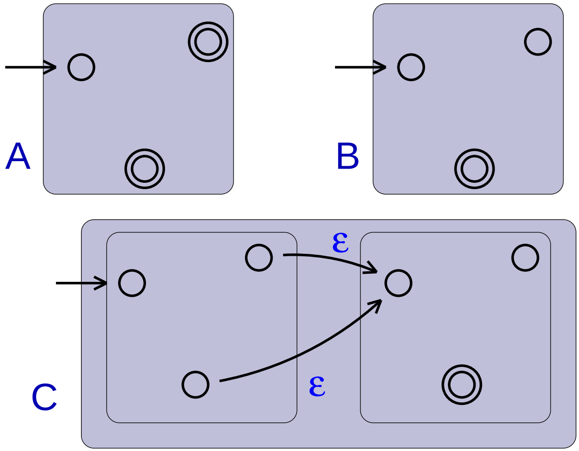
$$C = (A \cup B) = \{x \ : \ x \in A \text{ or } x \in B\}$$



How would you prove that this works? (exercise)

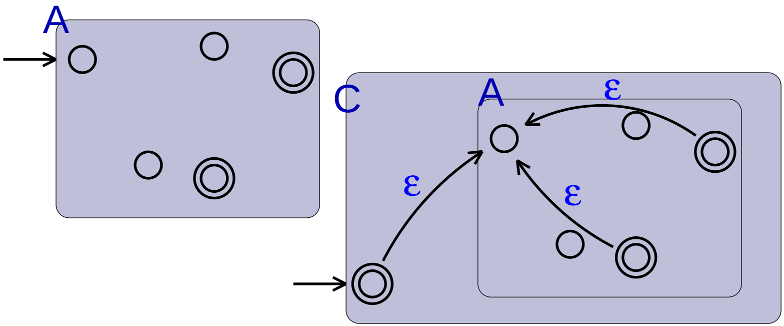
Closure under concatenation

$$C = (A \circ B) = \{xy \ : \ x \in A \text{ and } y \in B\}$$



Closure under star

$$C = A^* = \{x_1 x_2 x_3 \ldots x_k \ : \ k \geq 0 \text{ and each } x_i \in A\}$$



Note: we added a new initial state. (why?)

NFA, DFA equivalence

Theorem

a language L is recognized by a DFA if **and only if** *L is recognized by a NFA.*

Must prove **two** directions:
($\Rightarrow$) L is recognised by a DFA **implies** L is recognised by a NFA.
($\Leftarrow$) L is recognised by a NFA **implies** L is recognised by a DFA.

(as is often the case, one is easy, the other more difficult)

NFA, DFA equivalence

($\Rightarrow$) L is recognised by a DFA **implies** L is recognised by a NFA.

Proof.

A DFA is a NFA! $\square$

NFA, DFA equivalence

($\Leftarrow$) L is recognised by a NFA **implies** L is recognised by a DFA.

Proof.

we will build a DFA that *simulates* the NFA (and thus recognizes the same language).

- **alphabet will be the same**
- **what are the states of the DFA?**

continued...

□

NFA, DFA equivalence

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a NFA. We define a DFA M' as follows.

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

note that M' has the same alphabet as M.
The set of states of M' is $\mathcal{P}(Q)$ (the power set of Q; so a state of M' is a set of states of M).

Definition

Eps(S) denotes the set $q \in Q \; : \; q$ **is reachable from subset** S **by travelling along 0 or more** ϵ**-transitions**

The transition function of M' is the map $\delta' : \mathcal{P}(Q) \times \Sigma \rightarrow \mathcal{P}(Q)$ defined by

$$\delta'(P, a) = \bigcup_{q \in P} \mathrm{Eps}(\delta(q, a)).$$

For the initial state of M' we take $q_0' = \mathrm{Eps}(\{q_0\}) \in \mathcal{P}(Q)$.

NFA, DFA equivalence

Extending transition function δ to apply to words, not just letters:

$$w \in L(M) \iff \delta(q_0, w) \cap F \neq \emptyset \iff \delta'(\{q_0\}, w) \cap F \neq \emptyset.$$

Hence defining F' by

$$F' = \{P \in \mathcal{P}(Q) : P \cap F \neq \emptyset\}$$

we have now specified a deterministic automaton
$M' = (\mathcal{P}(Q), \Sigma, \delta', \{q_0\}, F')$ and

$$w \in L(M') \iff \delta'(\{q_0\}, w) \in F'$$

$$\iff \delta'(\{q_0\}, w) \cap F \neq \emptyset \iff w \in L(M).$$

We conclude $L(M) = L(M')$.

So far...

Theorem

the set of languages recognized by NFA is closed under union, concatenation, and star.

Theorem

a language L is recognized by a DFA if and only if L is recognized by a NFA.

Corollary

the set of languages recognized by DFA is closed under union, concatenation, and star.

More Closure Properties

DFA languages are also closed under complement and intersection (see exercises)

Regular expressions

R is a regular expression if R is

- a, for some $a \in \Sigma$
- ϵ, the empty string
- $\emptyset$, the empty set
- $(R_1 \cup R_2)$, where R_1 and R_2 are reg. exprs.
- $(R_1 \circ R_2)$, where R_1 and R_2 are reg. exprs.
- $(R_1)^*$, where R_1 is a regular expression

Examples:

$$(\mathtt{ab})^*\mathtt{ef} \qquad ((\mathtt{ab})^*\mathtt{e} \cup \mathtt{c})^*$$

Proving Kleene's Theorem

Theorem

a language L is recognized by a DFA **if and only if** *L is described by a regular expression.*

Must prove **two** directions:

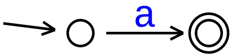
- ($\Rightarrow$) L is recognized by a DFA **implies** L is described by a regular expression
- ($\Leftarrow$) L is described by a regular expression **implies** L is recognized by a DFA.

We will do $\Leftarrow$ first, by converting regexps to NFA (and then applying previous result)

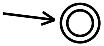
Regular expressions to DFA: *induction on expression structure*

First show there's an NFA for the basic regular expressions:

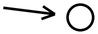
- a, for any a $\in \Sigma$



- ϵ, the empty string



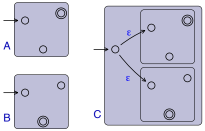
- $\emptyset$, the empty set



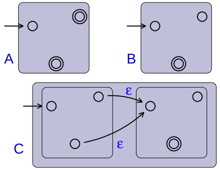
Regular expressions and DFA: Inductive step

Apply the closure properties:

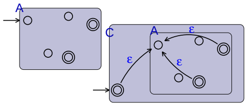
$(R_1 \cup R_2)$, where R_1 and R_2 are reg. exprs.



$(R_1 \circ R_2)$, where R_1 and R_2 are reg. exprs.



$(R_1)^*$, where R_1 is a reg. expression

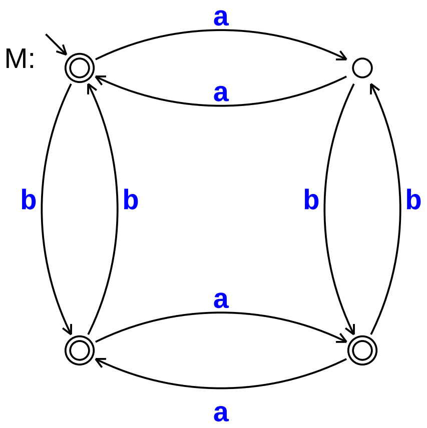


This is an *inductive construction* on the size of R (and the proof that it works would be by induction)

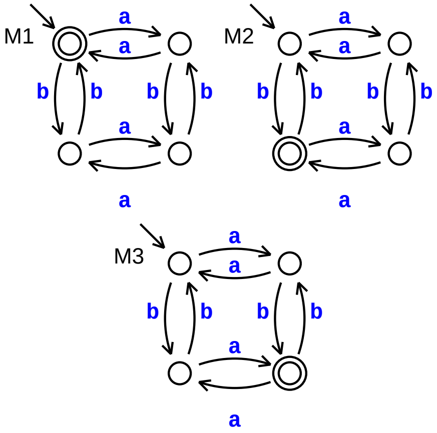
Other direction: DFA to regexp (via an example)

Convert this DFA *M* to equivalent r.e.

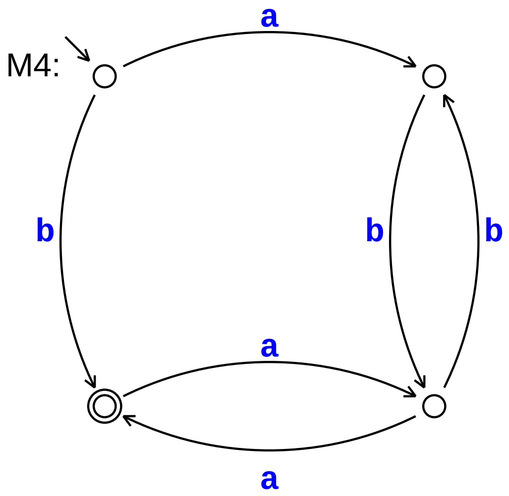
Language: even number of a's or an odd number of b's.



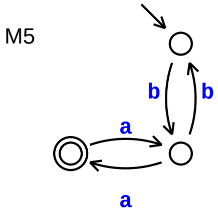
$$L(M) = L(M1) \cup L(M2) \cup L(M3)$$

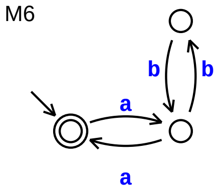


$L(M2) = L(M1)^* L(M4)$

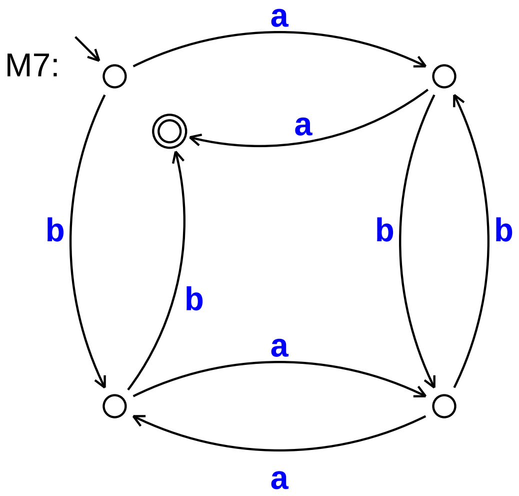


$L(M4) = \mathtt{a}L(M5) \cup \mathtt{b}L(M6)$

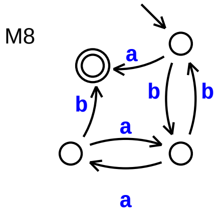


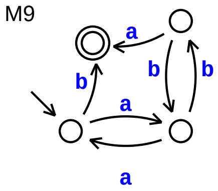


$L(M1) = L(M7)^*$



$L(M7) = \texttt{a}L(M8) \cup \texttt{b}L(M9)$





Algorithm: convert FA to regexp

Given a DFA $M = (Q, \Sigma, \delta, i, F)$.

Construct a regular expression for $L(M)$

Algorithm: convert FA to regexp

Given a DFA $M = (Q, \Sigma, \delta, i, F)$.

Construct a regular expression for $L(M)$

Recursive: express solution in terms of solutions to simpler instances of this problem.

Algorithm: convert FA to regexp

Given a DFA $M = (Q, \Sigma, \delta, i, F)$.

Construct a regular expression for $L(M)$

Recursive: express solution in terms of solutions to simpler instances of this problem.

Base case: M has no labelled transitions to an accepting state.
If $i \in F$ then $L(M) = \{\epsilon\}$
Else $L(M) = \emptyset$

The algorithm (continued)

Recursive case: there's a transition to accepting state

If $F = \{q_1, \ldots, q_k\}$ where $k > 1$
Then $L(M) = \cup_{j=1}^{k} L(M_j)$
where $M_j = (Q, \Sigma, \delta, i, \{q_j\})$

Evaluations of regular expressions for $L(M_j)$ are done recursively.

The algorithm (continued)

```
Else /* one accepting state */
    Let $F = \{t\}$ (i.e. $M = (Q, \Sigma, \delta, i, \{t\})$)
    If $t = i$
    Then $L(M) = L(M')^*$
        where $M' = (Q \cup \{t'\}, \Sigma, \delta', i, \{t'\})$
        and $\delta'(q, a) = t'$ whenever $\delta(q, a) = i$
            otherwise $\delta'(q, a) = \delta(q, a)$.
```

Regular expression for $L(M')$ is found recursively.

The algorithm (continued)

```
Else /* $F = \{t\}$, $t \neq i$ */
    If $M$ has labelled transitions to $i$
    Then $L(M) = L(M')^*L(M'')$
        where $M' = (Q, \Sigma, \delta, i, \{i\})$,
        $M''$ is like $M$ but without the labelled transitions to $i$
    Else /* no labelled transitions to $i$ */
        $L(M) = \cup_{a \in \Sigma} aL(M_a)$
        where $M_a$ is like $M$ but without transition
        from $i$ labelled by $a$ (don't include $M_a$
        if no such transition is in $M$)
        Initial state of $M_a$ is $\delta(i, a)$.
```

Regular expressions for $L(M')$, $L(M'')$, $L(M_a)$ etc are found recursively.

Justifying the DFA to r.e. algorithm

The algorithm expresses the language accepted by a given DFA M in terms of languages accepted by other DFAs - recursive
Want to verify that:

1. L(M) is the same as the language accepted by the combinations of DFAs constructed in the recursive calls
2. The algorithm terminates.

Justifying the DFA to r.e. algorithm

The algorithm expresses the language accepted by a given DFA M in terms of languages accepted by other DFAs - recursive
Want to verify that:

1. L(M) is the same as the language accepted by the combinations of DFAs constructed in the recursive calls
2. The algorithm terminates.

Regarding (1): We saw that in all the cases that arose the languages are the same

Justifying the DFA to r.e. algorithm

The algorithm expresses the language accepted by a given DFA M in terms of languages accepted by other DFAs - recursive
Want to verify that:

1. L(M) is the same as the language accepted by the combinations of DFAs constructed in the recursive calls
2. The algorithm terminates.

Regarding (1): We saw that in all the cases that arose the languages are the same
Regarding (2): Define "simpler" such that the algorithm decomposes a task into "simpler" tasks, until we reach base case (the "simplest" tasks where it's obvious what to do)

Regular expressions and DFA

- **Theorem:** (Kleene's theorem) a language L is recognized by a DFA iff L is described by a regular expr.
- Languages recognized by a DFA are called **regular languages**.
- Rephrasing what we know so far:
  - **regular languages** closed under 3 operations
  - NFA recognize exactly the **regular languages**
  - regular expressions describe exactly the **regular languages**

Limits on the power of FA

- Not all well-defined languages are accepted by DFAs (e.g., bit strings containing as many 1's as 0's; or syntactically-valid arithmetic expressions)
- To convince yourself, need to "prove a negative"... how to proceed?

Limits on the power of FA

- Not all well-defined languages are accepted by DFAs (e.g., bit strings containing as many 1's as 0's; or syntactically-valid arithmetic expressions)
- To convince yourself, need to "prove a negative"… how to proceed?

Intuition:

- FA can only remember finite amount of information. They cannot *count*
- languages that "entail counting" should be non-regular…

Limits on the power of FA

- Not all well-defined languages are accepted by DFAs (e.g., bit strings containing as many 1's as 0's; or syntactically-valid arithmetic expressions)
- To convince yourself, need to "prove a negative"... how to proceed?

Intuition:

- FA can only remember finite amount of information. They cannot *count*
- languages that "entail counting" should be non-regular...
- Intuition not enough:
  $\{w : w$ has an equal number of "01" and "10" substrings$\}$

$$= 0\Sigma^*0 \cup 1\Sigma^*1$$

Non-regular languages

A tool to establish non-regularity:

Pumping Lemma: Let L be a regular language. There exists an integer p ("pumping length") for which every $w \in L$ with $|w| \geq p$ can be written as $w = xyz$ such that

1. for every $i \geq 0$, $xy^iz \in L$, and
2. $|y| > 0$, and
3. $|xy| \leq p$.

Proof idea: if L has a DFA, then for every large enough word w in L, its path in the state machine goes through a small cycle. Copies of that cycle may be added to the accepting path (corresponds to inserting a sub-word in w). Details in textbook.