CISC 4090: Theory of Computation

Chapter 1
Regular Languages

Xiaolan Zhang, adapted from slides by Prof. Werschulz

Fordham University
Department of Computer and Information Sciences

Spring, 2014

Section 1.1: Finite Automata

What is a computer?

- Not a simple question to answer precisely
  - Computers are quite complicated
- We start with a computational model
  - Different models will have different features, and may match a real computer better in some ways, and worse in others
- Our first model is the *finite state machine* or *finite state automaton*

Finite automata

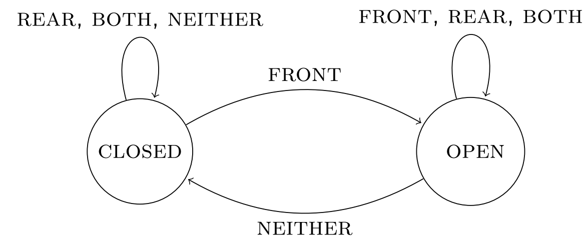
Models of computers with extremely limited memory

- Many simple computers have extremely limited memories and are (in fact) finite state machines.
- Can you name any? (Hint: several are in this building, but have nothing specifically to do with our department.)
  - Vending machine
  - Elevators
  - Thermostat
  - Automatic door at supermarket

Automatic door

- What is the desired behavior? Describe the actions and then list the states.
  - Person approaches, door should open
  - Door should stay open while person going through
  - Door should shut if no one near doorway
  - States are **Open** and **Closed**
- More details about automatic door
  - Components: front pad, door, rear pad
  - Describe behavior now:
    - Hint: action depends not only on what happens, but also on current state
    - If you walk through, door should stay open when you're on rear pad
    - But if door is closed and someone steps on rear pad, door does not open

Automatic door (cont'd)

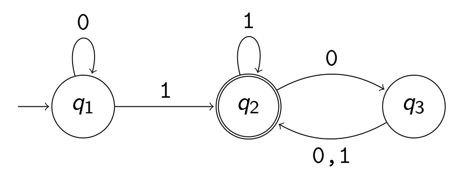


	NEITHER	FRONT	REAR	BOTH
CLOSED	CLOSED	OPEN	CLOSED	CLOSED
OPEN	CLOSED	OPEN	OPEN	OPEN

More on finite automata

- How may bits of data does this FSM store?
  - 1 bit: open or closed
- What about state information for elevators, thermostats, vending machines, etc.?
- FSM used in speech processing, special character recognition, compiler construction ...
- Have you implemented an FSM? When?
  - Network protocol for the game "Hangman"

A finite automaton M_1



Finite automaton M_1 with three states:

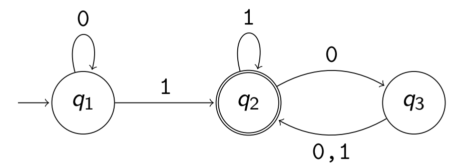
- We see the state diagram
  - Start state q_1
  - Accept state q_2 (double circle)
  - Several transitions
- A string like 1101 will be *accepted* if M_1 ends in accept state, and rejects otherwise. What will it do?
- Can you describe all strings that M_1 will accept?
  - All strings ending in 1, and
  - All strings having an even number of 0's following the final 1

Formal definition of finite state automata

A *finite (state) automaton* (FA) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$:

- Q is a finite set of *states*
- Σ is a finite set, called the *alphabet*
- $\delta\colon Q \times \Sigma \to Q$ is the *transition function*
- $q_0 \in Q$ is the *start state*
- $F \subseteq Q$ is the set of *accepting* (or *final*) *states*.

Describe M_1 using formal definition



$M_1 = (Q, \Sigma, \delta, q_0, F)$, where

- $Q = \{q_1, q_2, q_3\}$
- $\Sigma = \{0, 1\}$
- q_1 is the start state
- $F = \{q_2\}$ (only one accepting state)
- Transition function δ given by

δ	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

The language of an FA

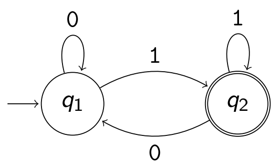
- If A is the set of all strings that a machine M accepts, then A is the *language* of M.
  - Write $L(M) = A$.
  - Also say that M *recognizes* or *accepts* A.
- A machine may accept many strings, but only one language.
- Convention: M *accepts strings* but *recognizes a language*.

What is the language of M_1?

- We write $L(M_1) = A$, i.e., M_1 recognizes A.
- What is A?
  - $A = \{\, w \in \{0, 1\}^* : \ldots \}$.
  - We have

$$A = \big\{\, w \in \{0,1\}^* : w \text{ contains at least one } 1 \text{ and an even number of 0's follow the last } 1 \,\big\}$$

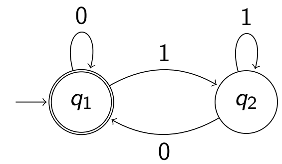
What is the language of M_2?



$M_2 = \{\{q_1, q_2\}, \{0, 1\}, \delta, q_1, \{q_2\}\}$ where

- I leave δ as an exercise.
- What is the language of M_2?
  - $L(M_2) = \{ \ w \in \{0, 1\}^* : \ldots \ \}$.
  - $L(M_2) = \{ \ w \in \{0, 1\}^* : w$ ends in a 1 $\}$.

What is the language of M_3?



- $M_3 = \{\{q_1, q_2\}, \{0, 1\}, \delta, q_1, \{q_1\}\}$ is M_2, but with accept state set $\{q_1\}$ instead of $\{q_2\}$.
- What is the language of M_3?
  - $L(M_3) = \{ \ w \in \{0, 1\}^* : \ldots \ \}$.
  - Guess $L(M_3) = \{ \ w \in \{0, 1\}^* : w$ ends in a 0 $\}$. Not quite right. Why?
  - $L(M_3) = \{ \ w \in \{0, 1\}^* : w = \varepsilon$ or w ends in a 0 $\}$.

What is the language of M_4?

- M_4 is a five-state automaton (Figure 1.12 on page 38).
- What does M_4 accept?
  - All strings that start and end with a or start and end with b.
  - More simply, $L(M_4)$ is all strings starting and ending with the same symbol.
    - Note that string of length 1 is okay.

Construct M_5 to do modular arithmetic

- Let $\Sigma = \{\langle \text{RESET} \rangle, 0, 1, 2\}$.
- Construct M_5 to accept a string iff the sum of each input symbol is a multiple of 3, and $\langle \text{RESET} \rangle$ sets the sum back to 0.

Now generalize M_5

- Generalize M_5 to accept if sum of symbols is a multiple of i instead of 3.

  $$M = \{\{q_0, q_1, q_2, \ldots, q_{i-1}\}, \{0, 1, 2, \langle \text{RESET} \rangle\}, \delta_i, q_0, \{q_0\}\},$$

  where
  - $\delta_i(q_j, 0) = q_j$.
  - $\delta_i(q_j, 1) = q_k$, where $k = j + 1 \bmod i$.
  - $\delta_i(q_j, 2) = q_k$, where $k = j + 2 \bmod i$.
  - $\delta_i(q_j, \langle \text{RESET} \rangle) = q_0$.
- Note: As long as i is finite, we are okay and only need finite memory (number of states).
- Could you generalize to $\Sigma = \{0,1, 2, \ldots, k\}$?

Formal definition of acceptance

Let $M = (Q, \Sigma, \delta, Q_0, F)$ be an FA and let $w = w_1 w_2 \ldots w_n \in \Sigma^*$. We say that M *accepts* w if there exists a sequence $r_0, r_1, \ldots, r_n \in Q$ of states such that

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

Regular languages

A language L is *regular* if it is recognized by some finite automaton.

- That is, there is a finite automaton M such that $L(M) = A$, i.e., M accepts all of the strings in the language, and rejects all strings *not* in the language.
- Why should you expect proofs by construction coming up in your next homework?

Designing automata

- You will need to design an FA that accept a given language L.
- Strategies:
  - Determine what you need to remember (The states).
    - How many states to determine even/odd number of 1's in an input?
    - What does each state represent?
  - Set the start and finish states, based on what each state represents.
  - Assign the transitions.
  - Check your solution: it should accept every $w \in L$, and it should not accept any $w \notin L$.
  - Be careful about ε.

You try designing FA

- Design an FA to accept the language of binary strings having an odd number of 1's (page 43).
- Design an FA to accept all strings containing the substring 001 (page 44).
  - What do you need to remember?
- Design an FA to accept strings containing the substring abab.

Regular operations

Let A and B be languages. We define three *regular operations*:

- *Union*: $A \cup B = \{ x : x \in A \text{ or } x \in B \}$.
- *Concatenation*: $A \cdot B = \{ xy : x \in A \text{ and } y \in B \}$.
- Kleene *star*: $A^* = \{ x_1 x_2 \ldots x_k : k \geq 0 \text{ and each } x_i \in A \}$.
  - Kleene star is a unary operator on a single language.
  - A^* consists of (possibly empty!) concatenations of strings from A.

Examples of regular operations

Let $A = \{\texttt{good}, \texttt{bad}\}$ and $B = \{\texttt{boy}, \texttt{girl}\}$. What are the following?

- $A \cup B = \{\texttt{good}, \texttt{bad}, \texttt{boy}, \texttt{girl}\}$.
- $A \cdot B = \{\texttt{goodboy}, \texttt{goodgirl}, \texttt{badboy}, \texttt{badgirl}\}$.
- $A^* =$ $\{\varepsilon, \texttt{good}, \texttt{bad}, \texttt{goodgood}, \texttt{goodbad}, \texttt{badgood}, \texttt{badbad}, \ldots\}$.

Closure

- A set of objects is *closed* under an operation if applying that operations to members of that set always results in a member of that set.
- The natural numbers $\mathbb{N} = \{1, 2, \ldots\}$ are closed under addition and multiplication, but not subtraction or division.

Closure for regular languages

- Regular languages are closed under the three regular operations we just introduced (union, concatenation, star).
- Can you look ahead to see why we care?
- We can build FA to recognize regular expressions!

Closure of union

Theorem 1.25: The class of regular languages is closed under the union operation. That is, if A_1 and A_2 are regular languages, then so is $A_1 \cup A_2$.
How can we prove this?

- Suppose that M_1 accepts A_1 and M_2 accepts A_2.
- Construct M_3 using M_1 and M_2 to accept $A_1 \cup A_2$.
- We need to simulate M_1 and M_2 running in parallel, and stop if either reaches an accepting state.
  - This last part is feasible, since we can have multiple accepting states.
  - You need to remember where you are in both machines.

Closure of union (cont'd)

- You need to generate a state to represent the state you are in with M_1 and M_2.
- Let $M_i = (Q_i, \Sigma, \delta_i, q_i, F_i)$ for $i \in \{1, 2\}$.
- Build $M = (Q, \Sigma, \delta, q, F)$ as follows:
  - $Q = Q_1 \times Q_2 = \{\,(r_1, r_2) : r_1 \in Q_1 \text{ and } r_2 \in Q_2\,\}$.
  - Σ is unchanged. (Note that if M_i used Σ_i for $i \in \{1, 2\}$, we could have chosen $\Sigma = \Sigma_1 \cup \Sigma_2$.)
  - $q_0 = (q_1, q_2)$.
  - $F = \{\,(r_1, r_2) : r_1 \in F_1 \text{ or } r_2 \in F_2\,\}$.
  - $\delta\big((r_1, r_2), a\big) = \big(\delta(r_1, a), \delta(r_2, a)\big)$.

Closure of concatenation

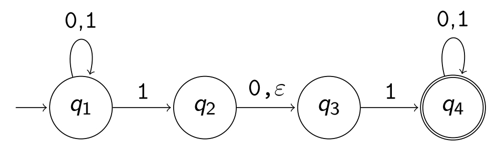
Theorem 1.26: The class of regular languages is closed under the concatenation operator. That is, if A_1 and A_2 are regular languages, then so is $A_1 \cdot A_2$.
Can you see how to do this simply?
Not trivial, since cannot just concatenate M_1 and M_2, where the finish states of M_1 becoming the start state of M_2.

- Because we do not accept a string as soon as it enters the finish state, we wait until string is done, so it can leave and come back.
- Thus we do not know when to start using M_2.
- The proof is easy if we use *nondeterministic* FA.

Section 1.2: Nondeterminism

Nondeterminism

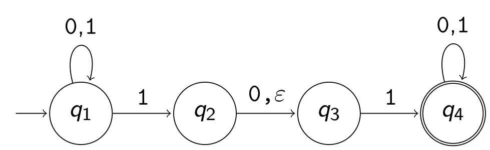
- So far, our FA have been *deterministic*: the current state and the input symbol determine the next state.
- In a *nondeterministic* machine, several choices may exist.
- DFA's have one transition arrow per input symbol
- NFA's . . .
  - have zero *or more* transitions for each input symbol, and
  - may have an ε-transition.



How does an NFA compute?

- When there is a choice, all paths are followed.
  - Think of it as cloning a process and continuing.
  - If there is no arrow, the path terminates and the clone dies (it does not accept if at an accept state when this happens).
  - An NFA may have the empty string cause a transition.
  - The NFA accepts any path is in the the accept state.
  - Can also be modeled as a tree of possibilities.
- An alternative way of thinking about this:
  - At each choice, you make one guess of which way to go.
  - You always magically guess the right way to go.

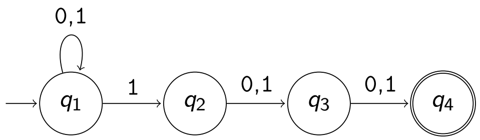
Try computing this!



- Try out 010110.
  Is it accepted? Yes!
- What is the language?
  Strings containing either 101 or 11 as a substring.

Construct an NFA

- Construct an NFA that accepts all strings over $\{0, 1\}$, with a 1 in the third position from the end.
- **Hint:** The NFA stays in the start state until it guesses that it is three places from the end.
- Solution?



Can we generate a DFA for this?

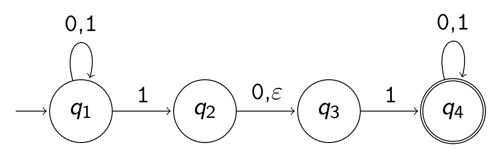
Yes, but it is more complicated and has eight states.

- See book, Figure 1.32, page 51.
- Each state represents the last three symbols seen, where we assume we start with 000.
- What is the transition from 010?
  - On a 1, we go to 101.
  - On a 0, we go to 100.

Formal definition of nondeterministic finite automata

- Similar to DFA, except transition function must work for ε, in addition to Σ, and a "state" is a *set* of states, rather than a single state.
- A nondeterministic finite automaton (NDFA) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$:
  - Q is a finite set of *states*
  - Σ is a finite set, called the *alphabet*
  - $\delta\colon Q \times \Sigma_\varepsilon \to \mathscr{P}(Q)$ is the *transition function*. (Here, $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$.)
  - $q_0 \in Q$ is the *start state*
  - $F \subseteq Q$ is the set of *accepting* (or *final*) *states*.

Example of formal definition of NFA



NFA $N_1 = (Q, \Sigma, \delta, q_1, F)$ where

- $Q = \{q_1, q_2, q_3, q_4\}$,
- $\Sigma = \{0, 1\}$,
- q_1 is the start state,
- $F = \{q_4\}$,

δ	0	1	ε
q_1	$\{q_1\}$	$\{q_1, q_2\}$	$\emptyset$
q_2	$\{q_3\}$	$\emptyset$	$\{q_3\}$
q_3	$\emptyset$	$\{q_4\}$	$\emptyset$
q_4	$\{q_4\}$	$\{q_4\}$	$\emptyset$

Equivalence of NFAs and DFAs

NFAs and DFAs recognize the same class of languages.

- We say two machines are *equivalent* if they recognize the same language.
- NFAs have no more power than DFAs:
  - with respect to what can be expressed.
  - But NFAs may make it much easier to describe a given language.
- Every NFA has an equivalent DFA.

Proof of equivalence of NFA and DFA

Proof idea:

- Need to *simulate* an NFA with a DFA.
- With NFAs, given an input, we follow all possible branches and keep a finger on the state for each.
- That is what we need to track: the states we would be in for each branch.
- If the NFA has k states, then it has 2^k possible subsets.
  - Each subset corresponds to one of the possibilities that the DFA needs to remember.
  - The DFA will have 2^k states.

Proof by construction

- Let $N = (Q, \Sigma, \delta, q_0, F)$ be an NFA recognizing language A.
- Construct a DFA $M = (Q', \Sigma, \delta', q_0', F')$.
  - Let's do the easy steps first (skip δ' for now).
  - $Q' = \mathscr{P}(Q)$
  - $q_0' = \{q_0\}$.
  - $F' = \{ R \in Q' : R \text{ contains an accept state of } N \}$.
  - Transition function?
    - The state $R \in M$ corresponds to a set of states in N.
    - When M reads symbol a in state R, it shows where a takes each state.
    - $\delta'(R, a) = \bigcup_{r \in R} \delta(r, a)$.
  - I ignore ε, but taking that into account does not fundamentally change the proof; we just need to keep track of more states.

Example: Convert an NFA to a DFA

See Example 1.41 on page 57. For now, don't look at solution DFA!

- The NFA has 3 states: $Q = \{1, 2, 3\}$. What are the states in the DFA?
  $\{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$.
- What are the start states of the DFA?
  - The start states of the NFA, including those reachable by ε-transitions
  - $\{1, 3\}$ (We include 3 because if we we start in 1, we can immediately move to 3 via an ε-transition.)
- What are the accept states?
  $\{\{1\}, \{1, 2\}, \{1, 3\}, \{1, 2, 3\}\}$.

Example: Convert an NFA to a DFA (cont'd)

Now, let's work on some of those transitions.

- Let's look at state 2 in NFA and complete the transitions for state 2 in the DFA.
  - Where do we go from state 2 on $\mathtt{a}$ or $\mathtt{b}$?
    - On $\mathtt{a}$ go to states 2 and 3.
    - On $\mathtt{b}$, go to state 3.
  - So what state does $\{2\}$ in DFA go to for a and b?
    - On $\mathtt{a}$ go to state $\{2, 3\}$.
    - On $\mathtt{b}$, go to state $\{3\}$.
- Now let's do state $\{3\}$.
  - On $\mathtt{a}$ go to $\{1, 3\}$.
    Why? First go to 1, then ε-transition back to 3.
  - On $\mathtt{b}$, go to $\emptyset$.
- Now check DFA, Figure 1.43, on page 58.

Any questions? Could you do this on a homework? an exam?

Closure under regular operations

- We started this before and did it only for union.
  - Union much simpler using NFA.
- Concatenation and star much easier using NFA.
- Since DFAs equivalent to NFAs, suffices to just use NFAs
- In all cases, fewer states to track, because we can always "guess" correctly.

Why do we care about closure?

We need to look ahead:

- A regular language is what a DFA/NFA accepts.
- We are now introducing regular operators and then will generate regular expressions from them (Section 1.3).
- We will want to show that the language of regular expressions is equivalent to the language accepted by NFAs/DFAs (i.e., a regular language)
- How do we show this?
  - Basic terms in regular expression can generated by a FA.
  - We can implement each operator using a FA and the combination is still able to be represented using a FA

Closure under union

- Given two regular languages A_1 and A_2, recognized by two NFAs N_1 and N_2, construct NFA N to recognize $A_1 \cup A_2$.
- How do we construct N? Think!
  - Start by writing down N_1 and N_2. Now what?
  - Add a new start state and then have it take ε-branches to the start states of N_1 and N_2.

Closure under concatenation

- Given two regular languages A_1 and A_2 recognized by two NFAs N_1 and N_2, construct NFA N to recognize $A_1 \cdot A_2$.
- How do we do this?
  - The complication is that we did not know when to switch from handling A_1 to A_2, since can loop thru an accept state.
  - Solution with NFA:
    - Connect every accept state in N_1 to every start state in N_2 using an ε-transition.
    - Don't remove transitions from accept state in N_1 back to N_1.

Closure under concatenation (cont'd)

- Given:
  - $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognizing A_1, and
  - $N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ recognizing A_2.
- Construct $N = (Q_1 \cup Q_2, \Sigma, \delta, q_1, F)$ recognizing $A_1 \cdot A_2$. Transition function

$$\delta \colon (Q_1 \cup Q_2) \times \Sigma_\varepsilon \to \mathscr{P}(Q_1 \cup Q_2)$$

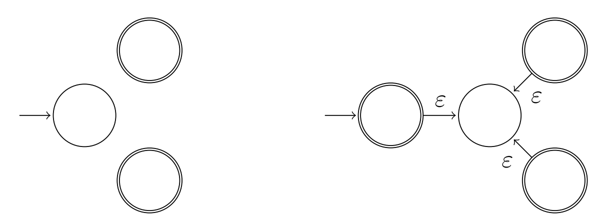
given as

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in Q_1 \text{ and } q \notin F_1 \\ \delta_1(q,a) & q \in F_1 \text{ and } a \neq \varepsilon \\ \delta_1(q,a) \cup \{q_2\} & q \in Q_1 \text{ and } a = \varepsilon \\ \delta_2(q,a) & q \in Q_2 \end{cases}$$

Closure under star

- We have a regular language A_1 and want to prove that A_1^* is also regular.
  Recall: $(\mathtt{ab})^* = \{\varepsilon, \mathtt{ab}, \mathtt{abab}, \mathtt{ababab}, \ldots\}$.
- Proof by construction:
  - Take the NFA N_1 that recognizes A_1 and construct from it an NFA N that recognizes A_1^*.
  - How do we do this?
    - Add new ε-transition from accept states to start state.
    - Then make the start state an additional accept state, so that ε is accepted.
    - This almost works, but not quite.
    - Problem? May have transition from intermediate state to start state; should not accept this.
    - Solution? Add a new start state with an ε-transition to the original start state, and have ε-transitions from accept states to old start state.

Closure under star (cont'd)



Section 1.3: Regular expressions

Regular expressions

- Based on the regular operators.
- Examples:
  - $(0 \cup 1)0^*$
    - A 0 or 1, followed by any number of 0's.
    - Concatenation operator implied.
  - What does $(0 \cup 1)^*$ mean?
    - Al possible strings of 0 and 1.
      Not 0^* or 1^*, so does not require we commit to 0 or 1 before applying * operator.
    - Assuming $\Sigma = \{0, 1\}$, equivalent to Σ^*.

Definition of regular expression

- Let Σ be an alphabet. R is a *regular expression* over Σ if R is:
  - a, for some $a \in \Sigma$
  - ε
  - $\emptyset$
  - $R_1 \cup R_2$, where R_1 and R_2 are regular expressions.
  - $R_1 \cap R_2$, where R_1 and R_2 are regular expressions.
  - R_1^*, where R_1 is a regular expression.
- Note:
  - This is a recursive definition, common to computer science. Since R_1 and R_2 are simpler than R, no issue of infinite recursion.
  - $\emptyset$ is a language containing no strings, and ε is the empty string.

Examples of regular expressions

- $0^*10^* = \{\, w \in \{0, 1\}^* : w \text{ contains a single } 1 \,\}$.
- $\Sigma^*1\Sigma^* = \{\, w \in \{0, 1\}^* : w \text{ contains at least one } 1 \,\}$.
- $01 \cup 10 = \{01, 10\}$.
- $(0 \cup \varepsilon)(1 \cup \varepsilon) = \{\varepsilon, 0, 1, 01\}$.

Equivalence of regular expressions and finite automata

Theorem: A language is regular if and only if some regular expression describes it.

- This has two directions, so we need to prove:
  - If a language is described by a regular expression, then it is regular.
  - If a language is regular, then it is described by a regular expression.
- We'll do both directions.

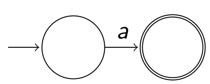
Proof: Regular expression $\implies$ regular language

- Proof idea: Given a regular expression R describing a language L, we should
  - Show that some FA recognizes it.
  - Use NFA, since may be easier (and it's equivalent to DFA).
- How do we do this?
  - We will use definition of a regular expression, and show that we can build an FA covering each step.
  - We will do quickly with two parts:
    - Steps 1, 2 and 3 of definition (handle a, ε, and $\emptyset$).
    - Steps 4, 5, and 6 of definition (handle union, concatenation, and star).

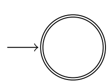
Proof (cont'd)

Steps 1–3 are fairly simple:

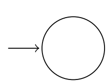
- a, for some $a \in \Sigma$. The FA is



- ε. The FA is



- $\emptyset$. The FA is



Proof (cont'd)

- For steps 4–6 (union, concatenation, and star), we use the proofs we used earlier, when we established that FA are closed under union, concatenation, and star.
- So we are done with the proof in one direction.
- So let's try an example.

Example: Regular expression $\implies$ regular language

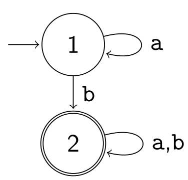
- Convert $(\mathtt{ab} \cup \mathtt{a})^*$ to an NFA.
  See Example 1.56 on page 68.
- Let's outline what we need to do:
  - Handle $\mathtt{a}$.
  - Handle $\mathtt{ab}$.
  - Handle $\mathtt{ab} \cup \mathtt{a}$.
  - Handle $(\mathtt{ab} \cup \mathtt{a})^*$.
- The book has states for ε-transitions. They seem unnecessary and may confuse you. In fact, they *are* unnecessary in this case.
- Now we need to do the proof in the other direction.

Proof: Regular language $\implies$ regular expression

- A regular language is described by a DFA.
- Need to show that can convert an DFA to a regular expression.
- The book goes through several pages (Lemma 1.60, pp. 69–74) that don't really add much insight.
  - You can skip this. For the most part, if you understand the ideas for going in the previous direction, you also understand this direction.
  - But you should be able to handle an example . . . .

Example: DFA $\implies$ regular expression

Find the regular expression that is equivalent to the DFA



Answer is $\mathtt{a}^*\mathtt{b}(\mathtt{a} \cup \mathtt{b})^*$.

Section 1.4: Non-regular languages

Non-regular languages

- Do you think every language is regular? That would mean that every language can be described by a FA.
- What might make a language non-regular? Think about main property of a finite automaton: finite memory!
- So a language requiring infinite memory cannot be regular!

Some example questions

- Are the following languages regular?
  - $L_1 = \{ w : w$ has an equal number of 0's and 1's$\}$.
  - $L_2 = \{ w : w$ has at least 100 1's$\}$.
  - $L_3 = \{ w : w$ is of the form $0^n 1^n$ for some $n \geq 0 \}$.
  - First, write out some of the elements in each, to ensure you have the terminology down.
    - $L_1 = \{\varepsilon, 01, 10, 1100, 0011, 0101, 1010, 0110, \dots\}$.
    - $L_2 = \{100\ 1\text{'s}, 0\ 100\ 1\text{'s}, 1\ 100\ 1\text{'s}, \dots\}$.
    - $L_3 = \{\varepsilon, 01, 0011, 000111, \dots\}$.

Answers

- L_1 and L_3 are not regular languages; they require infinite memory.
- L_2 certainly is regular.

We will only study *infinite* regular languages.

What is wrong with this?

- Question 1.36 from the book asks:

  Let $B_n = \{ \mathtt{a}^k : k$ is a multiple of $n\}$.
  Show that B_n is regular.

- How is this regular? How is this question different from the ones before?
- Each language B_n has a *specific* value of n, so n is not a free variable (unlike the previous examples).
- Although k is a free variable, the number of states is bounded by n, and not k.

More on regular languages

- Regular languages can be infinite, but must be described using finitely-many states.
- Thus there are restrictions on the *structure of* regular languages.
- For an FA to generate an infinite set of strings, what must there be between some states? A loop.
- This leads to the (in)famous *pumping lemma*.

Pumping Lemma for regular languages

- The Pumping Lemma states that all regular languages have a special *pumping property*.
- If a language does not have the pumping property, then it is not regular.
  - So one can use the Pumping Lemma to prove that a given language is not regular.
  - **Note:** Pumping Lemma is an implication, not an equivalence. Hence, there are non-regular languages that have the pumping property.

The Pumping Lemma

- *Let L be a regular language. There is a positive integer p such that any $s \in L$ with $|s| > p$ can be "pumped".*
- (p is the *pumping length* of L.)
- This means that every string $s \in L$ contains a substring that can repeated any number of times (via a loop).
- The statement "s can be pumped" means that we can write $s = xyz$, where
  1. $xy^iz \in L$ for all $i \geq 0$.
  2. $|y| > 0$,
  3. $|xy| \leq p$.

Pumping Lemma explained

- Condition 1: $xy^iz \in L$ for all $i \geq 0$.
  This simply says that there is a loop.
- Condition 2: $|y| > 0$.
  Without this condition, then there really would be no loop.
- Condition 3: $|xy| \leq p$.
  We don't allow more states than the pumping length, since we want to bound the amount of memory.
- All together, the conditions allow either x or z to be ε, but not both.
  The loop need not be in the middle (which would be limiting).

Pumping Lemma: Proof idea

- Let p = number of states in the FA.
- Let $s \in L$ with $|s| > p$.
- Consider the states that FA goes through for s.
- Since there are only p states and $|s| > p$, one state must be repeated (via pigeonhole principle).
- So, there is a loop.

Pumping Lemma: Example 1

- Let $B = \{ 0^n 1^n : n \geq 0 \}$ (Example 1.73).
  Show that B is not regular.
- Use proof by contradiction.
  Assume that B is regular.
  Now pick a string that will cause a problem.
- Try $0^p 1^p$.
- Since B is regular, we can write $0^p 1^p = xyz$ as in statement of Pumping Lemma.
  Look at y:
  - If y all 0's or all 1's, then $xyyz \notin B$. (Count is wrong.)
  - If y a mixture of 0's and 1's, then 0's and 1's not completely separated in $xyyz$, and so $xyyz \notin B$.

  So $0^p 1^p$ can't be pumped, and thus B is not regular.

Pumping Lemma: Example 2

- Let $C = \{ w \in \{0, 1\}^* : w \text{ has equal number of 0's and 1's} \}$ (Example 1.74).
  Show that C is not regular.
  - Use proof by contradiction.
    Assume that C is regular.
    Now pick a problematic string.
  - Let's try $0^p 1^p$ again.
    - If we pick $x = z = \varepsilon$ and $y = 0^p 1^p$, can we pump it and have pumped string $xy^i z \in C$? Yes! Each pumping adds one 0 and one 1. But this choice breaks condition $|xy| \leq p$.
    - Suppose we choose x, y, z such that $|xy| \leq p$ and $|y| > 0$. Since $|xy| \leq p$, y consists only of 0's. Hence $xyyz \notin C$ (too many zeros).
- Shorter proof: If C were regular, then $B = C \cap 0^* 1^*$ would also be regular. This contradicts previous example!

Common-sense interpretation

- FA can only use finite memory. If L has infinitely many strings, they must be handled by the loop.
- If there are two parts that can generate infinite sequences, we must find a way to link them in the loop.
  - If not, L is not regular.
  - Examples:
    - $0^n 1^n$
    - Equally many 0s and 1s.

Pumping Lemma: Example 3

- Let $F = \{ ww : w \in \{0,1\}^* \}$ (Example 1.75).
- $F = \{\varepsilon, 00, 11, 0000, 0101, 1010, 1111, \dots\}$.
- Use proof by contradiction. Pick problematic $s \in F$.
- Try $s = 0^p1^p1$. Let $s = xyz$ be a splitting as per the Pumping Lemma.
  - Since $|xy| \leq p$, y must be all 0's.
  - So $xyyz \notin F$, since 0's separated by 1 must be equal.

Pumping Lemma: Example 4

- Let $D = \{ 1^{n^2} : n \geq 0 \}$.
- $D = \{\varepsilon, 1, 1111, 111111111, \dots\}$.
- Choose 1^{p^2}.
  - Assume we have $xyz \in D$ as per Pumping Lemma.
  - What about $xyyz$? The number of 1's differs from those in xyz by $|y|$.
    - Since $|xy| \leq p$, then $|y| \leq p$.
    - So if $|xyz| \leq p^2$, then $|xyyz| \leq p^2 + p$.
    - But $p^2 + p < p^2 + 2p + 1 = (p+1)^2$.
    - Moreover, $|y| > 0$, and so $|xyyz| > p^2$.
    - So $|xyyz|$ lies between two consecutive perfect squares, and hence $xyyz \notin D$.

Pumping Lemma: Example 5

- Let $E = \{0^i1^j : i > j\}$.
- Assume E is regular and let $s = 0^{p+1}1^p$.
- Decompose $s = xyz$ as per statement of Pumping Lemma.
- By condition 3, y must be all 0's.
  - What can we say about $xyyz$?
    Adding the extra y increases number of 0's, which appears to be okay, since $i > j$ is okay.
  - But we can pump down. What about $xy^0z = xz$?
    Since s has one more 0 than 1, removing at least one 0 leads to a contradiction. So not regular.

What you must be able to do

- You should be able to handle examples like 1–3.
- Example 5 is not really any more difficult—just one more thing to think about.
- Example 4 was tough, so I won't expect everyone to get an example like that.
- You need to be able to handle the easy examples.
  On an exam, I would probably give you several problems that are minor variants of these examples.
- Try to reason about the problem using "common sense" and then use that to drive your proof.
- The homework problems will give you more practice.