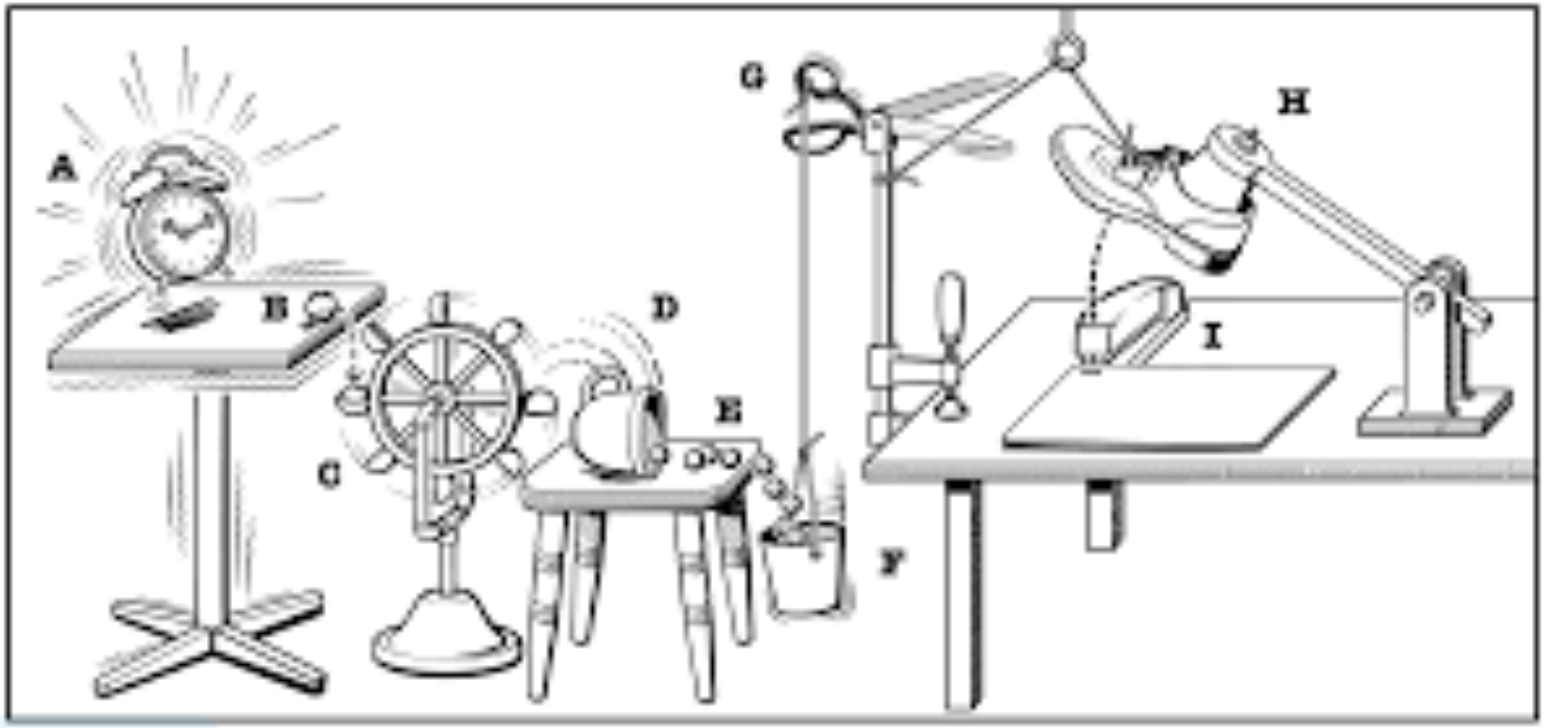
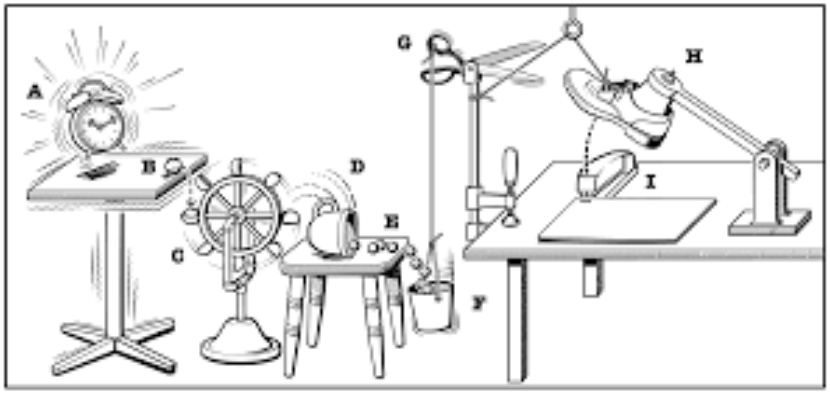
Welcome

INHN0013
Information Theory & Theory of Computation

Stephen Kobourov
Prof. of Computer Science



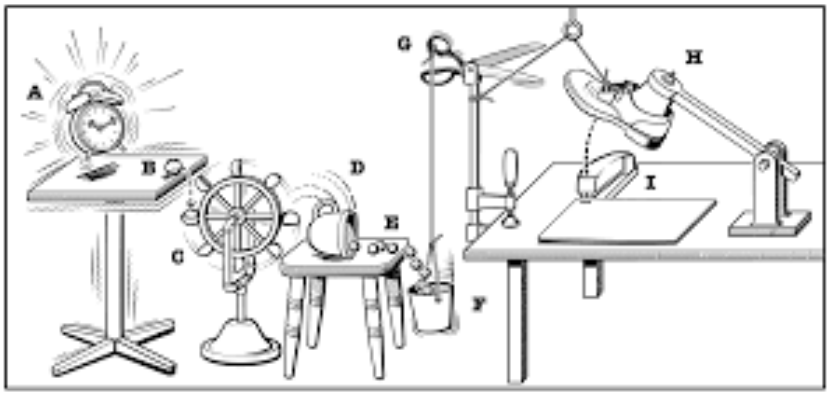
Theory of Computation



Computability:

- what problems cannot be solved?
- for solvable problems, how powerful a machine do we need?
- what computation models are there?

Theory of Computation



Computability:

- what problems cannot be solved?
- for solvable problems, how powerful a machine do we need?
- what computation models are there?

Complexity:

- the big divide: the classes P and NP ($1M Millennium Prize)
- NP-completeness and reductions
- approximation/randomized algorithms

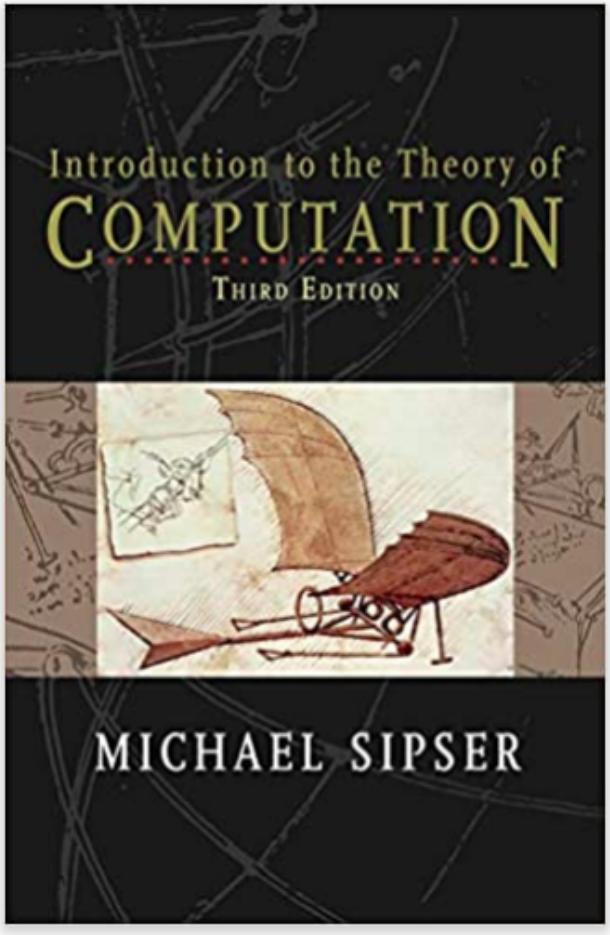
Famous and Important Results Everyone Should Know

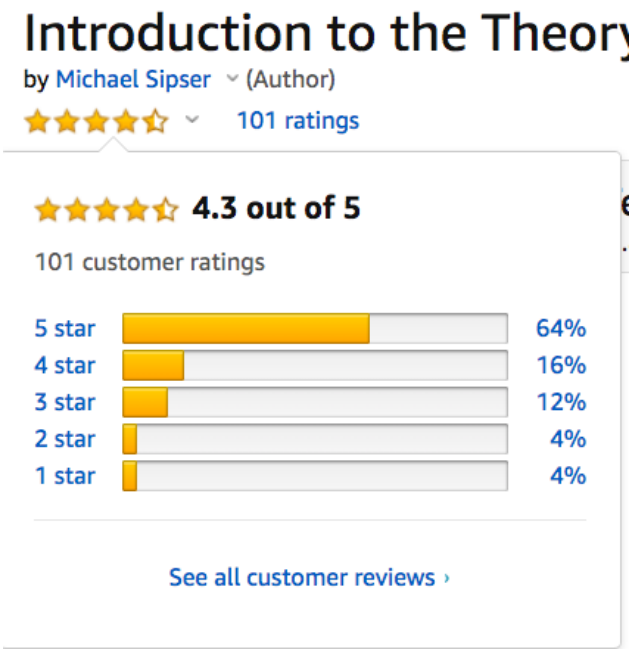
- Chomsky's Language Hierarchy
- The Church-Turing thesis
- Der Entscheidungsproblem
- Gödel's Incompleteness Theorems
- The Cook-Levin Theorem
- Cantor's Infinities

Automata, Grammars, and Languages

- This course introduces the fundamental models of computation used throughout computer science: finite automata, pushdown automata, and Turing machines.
- We will investigate the limitations of computation: what can and what cannot be computed, and what can be computed using a realistic amount of resources.
- We will study computational complexity, complexity classes, relationships among complexity classes, NP-hardness and NP-completeness.
- The course stresses methods for formal reasoning and modeling of general computation. This is a theory class and there will be no programming.
- The key techniques to be learned are those of simulation of one computational model by another, reduction of one problem to another, and methods for classifying problem complexity.

Textbook and Other Reading





Edition: 2nd or 3rd

e-book available through the TUM library

Course Syllabus

- Chapter 0 (Review): Mathematical Notation and Terminology, Definitions, Theorems, and Proofs, Types of Proofs
- Chapter 1. Regular Languages: 1.1: DFAs, 1.2: NFAs, 1.3: Regular Expressions, 1.4: Non-regular languages
- Chapter 2. Context-Free (CF) Languages : 2.1: CF Grammars, 2.2: Pushdown Automata, 2.3: Non-CF Languages
- Chapter 3. Church-Turing Thesis: 3.1: Turing Machines, 3.2: Turing Machine Variants, 3.3: The Definition of Algorithm
- Chapter 4. Decidability: 4.1: Decidable Languages, 4.2: Undecidability and The Halting Problem
- Chapter 5. Reducibility: 5.1: Undecidable Problems, 5.2: More Undecidable Problems
- Chapter 7. Time Complexity: 7.1: Measuring Complexity, 7.2, 7.3: The P and NP Classes, 7.4: NP-completeness Chapter
- 8: Space Complexity: 8.1: Savitch's Theorem, 8.2: PSPACE

Course Participation and Learning Environment

- Actively participating in the course, attending lectures and exercises, asking and answering questions on moodle are vital to the learning process; absences will very likely affect a student's performance.
- TUM is committed to providing and maintaining a supportive educational environment for all. We strive to be welcoming and inclusive, respect privacy and confidentiality, behave respectfully and courteously, and practice intellectual honesty. Disruptive behaviors (such as physical or emotional harassment, dismissive attitudes, and abuse of department resources) will not be tolerated.
- To foster a positive learning environment, students and instructors have a shared responsibility. To that end, our focus is on the tasks at hand and not on extraneous activities (e.g., texting, chatting, making phone calls, web surfing, etc.).

Grading

- The five exams during the semester are optional, but highly recommended.
- Each 60-minute exam will show you the type of questions likely to appear on the final exam and also how such questions are graded.
- The exams will include a subset of assigned exercise problems, to give you time to prepare for deeper exam questions than multiple choice and true/false ones.
- By achieving at least 50% of the total number of points for each exam, you will obtain a grade bonus of 0.3 that applies if you pass the final exam.
- To obtain a passing grade on the final exam, you also need at least 50% of the total number of points.
- Moodle has more and more detailed information.
- Use moodle for all questions: from lecture material, to exercises, exams, dates, etc.

Class Format

- Lectures, Mondays, **10:00**-11:30
- Lectures, Wednesdays, **9:00**-10:15
- Exercises, 90 minutes every week:
  - 2*x* Thursday 14:15-15:45
  - 2*x* Thursday 16:15-17:45
  - 1*x* Friday 14:15-15:45
- Five optional graded exams during Wednesday lectures
- Comprehensive final exam
- Moodle page: more info, lecture notes, exercise sheets, etc.
- Reading: Chapters 0 and 1

Teaching Team

Aaron Buengener

Attila Berczik

Danny Lee

Timo Brand

Automata, Grammars, and Languages

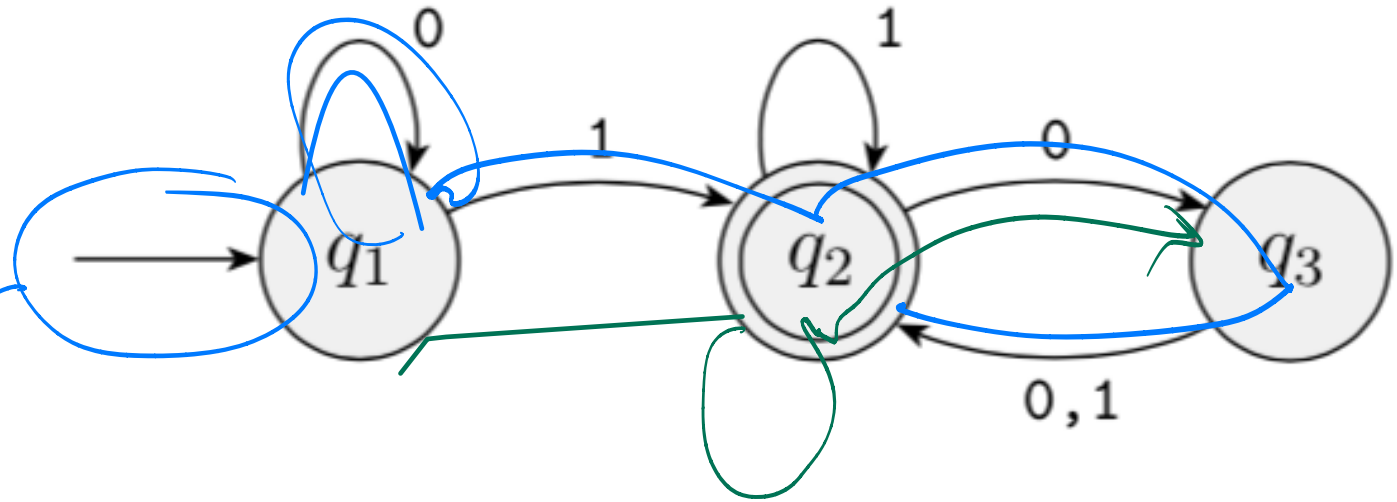
Recall that Theory of Computation deals with two majors aspects: computability and complexity

- Computability: What are the fundamental capabilities and limitations of computers?
- Complexity: What makes some problems computationally hard and others easy?

Automata, Grammars, and Languages

- Automata (machines) come in different strengths, depending on their ability to remember stuff (memory) and on the way to look up stuff (access)
- Automata can be seen as processing strings over some alphabet (e.g., binary) and either accept or reject each valid string
- The language of an automaton is the set of strings it accepts
- A grammar is made of rules that generate all strings in a given language
- For now computation simply means determining whether a given string is in the language or not (decision problem)

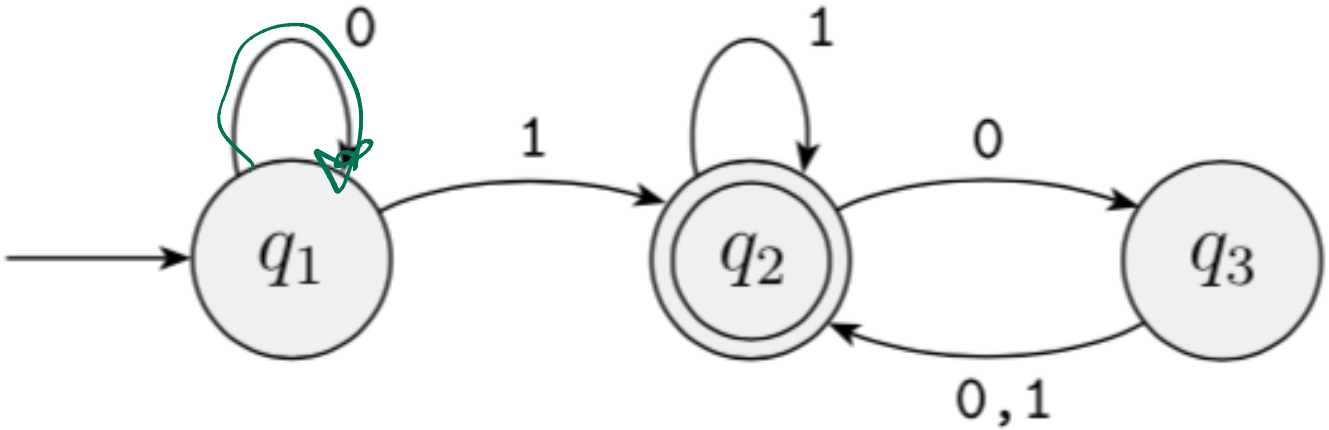
The Simplest Computational Model: Finite Automata



A finite automaton is a 5-tuple $(Q, \Sigma, \delta, q_s, F)$, where

- Q is a finite set called the states,
- Σ is a finite set called the alphabet,
- $\delta : Q \times \Sigma \to Q$ is the transition function,
- $q_s \in Q$ is the start state,
- $F \subseteq Q$ is the set of accept states.

Finite Automata



This particular finite automaton is given by the 5-tuple $(Q, \Sigma, \delta, q_s, F)$, where

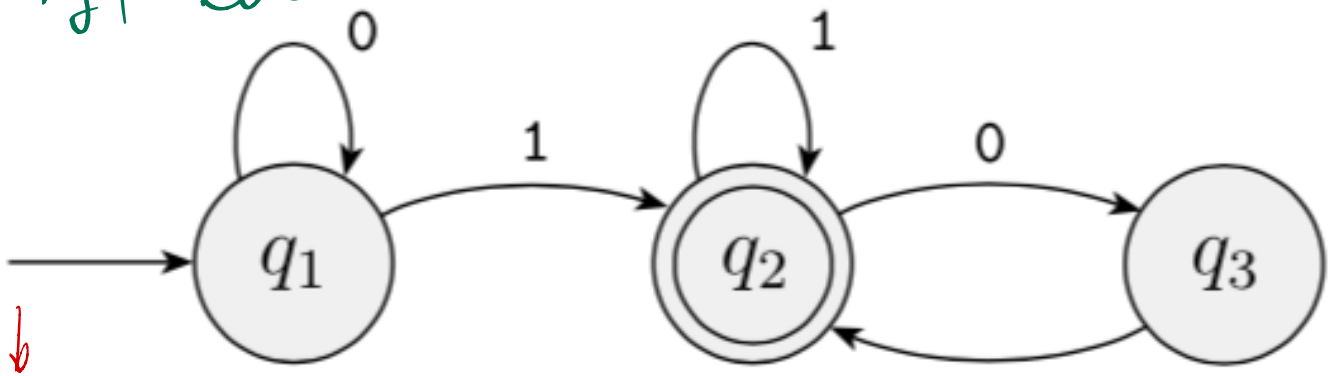
- $Q = \{q_1, q_2, q_3\}$
- $\Sigma = \{0, 1\}$
- δ is:

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

- start state is: q_1
- accepts states are: $\{q_2\}$

Finite Automata

#0's after 1st 1 is even & at least one 1

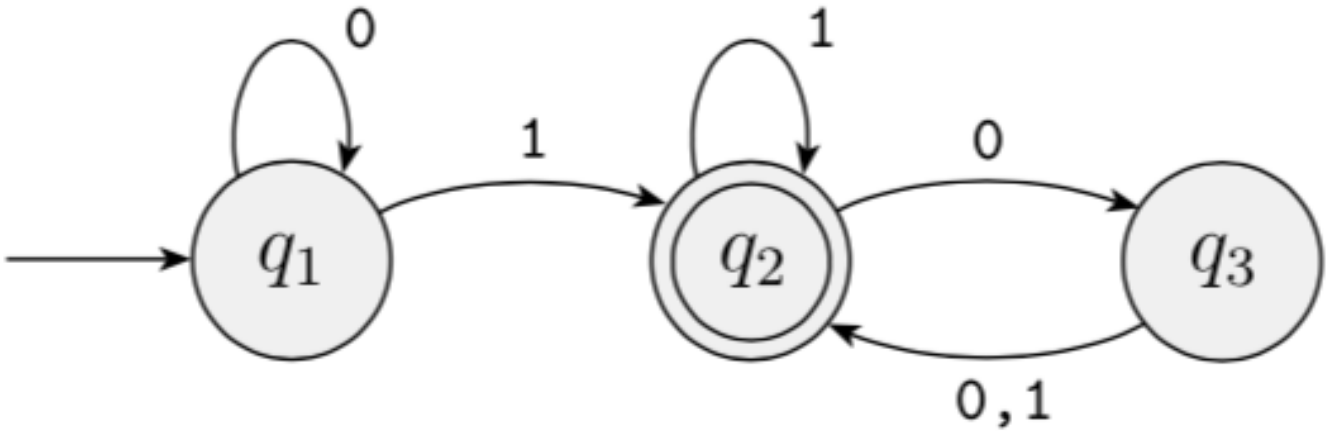


0 1 2 3 4 5 ...

- What does this machine M do?

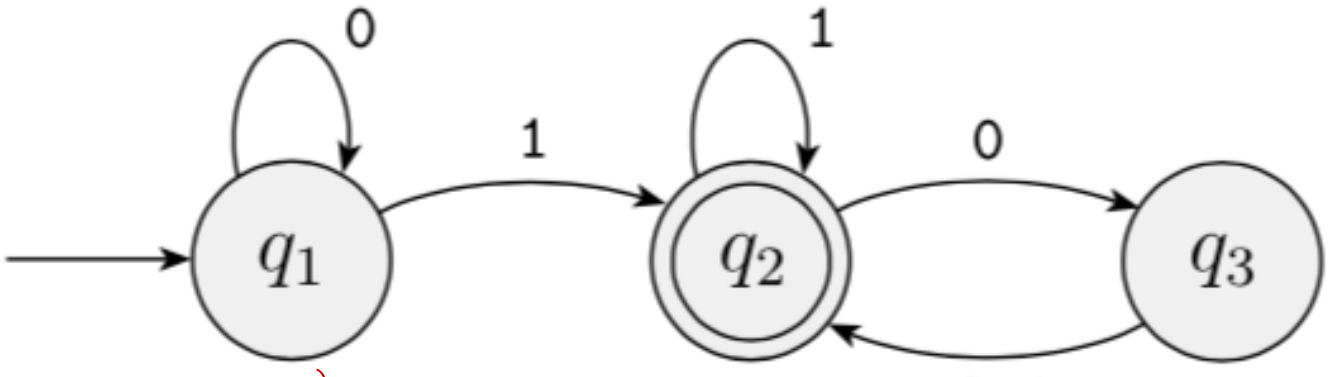
#0's after the last 1 is even & at least one 1

Finite Automata



- What does this machine M do?
- The set of all strings that M accepts is the **language** of the machine, $L(M)$.

Finite Automata



$\delta(q_1, 1) = q_2$, $\delta(q_2, 1) = q_2$

- What does this machine M do?
- The set of all strings that M accepts is the **language** of the machine, $L(M)$.
- $A = \{w | w$ contains at least one 1 and an even number of 0s follow the last 1$\}$.

Computation of a Finite Automaton

Let $M = (Q, \Sigma, \delta, q_S, F)$ be a finite automaton and let $w = w_1 w_2 \ldots w_n$ be a string where each w_i is a member of the alphabet Σ. Then M accepts w if a sequence of states $r_0, r_1, \ldots, r_n$ in Q exists with three conditions:

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

Computation of a Finite Automaton

Let $M = (Q, \Sigma, \delta, q_S, F)$ be a finite automaton and let $w = w_1 w_2 \dots w_n$ be a string where each w_i is a member of the alphabet Σ. Then M accepts w if a sequence of states $r_0, r_1, \dots, r_n$ in Q exists with three conditions:

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

About automaton computation:

- A machine may accept several strings, but it always recognizes only one language.

Computation of a Finite Automaton

Let $M = (Q, \Sigma, \delta, q_S, F)$ be a finite automaton and let $w = w_1 w_2 \dots w_n$ be a string where each w_i is a member of the alphabet Σ. Then M accepts w if a sequence of states $r_0, r_1, \dots, r_n$ in Q exists with three conditions:

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

About automaton computation:

- A machine may accept several strings, but it always recognizes only one language.
- If the machine accepts no strings, it still recognizes one language, the empty language $\emptyset$.

Designing Automata

Let's design some simple automata over the alphabet $\{0, 1\}$:

Designing Automata

Let's design some simple automata over the alphabet $\{0, 1\}$:

- accepts everything

Designing Automata

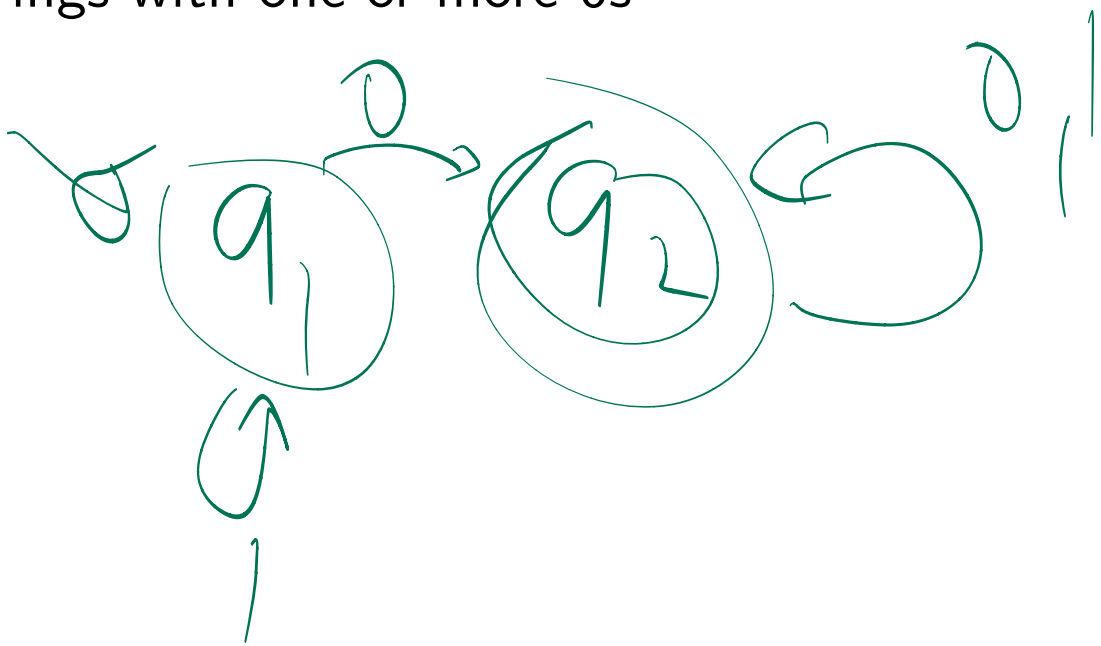
Let's design some simple automata over the alphabet $\{0, 1\}$:

- accepts everything
- rejects everything

Designing Automata

Let's design some simple automata over the alphabet $\{0, 1\}$:

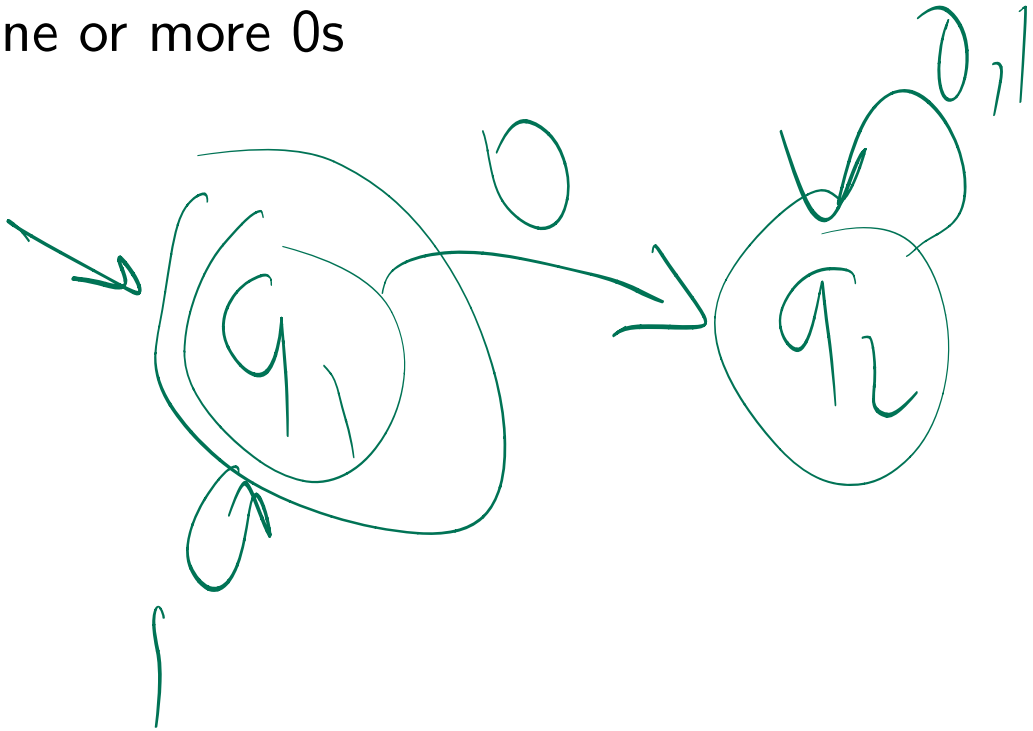
- accepts everything
- rejects everything
- accepts strings with one or more 0s



Designing Automata

Let's design some simple automata over the alphabet $\{0, 1\}$:

- accepts everything
- rejects everything
- accepts strings with one or more 0s
- rejects strings with one or more 0s



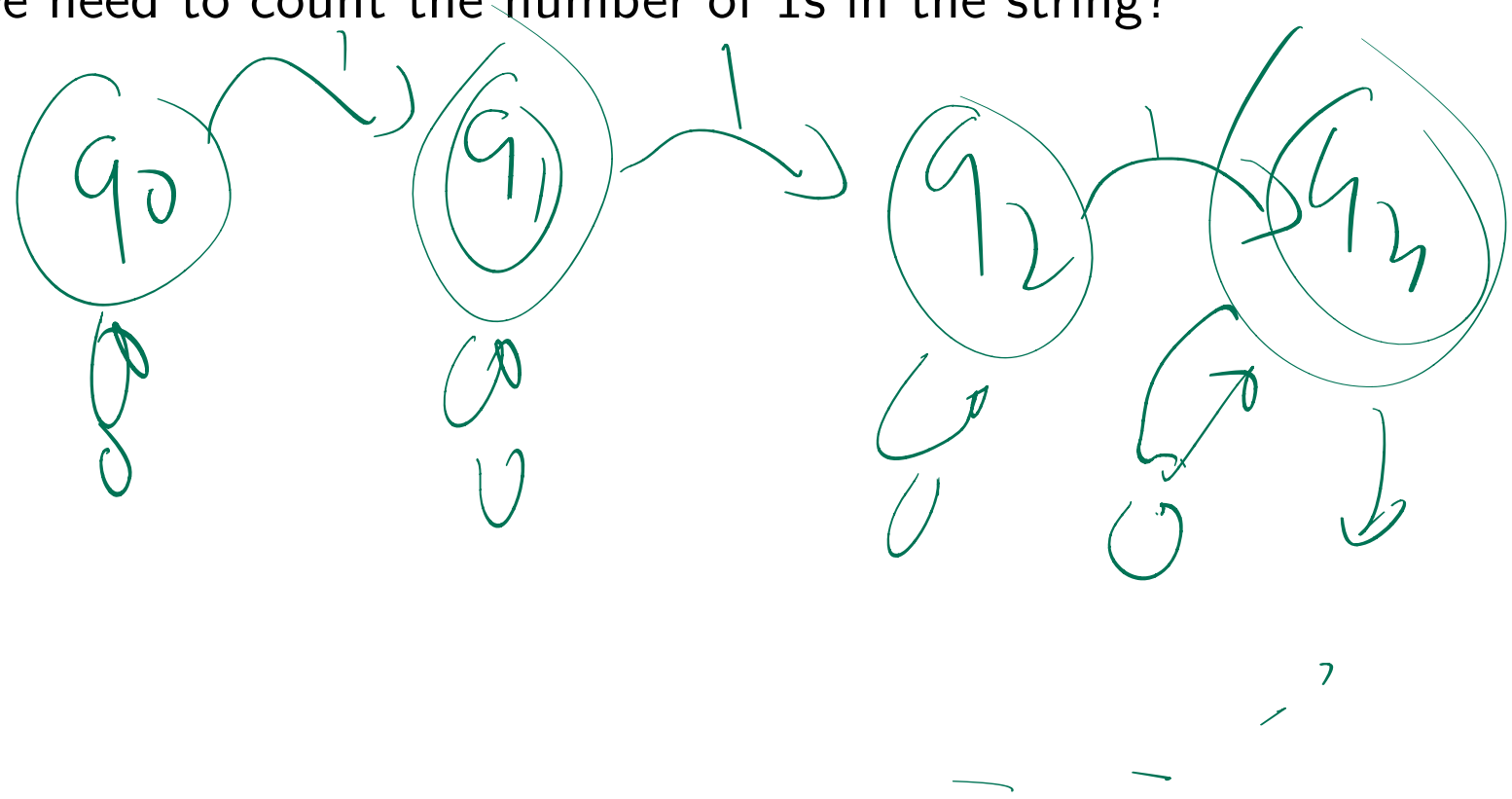
Designing Automata

Let's design an automaton that recognizes strings with an odd number of 1s, over the alphabet $\{0, 1\}$.

Designing Automata

Let's design an automaton that recognizes strings with an odd number of 1s, over the alphabet $\{0, 1\}$.

- do we need to count the number of 1s in the string?



Designing Automata

Let's design an automaton that recognizes strings with an odd number of 1s, over the alphabet $\{0, 1\}$.

- do we need to count the number of 1s in the string?
- just keep track whether the number so far is odd or even

Designing Automata

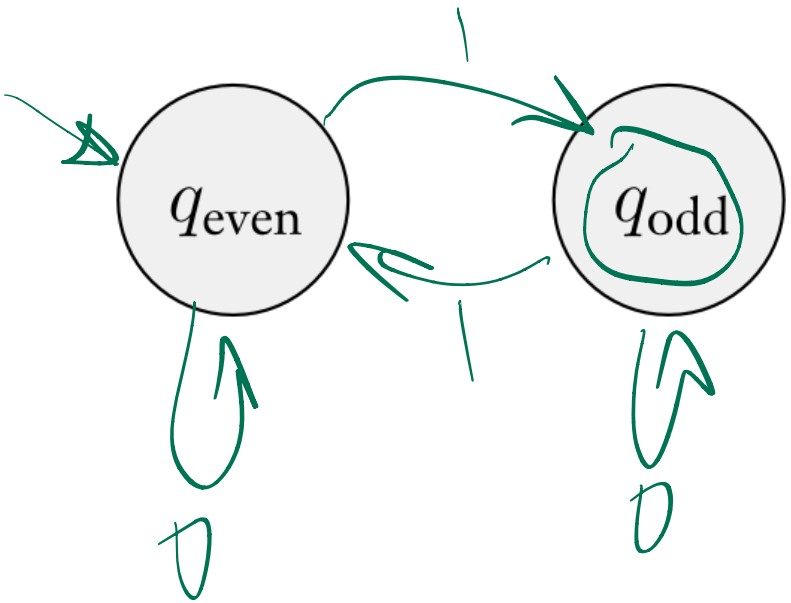
Let's design an automaton that recognizes strings with an odd number of 1s, over the alphabet $\{0, 1\}$.

- do we need to count the number of 1s in the string?
- just keep track whether the number so far is odd or even
- we can do this with just two states

Designing Automata

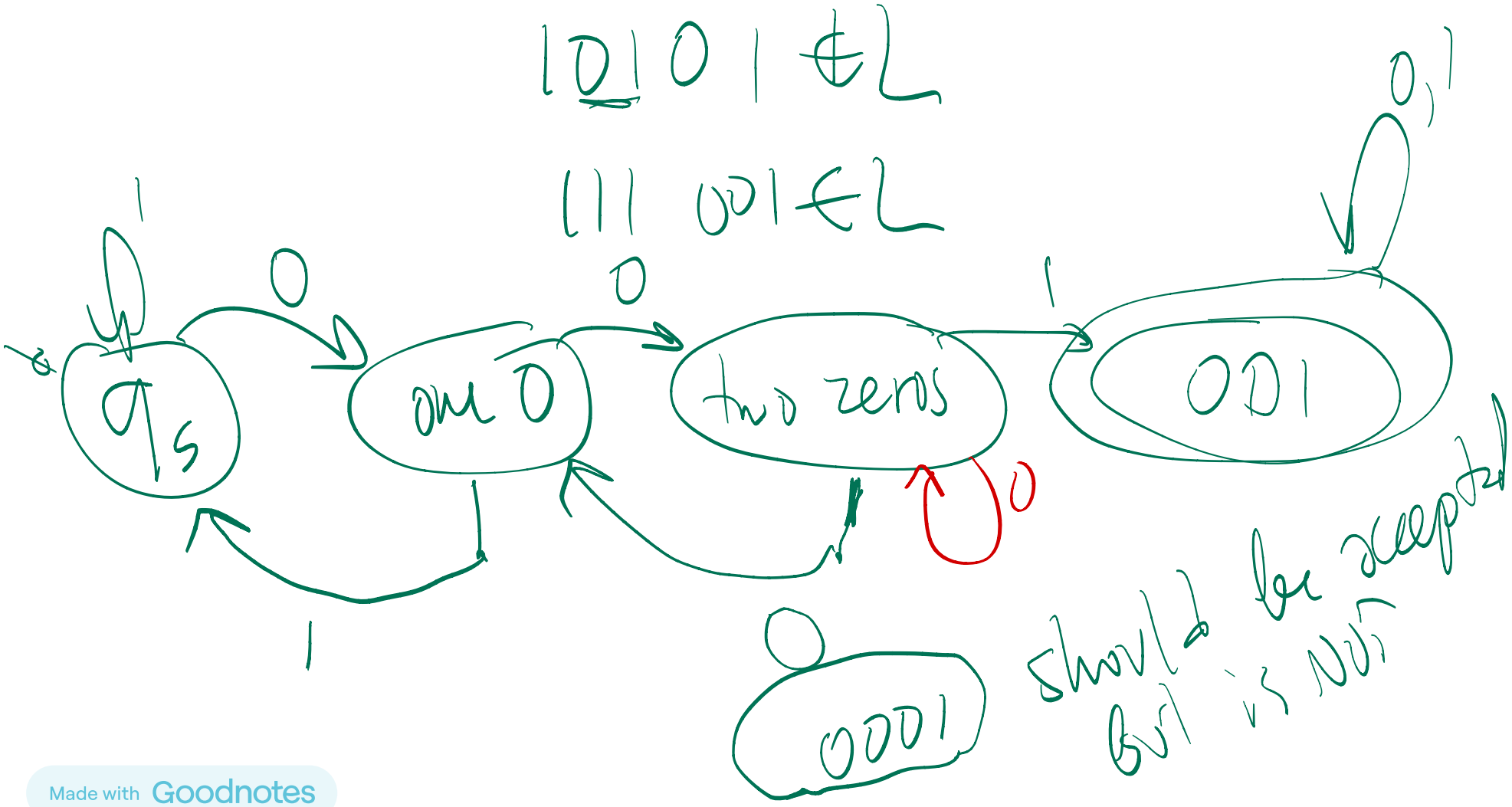
Let's design an automaton that recognizes strings with an odd number of 1s, over the alphabet $\{0, 1\}$.

- do we need to count the number of 1s in the string?
- just keep track whether the number so far is odd or even
- we can do this with just two states



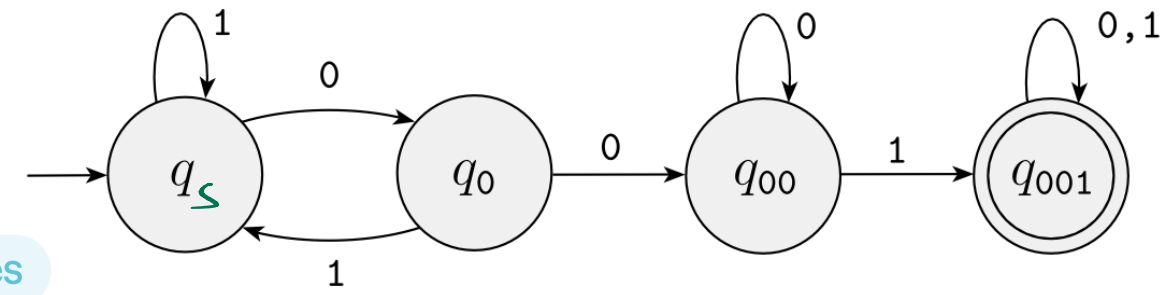
Designing Automata

Let's design an automaton that recognizes strings that contain 001 as a substring, again over the alphabet $\{0, 1\}$.



Designing Automata

Let's design an automaton that recognizes strings that contain 001 as a substring, again over the alphabet $\{0, 1\}$.



Regular Operations

Definition

A language is called a **regular language** if some finite automaton recognizes it.

Regular Operations

Definition

A language is called a **regular language** if some finite automaton recognizes it.

Let A and B be languages. We define the regular operations union, concatenation, and star as follows:

- Union: $A \cup B = \{x | x \in A \text{ or } x \in B\}$.
- Concatenation: $A \circ B = \{xy | x \in A \text{ and } y \in B\}$.
- Star: $A^* = \{x_1 x_2 \dots x_k | k \geq 0 \text{ and each } x_i \in A\}$.

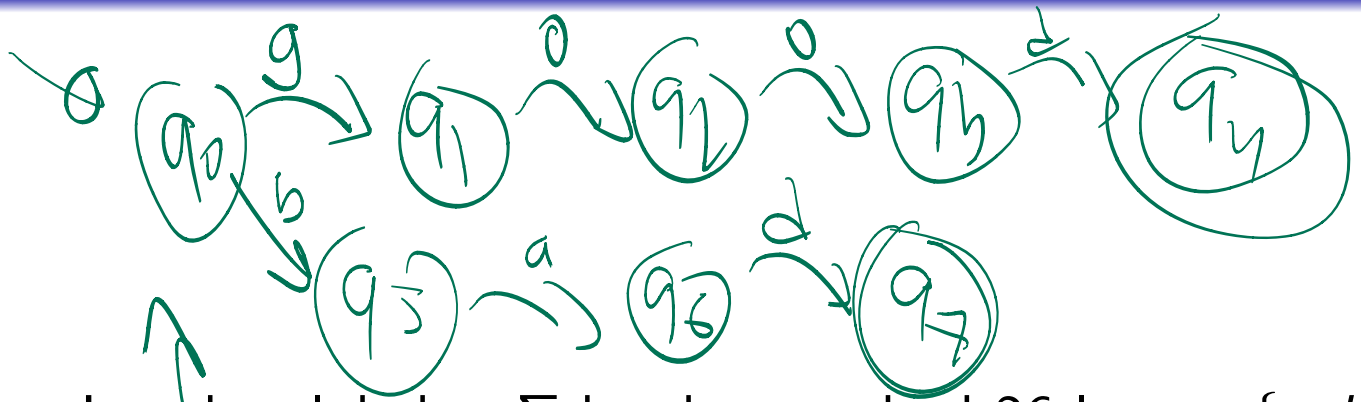
There are two special characters:

- ϵ: the empty string
- $\emptyset$: the empty language

Example of Regular Operations

Let the alphabet Σ be the standard 26 letters $\{a, b, \ldots, z\}$.

Example of Regular Operations



Let the alphabet Σ be the standard 26 letters $\{a, b, \ldots, z\}$.

If $A = \{good, bad\}$ and $B = \{boy, girl\}$, then

Example of Regular Operations

Let the alphabet Σ be the standard 26 letters $\{a, b, \ldots, z\}$.

If $A = \{good, bad\}$ and $B = \{boy, girl\}$, then

$A \cup B = \{good, bad, boy, girl\}$

Example of Regular Operations

Let the alphabet Σ be the standard 26 letters $\{a, b, \ldots, z\}$.

If $A = \{good, bad\}$ and $B = \{boy, girl\}$, then

$A \cup B = \{good, bad, boy, girl\}$

$A \circ B = \{goodboy, goodgirl, badboy, badgirl\}$

Example of Regular Operations

Let the alphabet Σ be the standard 26 letters $\{a, b, \ldots, z\}$.

If $A = \{good, bad\}$ and $B = \{boy, girl\}$, then

$A \cup B = \{good, bad, boy, girl\}$

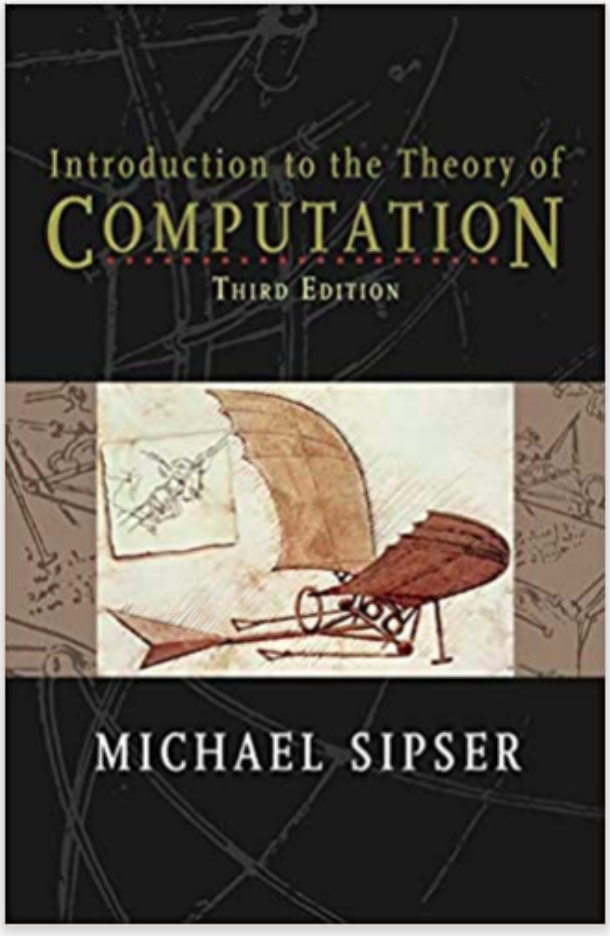
$A \circ B = \{goodboy, goodgirl, badboy, badgirl\}$

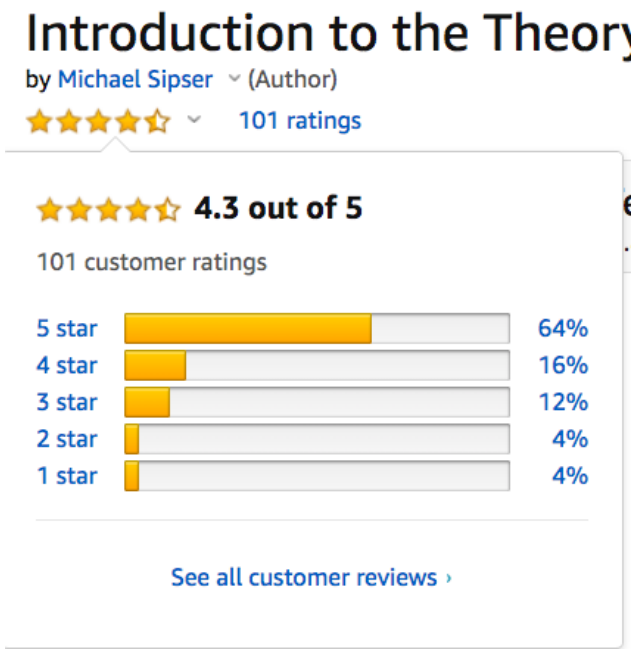
$A^* = \{\epsilon, good, bad, goodgood, goodbad, badgood, badbad, goodgoodgood, ...\}$

Administrivia

- Lectures, Mondays, **10:00**-11:30
- Lectures, Wednesdays, **9:00**-10:15
- Exercises, 90 minutes every week:
  - 2x Thursday 14:15-15:45
  - 2x Thursday 16:15-17:45
  - 1x Friday 14:15-15:45
- Five optional graded exams during Wednesday lectures
- Moodle page: more info, lecture notes, exercise sheets, etc.
- Exercise sheet #1 assigned today: work on in for 1 week
- Reading: Chapters 0 and 1
- Last time: DFAs,regular languages
- Today: DFAs, NFAs, properties of regular languages, regular expressions

Textbook and Other Reading



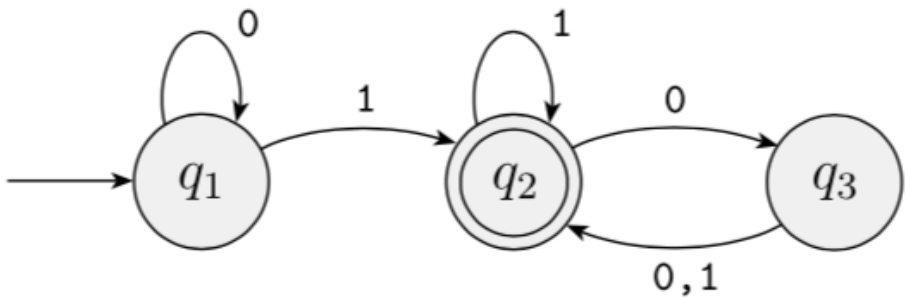


- Course textbook available online from the TUM Library.
- Information about accessing TUM ebooks.

Recall Finite Automata

A finite automaton is a 5-tuple $(Q, \Sigma, \delta, q_s, F)$, where

- Q is a finite set called the states,
- Σ is a finite set called the alphabet,
- δ is the transition function,
- $q_s \in Q$ is the start state,
- $F \subseteq Q$ is the set of accept states.



Also recall the two special symbols: the empty string ϵ, and the empty language $\emptyset$

Recall Regular Languages

Definition

A language is called a **regular language** if some finite automaton recognizes it.

- The language of an automaton is the set of strings it accepts
- For now **computation** simply means determining whether a given string is in the language or not (decision problem)
- There are 3 regular operations: union, concatenation and Kleene star

Properties of Regular Languages

Theorem

The class of regular languages is closed under the union operation.

Properties of Regular Languages

Theorem

The class of regular languages is closed under the union operation.

- What does "closed under union" mean?
- "If languages A_1 and A_2 are regular, then their union $A_1 \cup A_2$ is also regular."
- For example:
  - the class of integers is closed under addition
  - the class of integers is closed under multiplication
  - the class of integers is not closed under division

Properties of Regular Languages

Theorem

The class of regular languages is closed under the union operation.

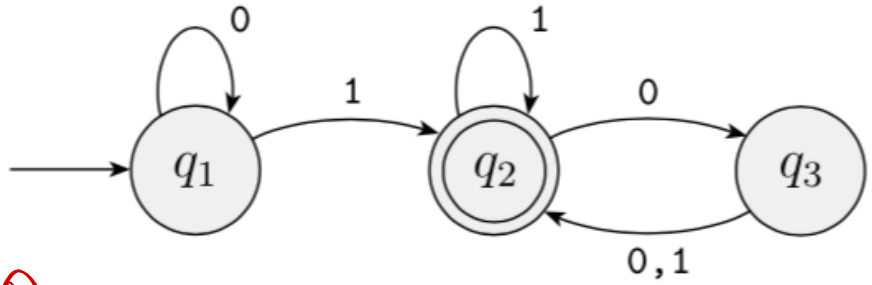
Proof.

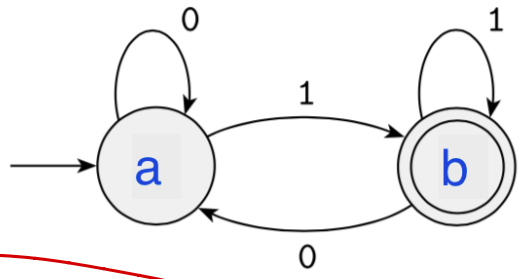
Given two regular languages A_1 and A_2 we want to show that $A_1 \cup A_2$ also is regular. Let M_1 be a finite automaton for A_1 and M_2 be a finite automaton for A_2. To prove that $A_1 \cup A_2$ is regular, we construct a finite automaton M that recognizes $A_1 \cup A_2$, using M_1 and M_2 as building blocks. M works by simultaneously simulating M_1 and M_2 and accepting if either of the simulations accept. M can keep track of the simulations by using as many states as the product of the states in M_1 and M_2. □

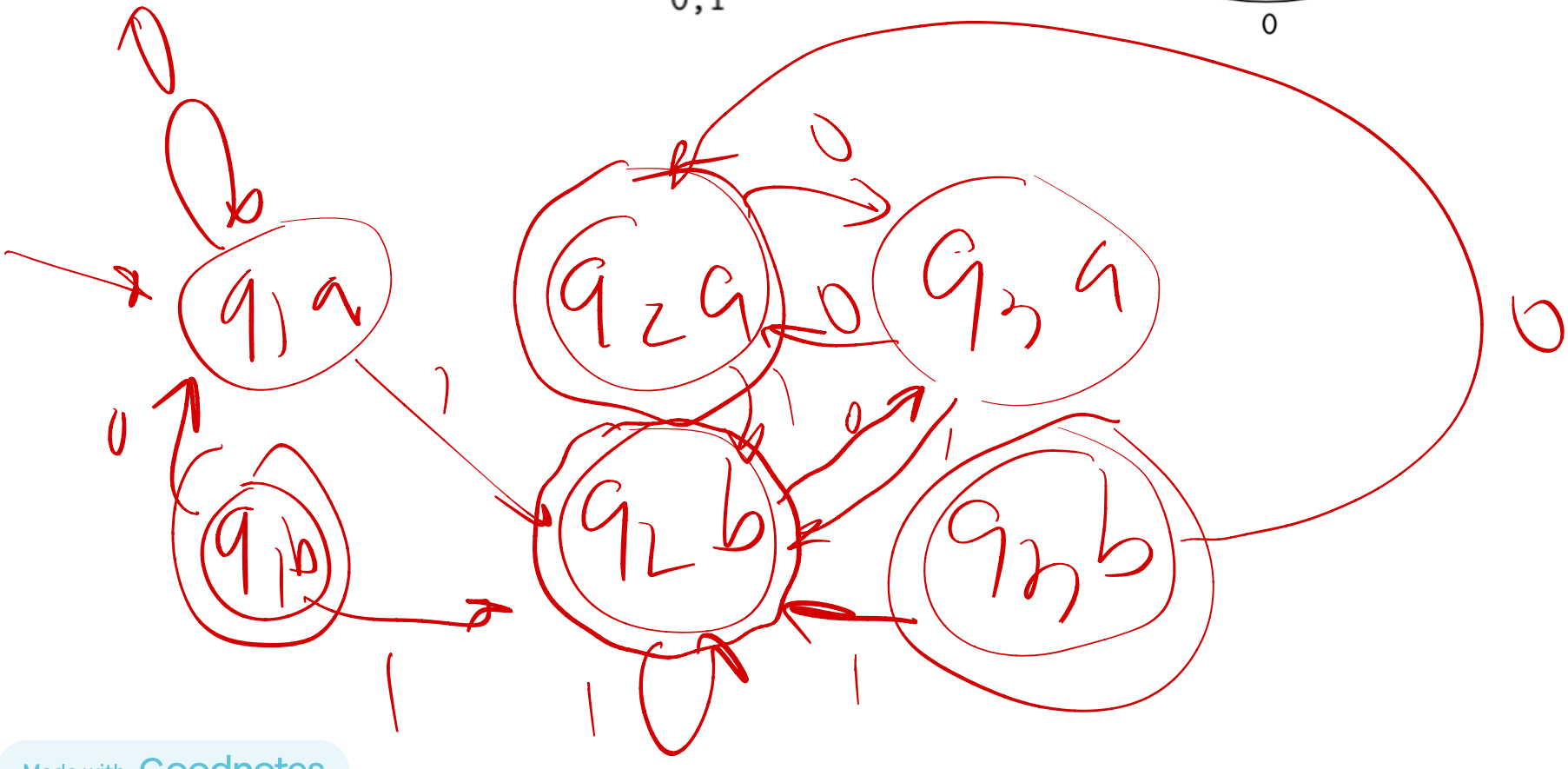
Closure under Union

Example

try replacing 0 & 1 with 2 & 3







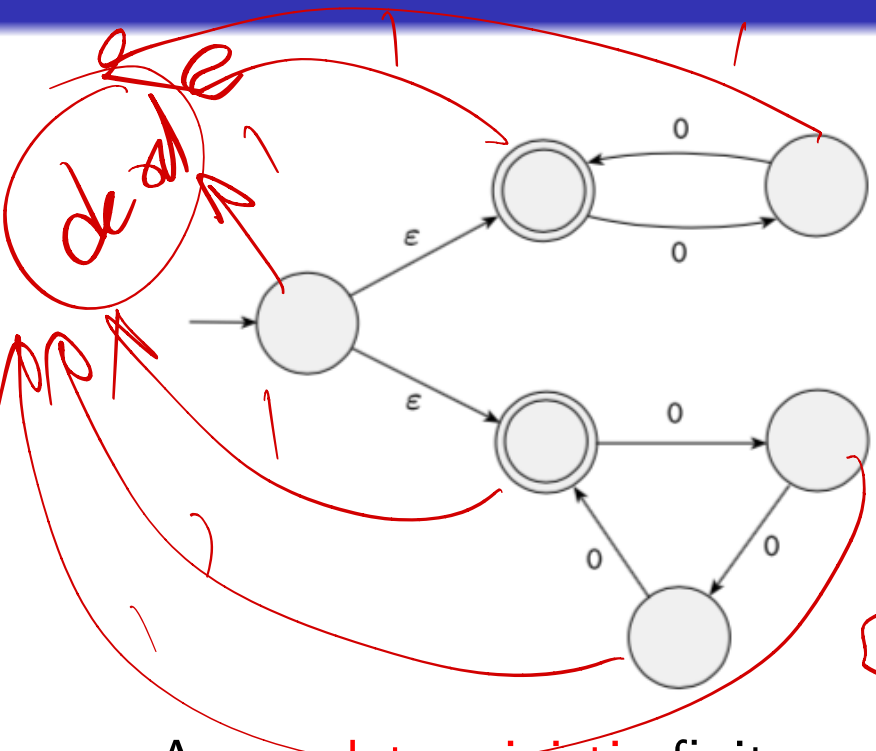
Closure under Union: Detailed Proof

Proof.

Let M_1 recognize A_1, where $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$, and M_2 recognize A_2, where $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$. Construct M to recognize $A_1 \cup A_2$, where $M = (Q, \Sigma, \delta, q_0, F)$.

1. $Q = \{(r_1, r_2) | r_1 \in Q_1 \text{ and } r_2 \in Q_2\}$.
   Q is the Cartesian product of the sets $Q1 \times Q_2$.
2. Σ is the alphabet of M_1 and M_2.
   Theorem is true if alphabets are different; then $\Sigma = \Sigma_1 \cup \Sigma_2$.
3. $\delta((r_1, r_2), a) = (\delta_1(r_1, a), \delta_2(r_2, a))$, for each $(r_1, r_2) \in Q$ and each $a \in \Sigma$
4. q_0 is the pair (q_1, q_2).
5. $F = \{(r_1, r_2) | r_1 \in F_1 \text{ or } r_2 \in F_2\}$.

Nondeterministic Finite Automata



A non-deterministic finite automaton is a 5-tuple $(Q, \Sigma, \delta, q_s, F)$, where

- Q is a finite set called the states,
- Σ is a finite set called the alphabet,
- $\delta : Q \times \Sigma_\epsilon \to P(Q)$ is the transition function,
- $q_s \in Q$ is the start state,
- $F \subseteq Q$ is the set of accept states.

$\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$

powerset of Q: every possible subset of states

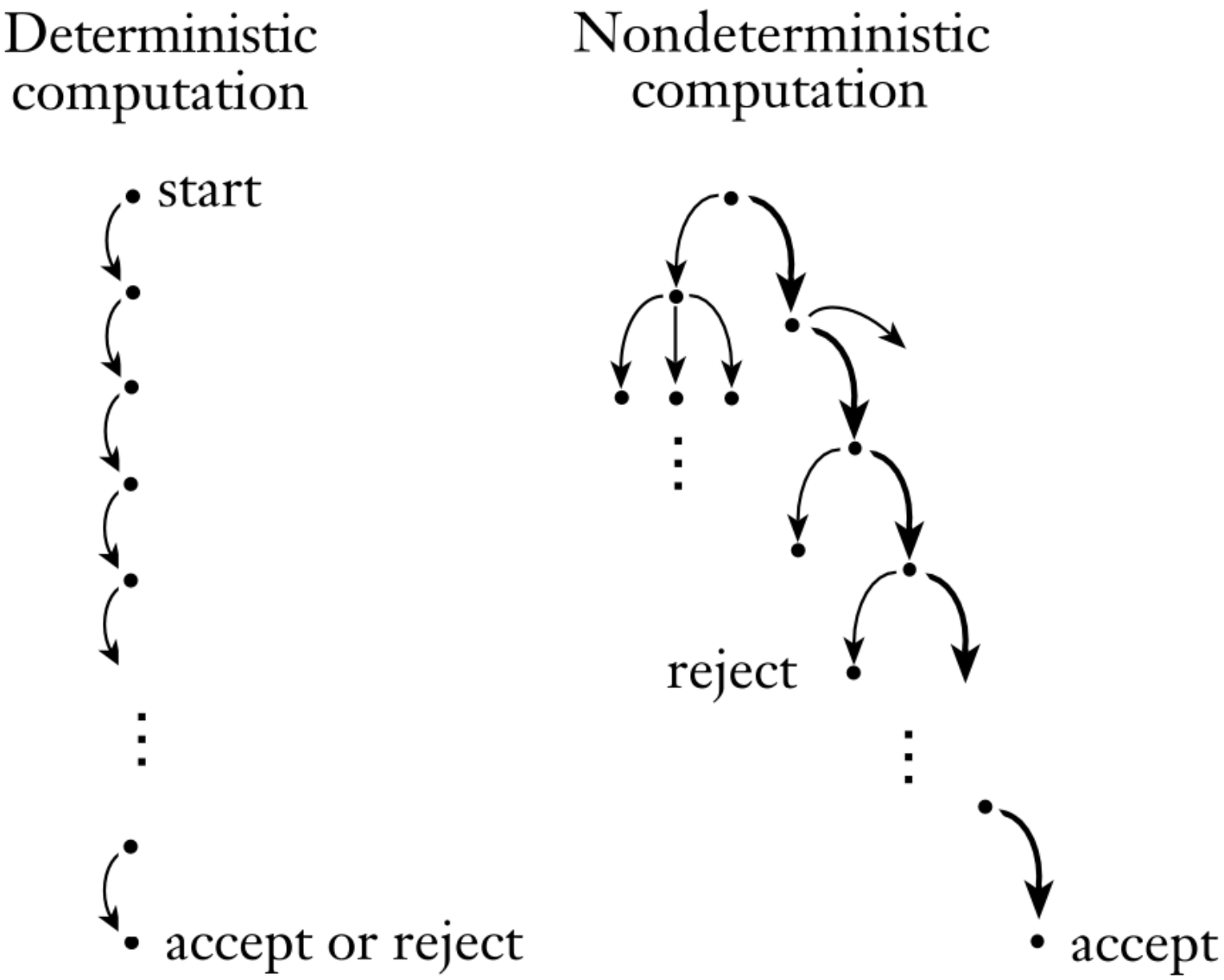
NFAs

- Nondeterminism is a generalization of determinism, so every deterministic finite automaton (DFA) is automatically a nondeterministic finite automaton (NFA).
- Every state of a DFA always has exactly one exiting transition arrow for each symbol in the alphabet. In an NFA, a state may have zero, one, or many exiting arrows for each alphabet symbol.
- In a DFA, labels on the transition arrows are symbols from the alphabet. In general, an NFA may have arrows labeled with members of the alphabet or ϵ. Zero, one, or many arrows may exit from each state with the label ϵ.

NFA Computation

- Suppose an NFA runs on an input string and comes to a state with multiple ways to proceed on the same symbol.
- After reading that symbol, the machine splits into multiple copies of itself and follows all the possibilities in parallel.
- Each copy of the machine takes one of the possible ways to proceed and continues as before.
- If there are subsequent choices, the machine splits again.
- If the next input symbol doesn't appear on any of the arrows exiting the state occupied by a copy of the machine, that copy of the machine dies, along with the branch of the computation associated with it.
- Finally, if any one of these copies of the machine is in an accept state at the end of the input, the NFA accepts the input string.

NFA Computation



NFA Computation

Let $N = (Q, \Sigma, \delta, q_S, F)$ be a finite automaton and let $w = w_1 w_2 \ldots w_n$ be a string where each w_i is a member of the alphabet Σ_ϵ. Then N accepts w if a sequence of states $r_0, r_1, \ldots, r_n$ in Q exists with three conditions:

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

An NFA accepts if any of its (possibly exponentially many) computation paths ends in an accept state.

Are Nondeterministic FA More Powerful?

What do we mean by *powerful*?

- the computational power of a machine is measured by the class of languages that it can recognize
- the computational power of a machine then is measured by the number of problems it can solve

Are Nondeterministic FA More Powerful?

What do we mean by *powerful*?

- the computational power of a machine is measured by the class of languages that it can recognize
- the computational power of a machine then is measured by the number of problems it can solve

So, while it's easier to design NFAs (they can be smaller and simpler), it turns out that NFAs recognize exactly the same class of languages as DFAs

NFAs

Theorem

A language is regular if and only if some NFA recognizes it.

Proof.

$\Rightarrow$ Let L be a regular language. Then by definition there exists a DFA for L. Since a DFA is a special case of an NFA (an NFA without multiple transitions on the same symbol), we are done.

NFAs

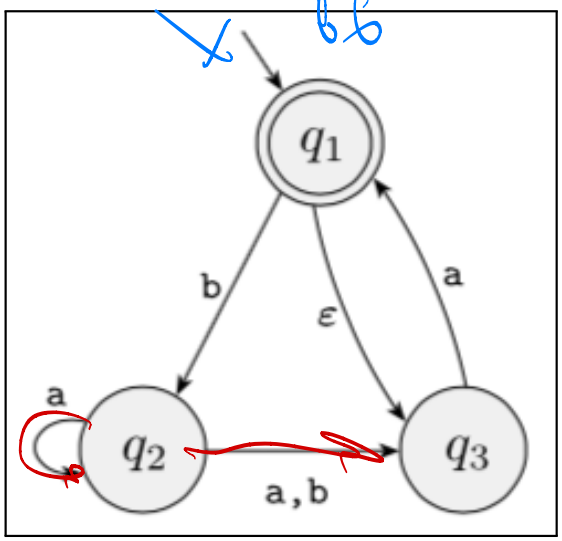
Theorem

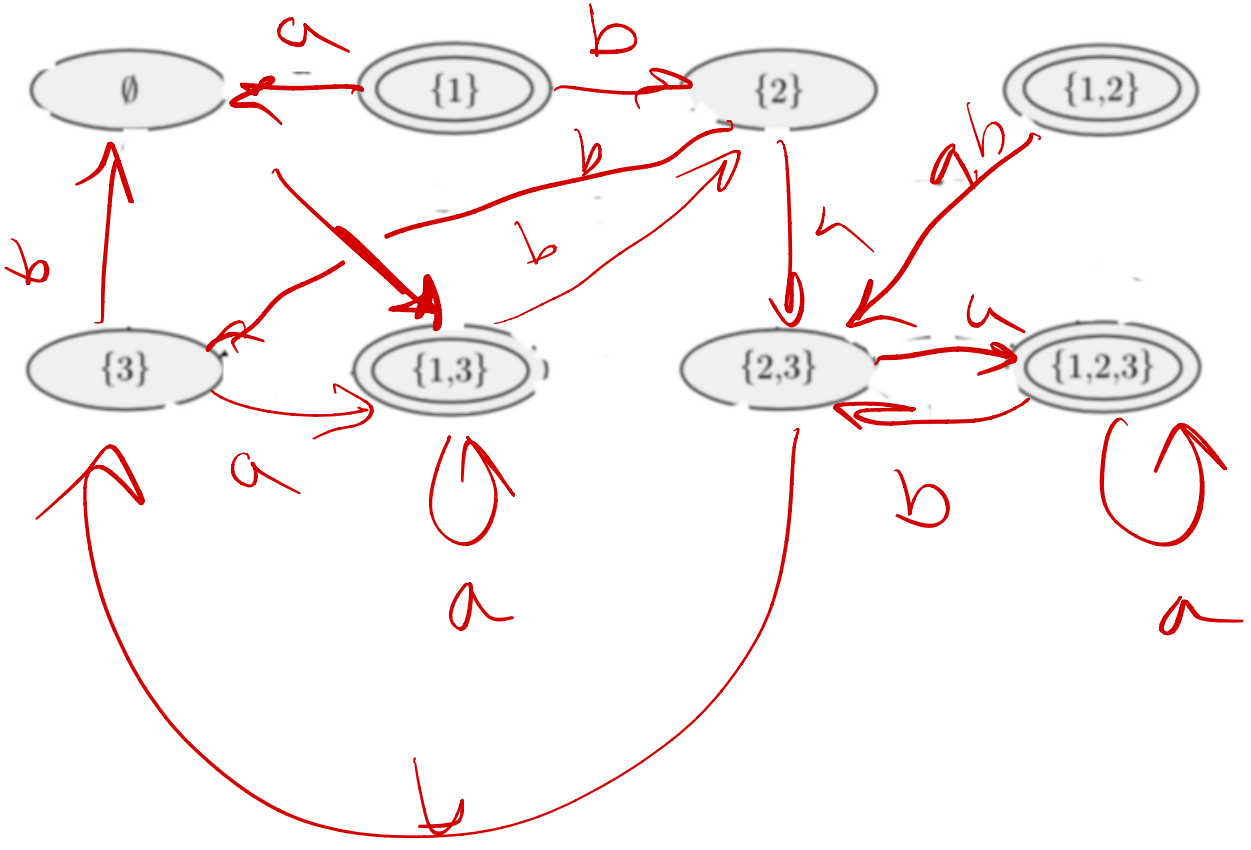
A language is regular if and only if some NFA recognizes it.

Proof.

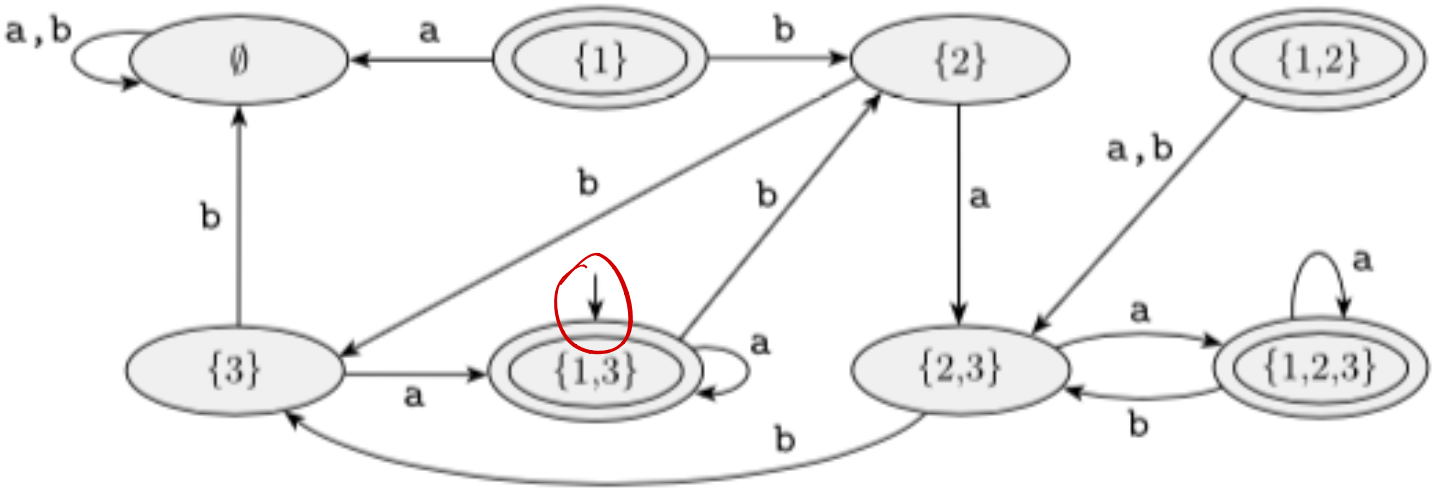
$\Rightarrow$ Let L be a regular language. Then by definition there exists a DFA for L. Since a DFA is a special case of an NFA (an NFA without multiple transitions on the same symbol), we are done.
$\Leftarrow$ Let N be an NFA. We will show how to create an equivalent DFA M. Then $L(N) = L(M)$ and by definition, this language is regular. $\square$

NFA to DFA Example





NFA to DFA Example



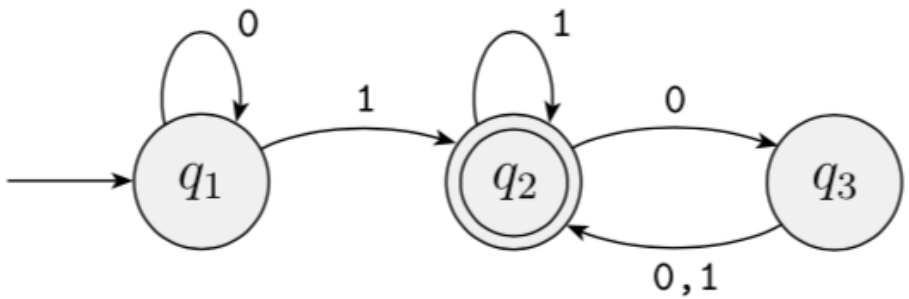
Administrivia

- Exercises, 90 minutes every week, starting this week:
  - 2x Thursday 14:15-15:45
  - 2x Thursday 16:15-17:45
  - 1x Friday 14:15-15:45
- Exercise sheet #1 will be discussed in these meetings
- Optional Exam #1: Wednesday October 29, 9:00-10:00
- Reading: Chapters 0 and 1
- Last time: DFAs, NFAs, regular languages
- Today: NFAs, properties of regular languages, regular expressions

Recall Finite Automata

A finite automaton is a 5-tuple $(Q, \Sigma, \delta, q_s, F)$, where

- Q is a finite set called the states,
- Σ is a finite set called the alphabet,
- δ is the transition function,
- $q_s \in Q$ is the start state,
- $F \subseteq Q$ is the set of accept states.



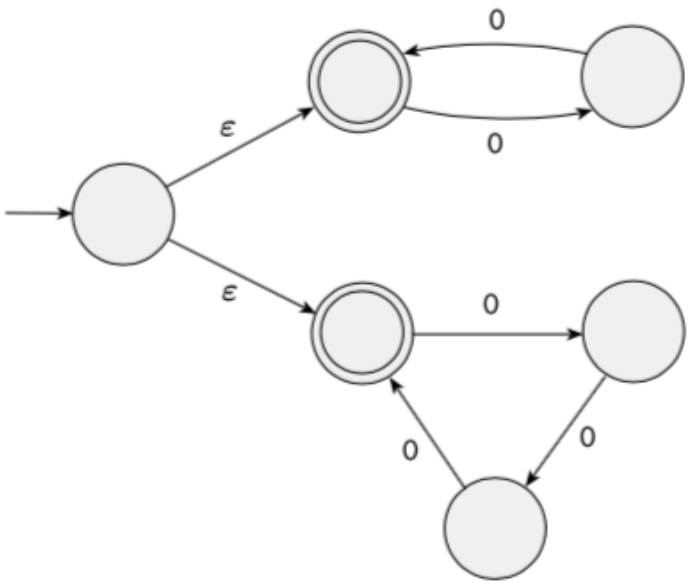
Definition

A language is called a **regular language** if some finite automaton recognizes it.

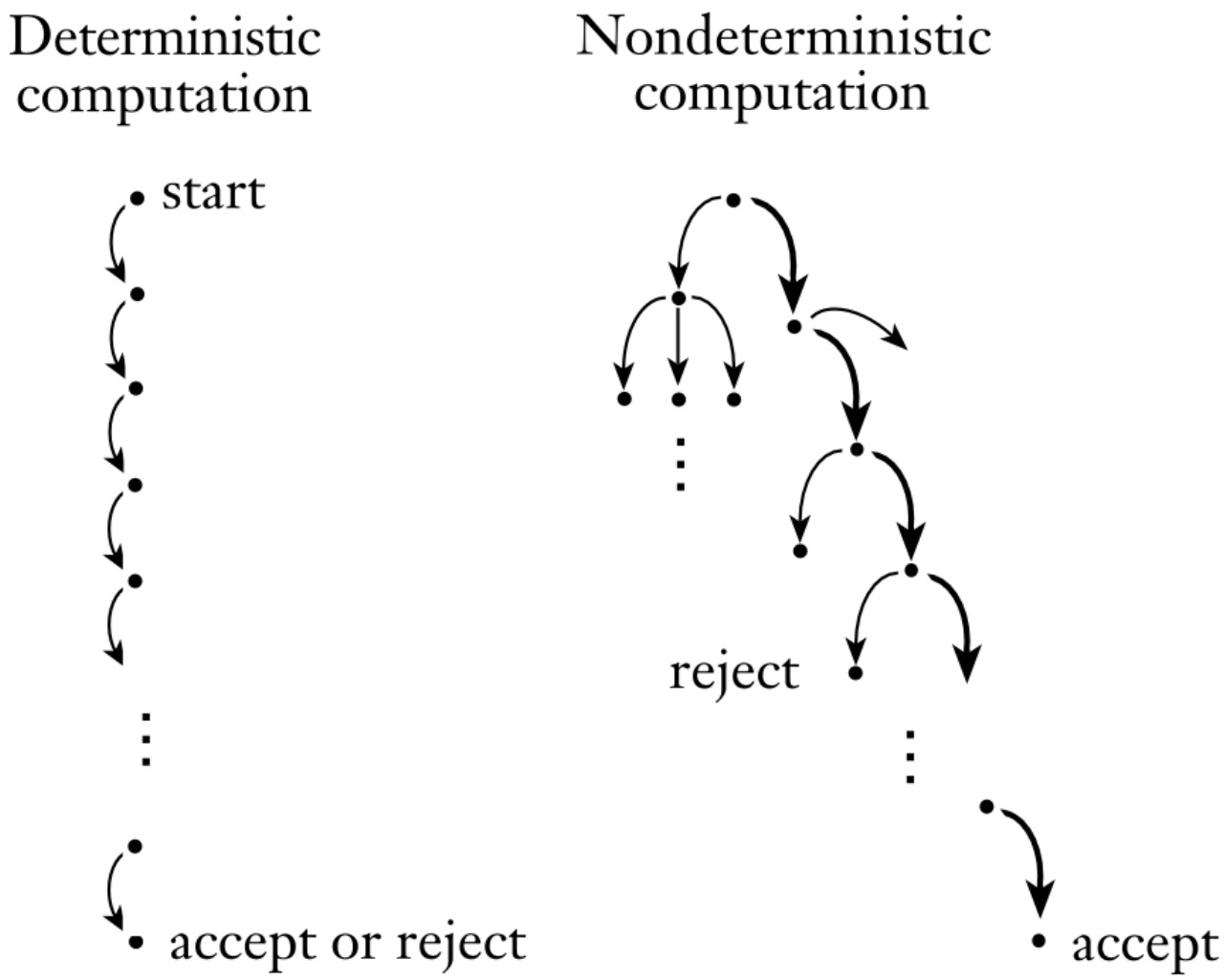
Nondeterministic Finite Automata

A non-deterministic finite automaton is a 5-tuple $(Q, \Sigma, \delta, q_s, F)$, where

- Q is a finite set called the states,
- Σ is a finite set called the alphabet,
- $\delta : Q \times \Sigma_\epsilon \to P(Q)$ is the transition function,
- $q_s \in Q$ is the start state,
- $F \subseteq Q$ is the set of accept states.



NFA Computation



An NFA accepts if any of its (possibly exponentially many) computation paths ends in an accept state.

NFAs

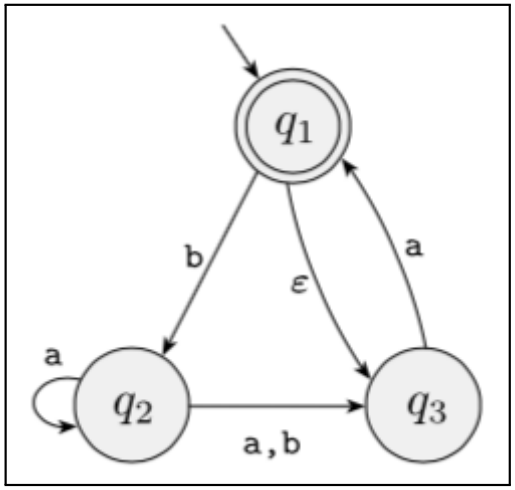
Theorem

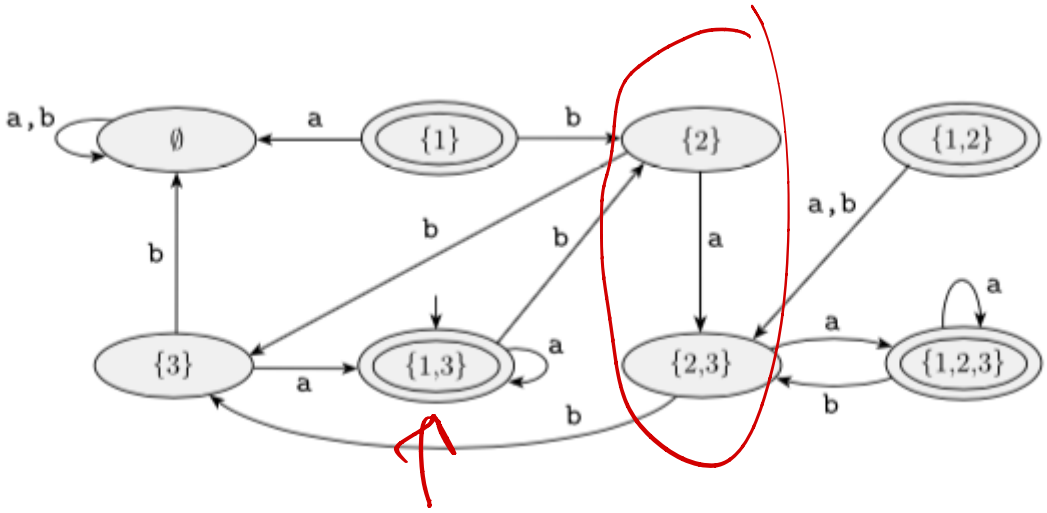
A language is regular if and only if some NFA recognizes it.

Proof.

$\Rightarrow$ Let L be a regular language. Then by definition there exists a DFA for L. Since a DFA is a special case of an NFA (an NFA without multiple transitions on the same symbol), we are done.
$\Leftarrow$ Let N be an NFA. We will show how to create an equivalent DFA M. Then $L(N) = L(M)$ and by definition, this language is regular. $\square$

NFA to DFA Example





To create the DFA M that recognizes the same language as the given NFA N, we make M *simulate* the behavior of N.

Note that the number of states in the DFA could be exponentially larger than the number of states in the NFA.

Equivalence of NFAs and DFAs (cont.)

Theorem

Every NFA has an equivalent DFA

Proof.

Let $N = (Q, \Sigma, \delta, q_s, F)$ be an NFA that recognizes language A.
We will construct DFA $M = (Q', \Sigma, \delta', q'_s, F')$ that recognizes A.

- $Q' = P(Q)$
- for $R \in Q'$ and $a \in \Sigma$ let $\delta'(R, a) = \{q \in Q | q \in E(\delta(r, a))$ for some $r \in R\}$
- $q'_S = E(q_s)$
- $F' = \{R \in Q' | R$ contains an accept state of $N\}$

where $E(R) = \{q | q$ can be reached from R by traveling along 0 or more ϵ arrows. $\square$

Closure under Union

Theorem

The class of regular languages is closed under the union operation.

Proof.

We proved this with DFAs; now we redo it with NFAs.

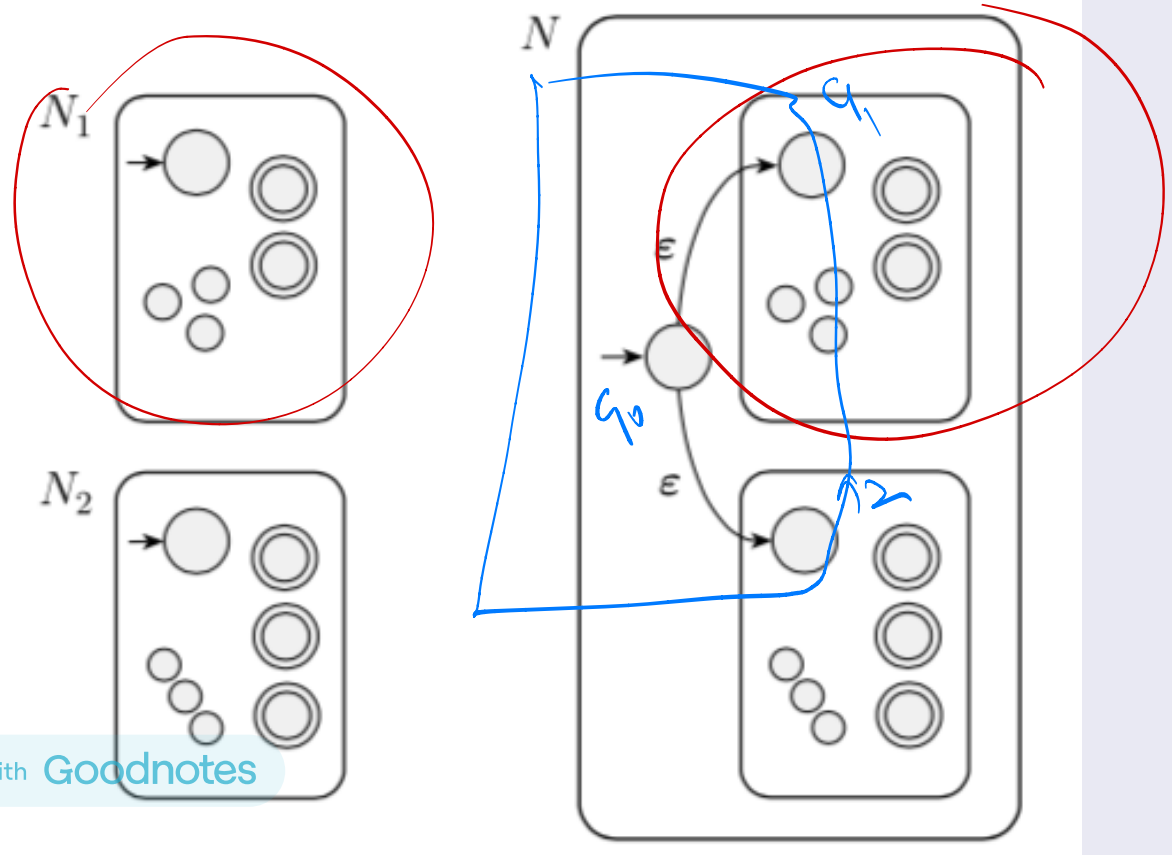
Closure under Union

Theorem

The class of regular languages is closed under the union operation.

Proof.

We proved this with DFAs; now we redo it with NFAs.



Closure under Union

Theorem

The class of regular languages is closed under the union operation.

Proof.

Let $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognize A_1, and
$N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ recognize A_2.
Construct $N = (Q, \Sigma, \delta, q_0, F)$ to recognize $A_1 \cup A_2$.

- $Q = \{q_0\} \cup Q_1 \cup Q_2$.
- start state q_0
- $F = F_1 \cup F_2$.

$$\delta(q, a) = \begin{cases} \delta_1(q, a) : q \in Q_1 \\ \delta_2(q, a) : q \in Q_2 \\ \{q_1, q_2\} : q = q_0, a = \epsilon \\ \emptyset : q = q_0, a \neq \epsilon \end{cases}$$

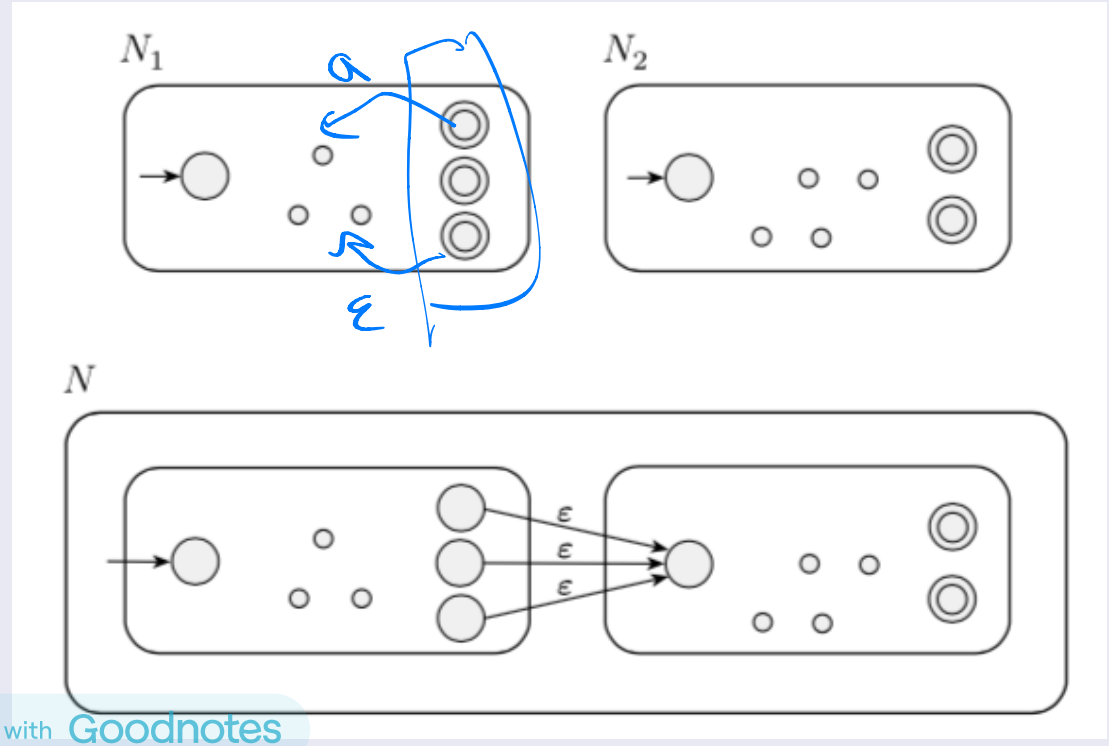
Closure under Concatenation

Recall concatenation: $A_1 \circ A_2 = \{xy | x \in A_1 \text{ and } y \in A_2\}$.

Theorem

The class of regular languages is closed under the concatenation operation.

Proof.



□

Closure under Concatenation

Theorem

The class of regular languages is closed under the concatenation operation.

Proof.

Let $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognize A_1, and
$N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ recognize A_2.
Construct $N = (Q, \Sigma, \delta, q_1, F_2)$ to recognize $A_1 \circ A_2$.

- $Q = Q_1 \cup Q_2$.
- start state q_1
- accept states F_2

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

□

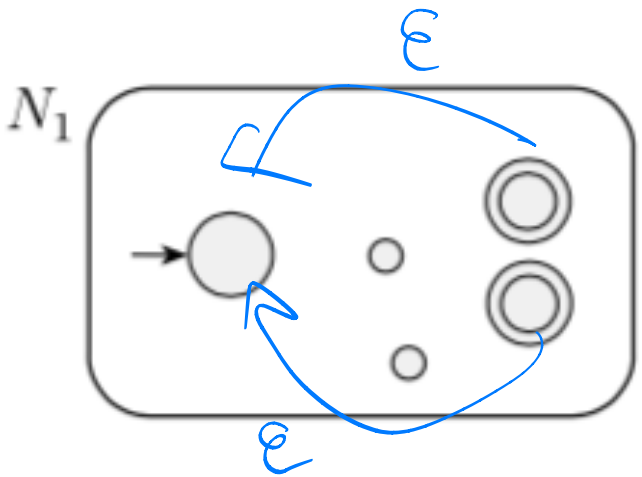
Closure under Star

Recall star: $A^* = \{x_1 x_2 \ldots x_k | k \geq 0 \text{ and each } x_i \in A\}$.

Theorem

The class of regular languages is closed under the star operation.

Proof.



almost correct
- recognizes $A^* - \{\varepsilon\}$
- we also want ε

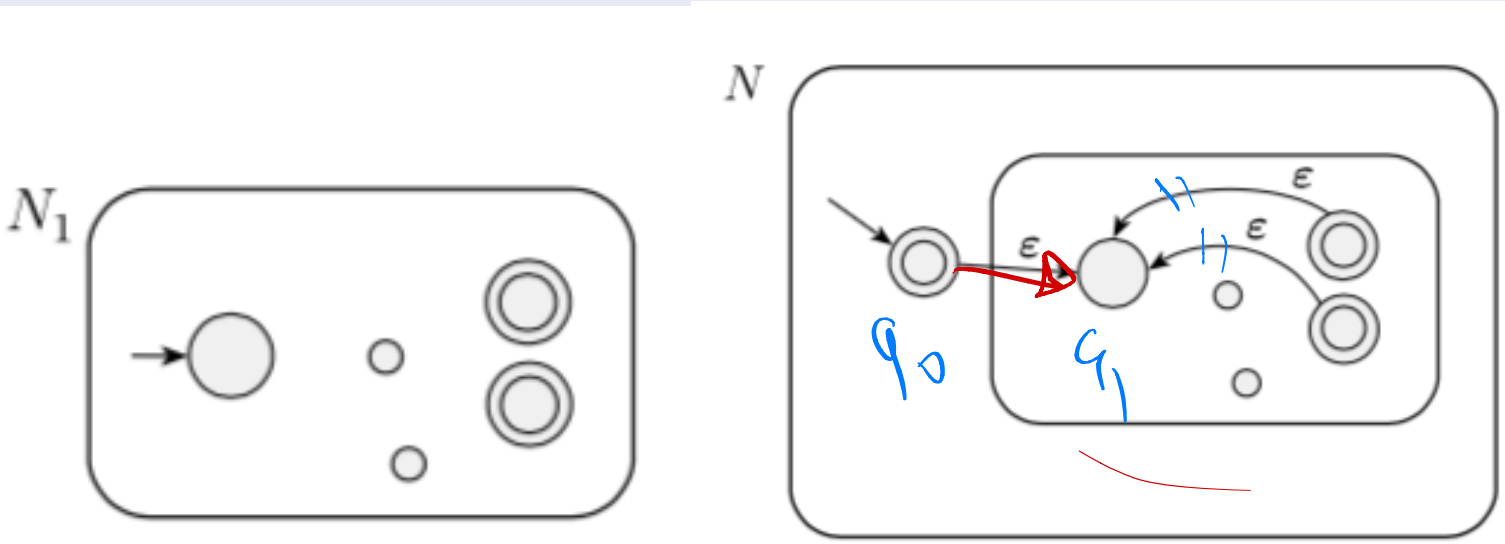
Closure under Star

Recall star: $A^* = \{x_1 x_2 \ldots x_k | k \geq 0 \text{ and each } x_i \in A\}$.

Theorem

The class of regular languages is closed under the star operation.

Proof.



□

Closure under Star

Theorem

The class of regular languages is closed under the star operation.

Proof.

Let $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognize A_1.
Construct $N = (Q, \Sigma, \delta, q_0, F)$ to recognize A_1^*.

- $Q = \{q_0\} \cup Q_1$
- start state q_0
- $F = \{q_0\} \cup F_1$.

$$\delta(q, a) = \begin{cases} \delta_1(q, a) : q \in Q_1, q \notin F_1 \\ \delta_1(q, a) : q \in F_1, a \neq \epsilon \\ \delta_1(q, a) \cup \{q_1\} : q \in F_1, a = \epsilon \\ \{q_1\} : q = q_0, a = \epsilon \end{cases}$$

DFAs and NFAs Again

- We have proven that DFAs and NFAs are equivalent in power
- DFAs and NFAs recognize the same class of languages: the regular languages
- We used NFAs to prove some closure properites of regular languages
- Some of these proofs can be easily replicated with DFAs but others are not so easy (e.g., Kleene star, concatenation)

Regular Expressions: Examples

Let $\Sigma = \{0, 1\}$

- $0^*10^* = \{w | w \text{ contains a single 1}\}$
- $\Sigma^*1\Sigma^* = \{w | w \text{ has at least one 1}\}$
- $\Sigma^*001\Sigma^* = \{w | w \text{ contains the string 001 as a substring}\}$
- $(\Sigma\Sigma)^* = \{w | w \text{ is a string of even length}\}$
- $(\Sigma\Sigma\Sigma)^* = \{w | w \text{ has length that is a multiple of 3}\}$
- $01 \cup 10 = \{01, 10\}$
- $0\Sigma^*0 \cup 1\Sigma^*1 \cup 0 \cup 1 = \{w | w \text{ starts and ends with the same symbol}\}$
- $00^*1^* = 0^+1^* = \{w | w \text{ starts with one or more 0s and ends in zero or more 1s}\}$
- $(0 \cup \epsilon)1^* = \{01^* \cup 1^*\}$
- $(0 \cup \epsilon)(1 \cup \epsilon) = \{\epsilon, 0, 1, 01\}$
- $1^*\emptyset = \emptyset$
- $\emptyset^* = \{\epsilon\}$

Regular Expressions

R is a regular expression if R is

1. a for some a in the alphabet Σ,
2. ϵ,
3. $\emptyset$,
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions,
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions, or
6. (R_1^*), where R_1 is a regular expression.

RE Identities

- $R \cup \emptyset = R$
- $R \circ \epsilon = R$
- $R \cup \epsilon$ may NOT equal R – why?
  if $R = 0$, then $L(R) = \{0\}$, but $L(R \cup \epsilon) = \{0, \epsilon\}$
- $R \circ \emptyset$ may not equal R – why?
  if $R = 0$, then $L(R) = \{0\}$, but $L(R \circ \emptyset) = \emptyset$

Regular expressions are useful tools in the design of compilers

- Programming language tokens (such as variable names and constants) are usually described by REs
- a numerical constant that may include a fractional part and/or a sign may be described as a member of the language $(+ \cup - \cup \epsilon)(D^+ \cup D^+.D^* \cup D^*.D^+)$, where $D = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ is the alphabet of decimal digits
- examples of valid strings are 42, 3.14159, +7., and -.01

RE $\iff$ FAs

Theorem

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

Proof.

$\Leftarrow$ Consider some regular expression R that describes language A. We show how to convert R into an NFA recognizing A.

1. $R = a$ for some $a \in \Sigma$. Then $L(R) = \{a\}$, and the following NFA recognizes $L(R)$

RE $\iff$ FAs

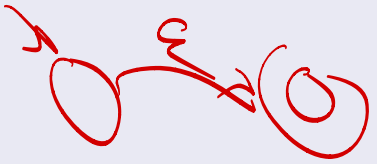
Theorem

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

Proof.

$\Leftarrow$ Consider some regular expression R that describes language A. We show how to convert R into an NFA recognizing A.

1. $R = a$ for some $a \in \Sigma$. Then $L(R) = \{a\}$, and the following NFA recognizes $L(R)$
2. $R = \epsilon$. Then $L(R) = \{\epsilon\}$, and the following NFA recognizes $L(R)$



RE $\iff$ FAs

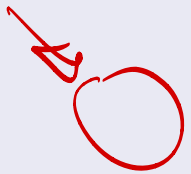
Theorem

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

Proof.

$\Leftarrow$ Consider some regular expression R that describes language A. We show how to convert R into an NFA recognizing A.

1. $R = a$ for some $a \in \Sigma$. Then $L(R) = \{a\}$, and the following NFA recognizes $L(R)$
2. $R = \epsilon$. Then $L(R) = \{\epsilon\}$, and the following NFA recognizes $L(R)$
3. $R = \emptyset$. Then $L(R) = \emptyset$, and the following NFA recognizes $L(R)$



RE $\iff$ FAs

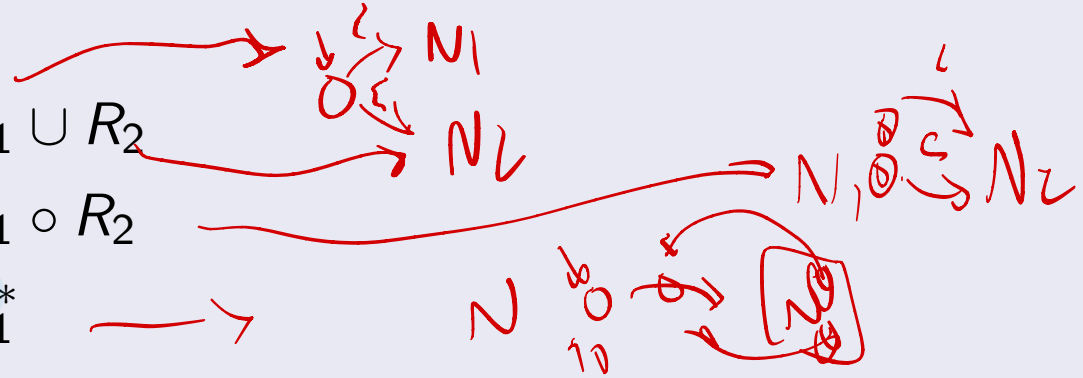
Theorem

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

Proof.

$\Leftarrow$ Consider some regular expression R that describes language A. We show how to convert R into an NFA recognizing A.

1. $R = a$ for some $a \in \Sigma$. Then $L(R) = \{a\}$, and the following NFA recognizes $L(R)$
2. $R = \epsilon$. Then $L(R) = \{\epsilon\}$, and the following NFA recognizes $L(R)$
3. $R = \emptyset$. Then $L(R) = \emptyset$, and the following NFA recognizes $L(R)$
4. $R = R_1 \cup R_2$
5. $R = R_1 \circ R_2$
6. $R = R_1^*$

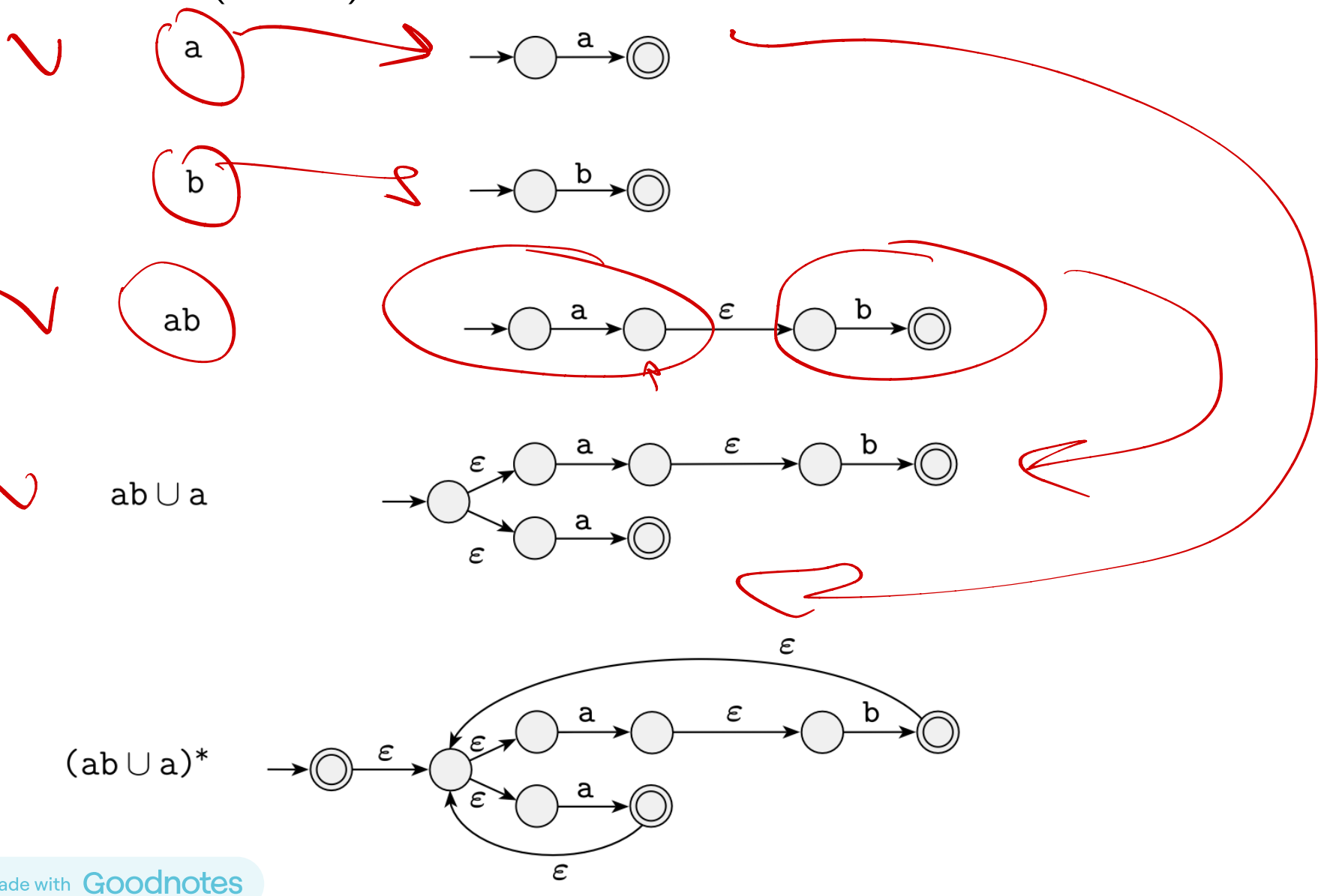


Example of RE to NFA conversion

Let $R = (ab \cup a)^*$

Example of RE to NFA conversion

Let $R = (ab \cup a)^*$



Administrivia

- Exercises, 90 minutes every week, starting this week:
  - 2x Thursday 14:15-15:45
  - 2x Thursday 16:15-17:45
  - 1x Friday 14:15-15:45
- Exercise sheet #1 will be discussed in these meetings
- Exercise sheet #2 assigned today
- Optional Exam #1: Wednesday October 29, 9:00-10:00
- Reading: Chapters 0 and 1
- Last time: DFAs, NFAs, regular languages
- Today: regular expressions, non-regular languages, Pumping Lemma

RE $\iff$ FAs

Theorem

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

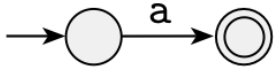
Proof.

$\Leftarrow$ Consider some regular expression R that describes language A. We show how to convert R into an NFA recognizing A. (We showed this last time, by example and gave a formal proof using the inductive definition of a RE.)
Rightarrow We show how to get a RE from a FA. $\square$

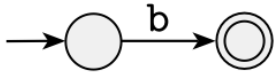
Example of RE to NFA conversion

Let $R = (ab \cup a)^*$

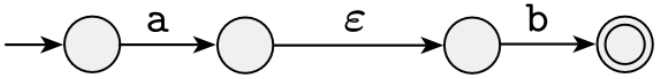
a



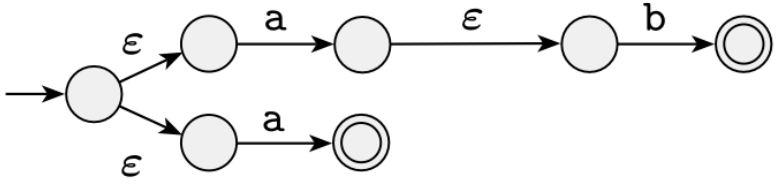
b



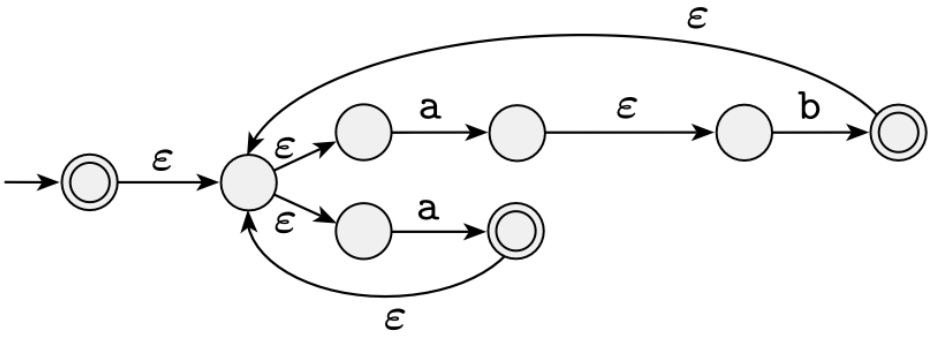
ab



ab ∪ a



(ab ∪ a)*



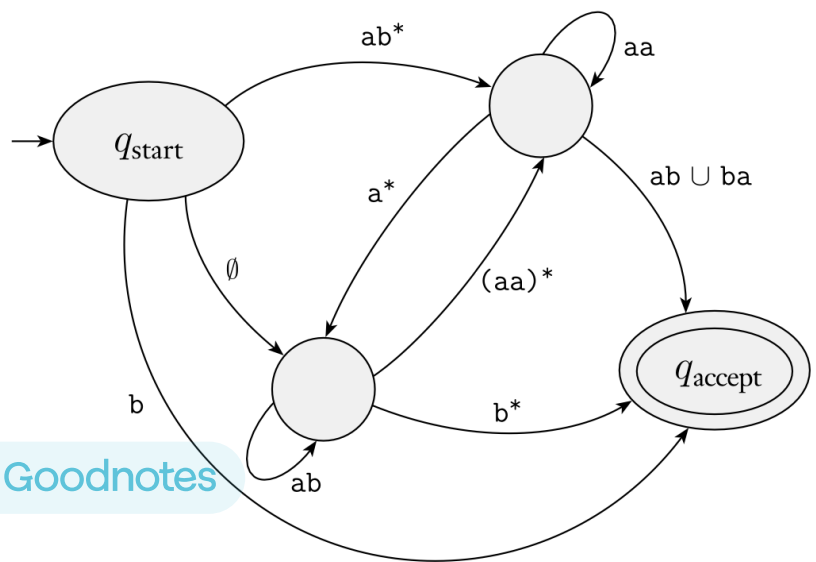
RE $\iff$ FAs

Theorem

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

Proof.

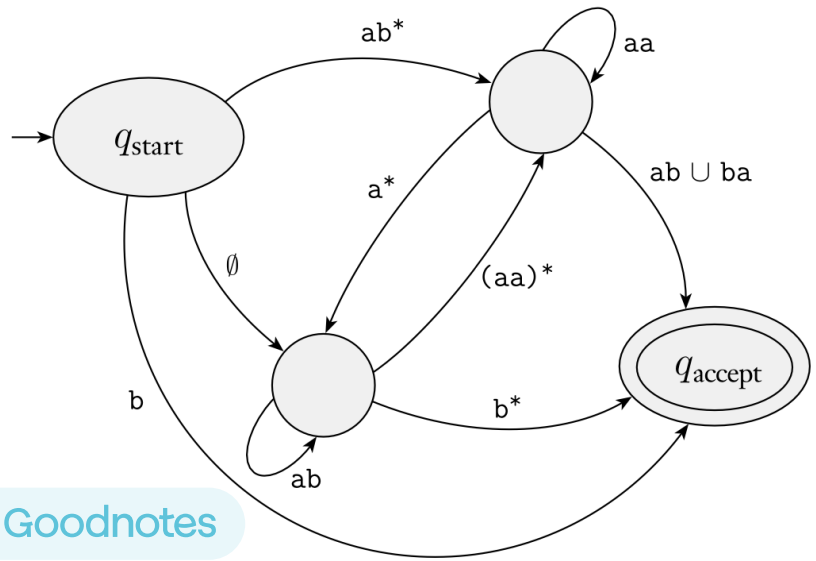
$\Rightarrow$ Given some regular language A, there exists a DFA that recognizes A. We show how to extract an equivalent regular expression, by means of a *generalized nondeterministic finite automaton (GNFA)*. $\square$



Example of DFA to RE conversion

Start with the given NFA and convert into GNFA:

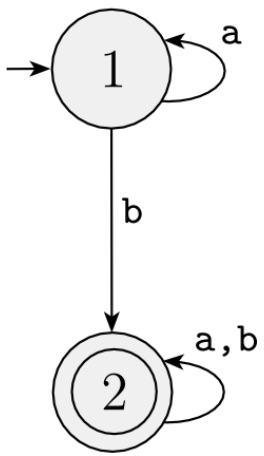
- start state has transition arrows to every other state
- start state has no arrows coming in from any other state
- there is only a single accept state, and it has arrows coming in from every other state
- accept state has no arrows going to any other state
- except for start and accept states, one arrow goes from every state to every other state; also from each state to itself



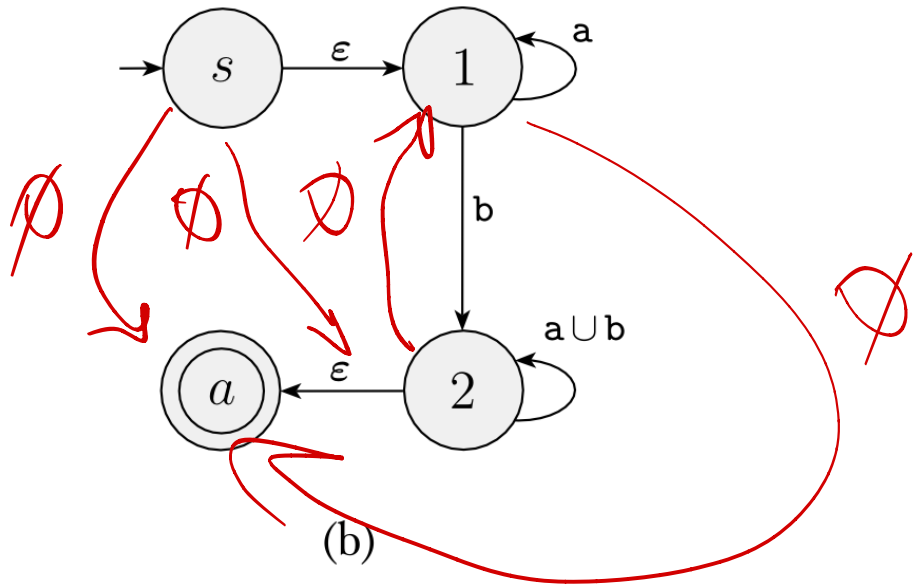
Example of DFA to RE conversion

This is easy to do:

- add new start state with ϵ arrow to the old start state
- add new accept state with ϵ arrows from old accept states
- If any arrows have multiple labels (or if there are multiple arrows going between the same two states in the same direction), we replace each with a single arrow whose label is the union of the previous labels.
- add arrows labeled $\emptyset$ between states that had no arrows



(a)

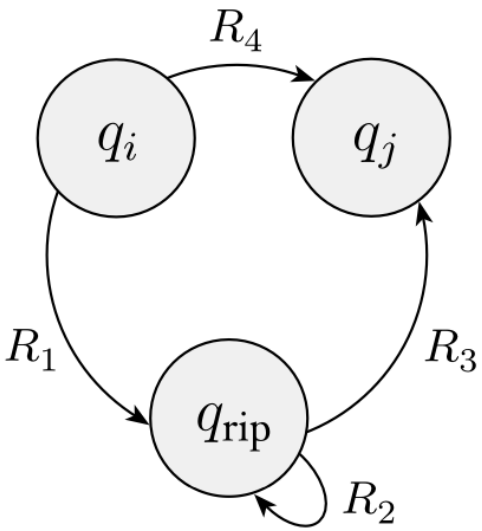


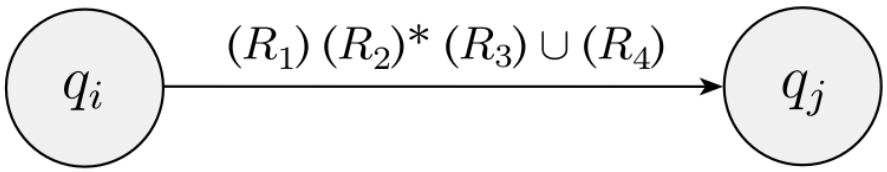
(b)

Example of DFA to RE conversion

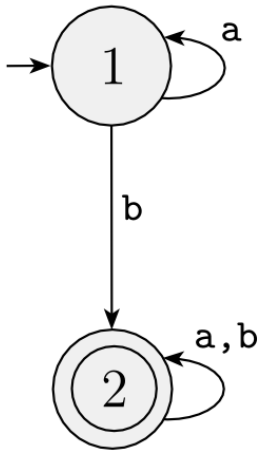
Now we remove one state at a time from the GNFA, while ensuring it still recognizes the same language, until only the start state and the accept state remain:

- let q_{rip} be the state we remove
- consider every pair of states (q_i, q_j)
- update (q_i, q_j) transition label to account for removal of q_{rip}

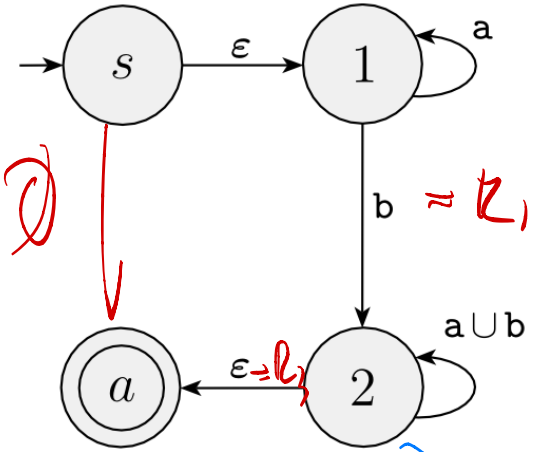




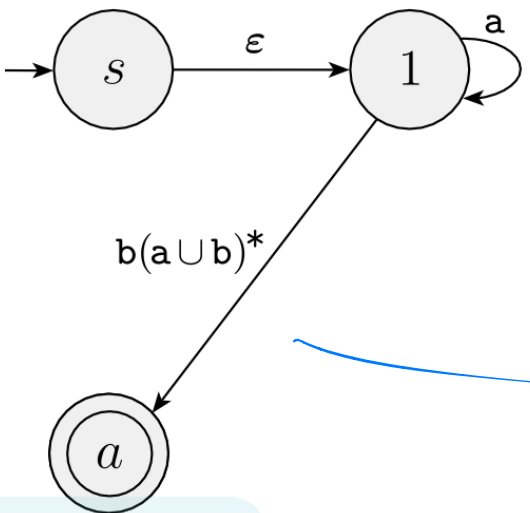
Example of DFA to RE conversion



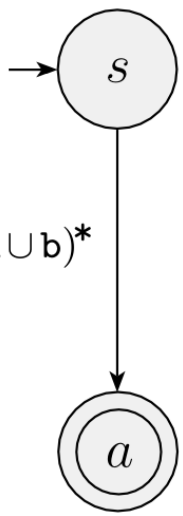
(a)



(b)

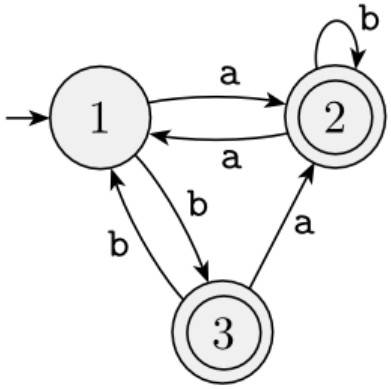


(c)

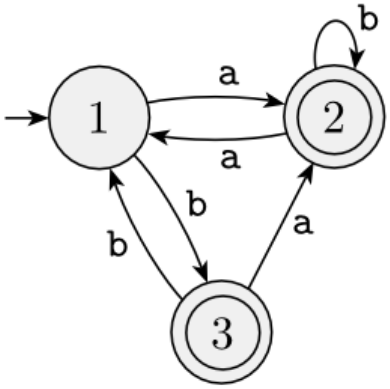


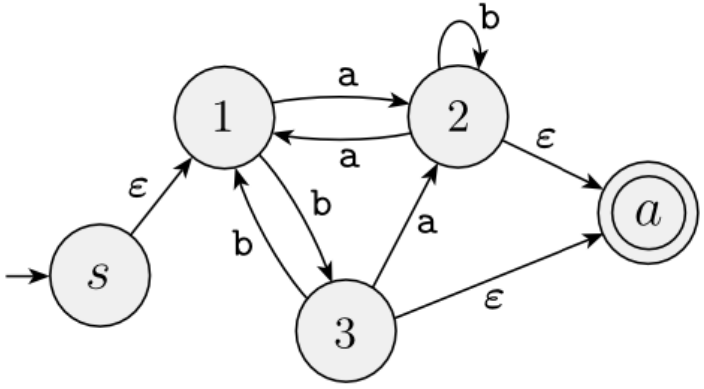
(d)

Example of DFA to RE conversion

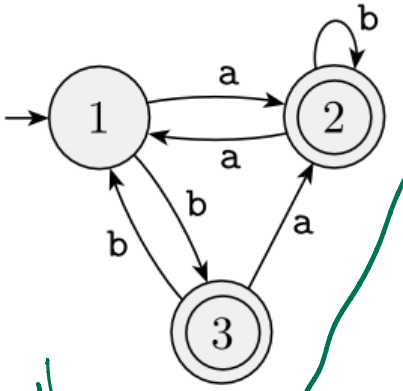


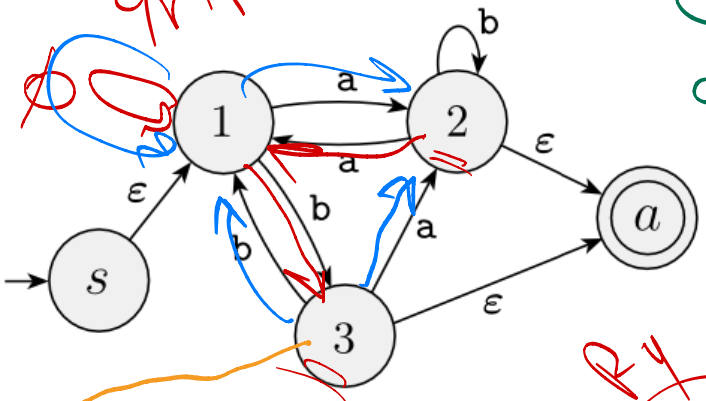
Example of DFA to RE conversion

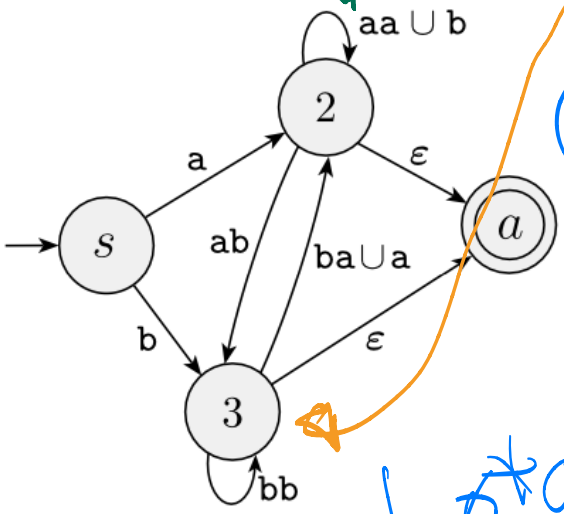


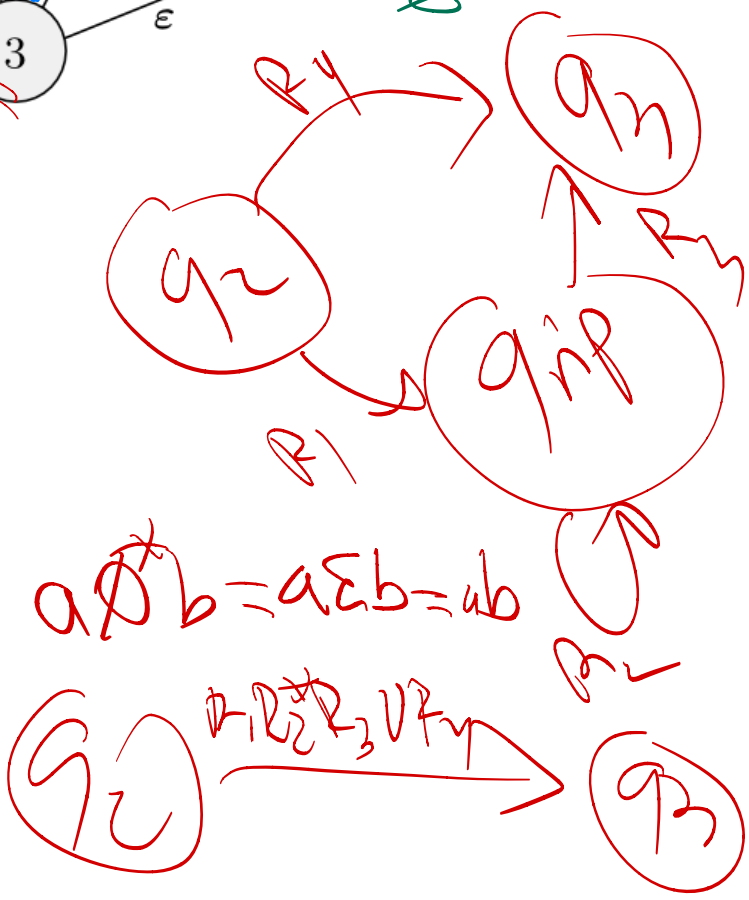


Example of DFA to RE conversion

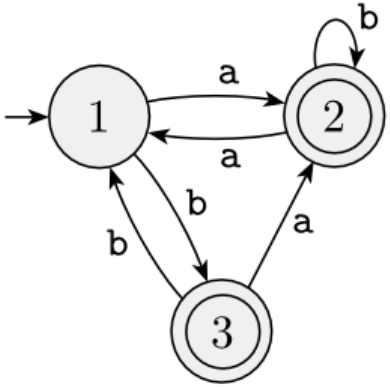


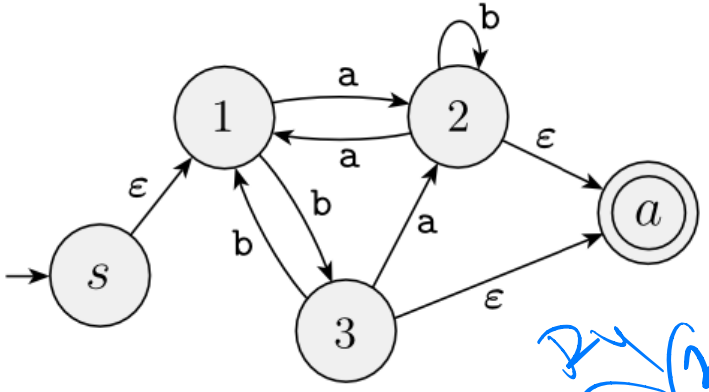


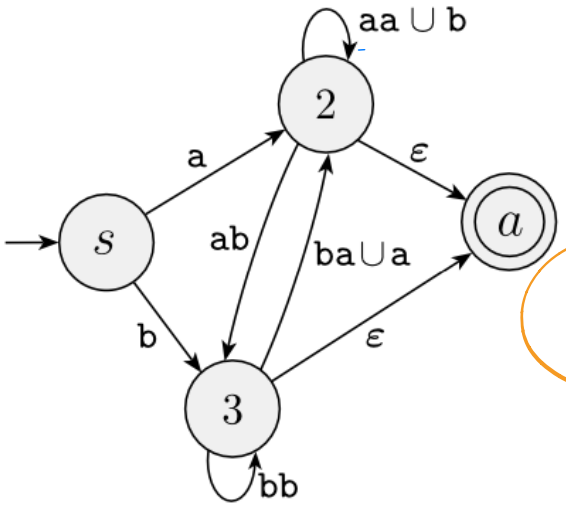


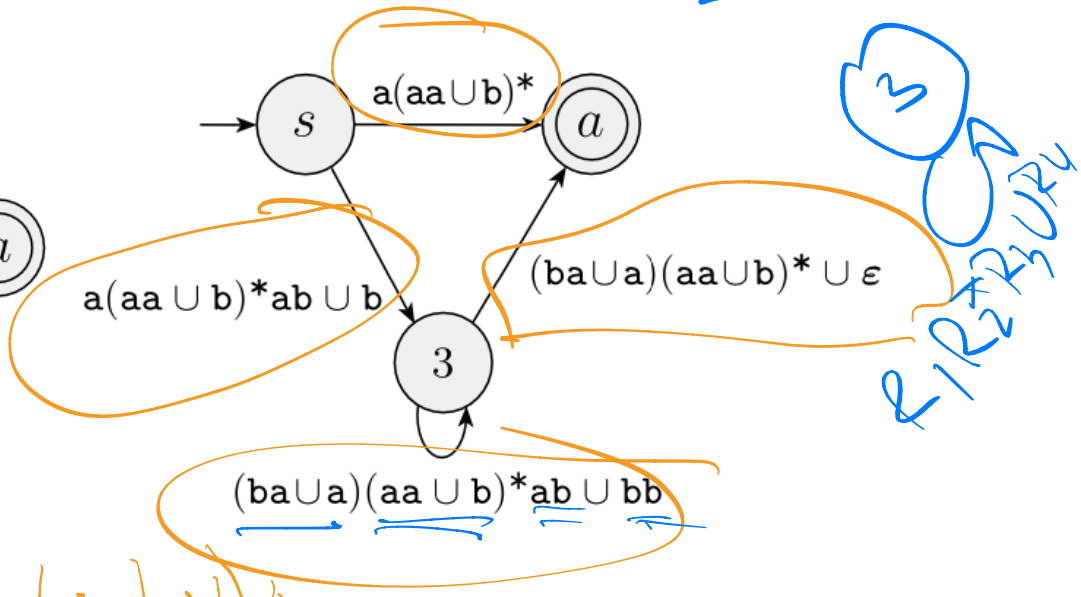


Example of DFA to RE conversion

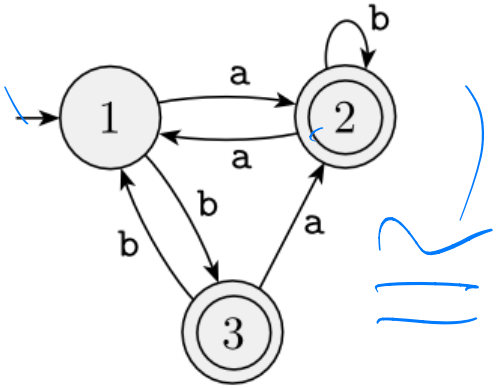


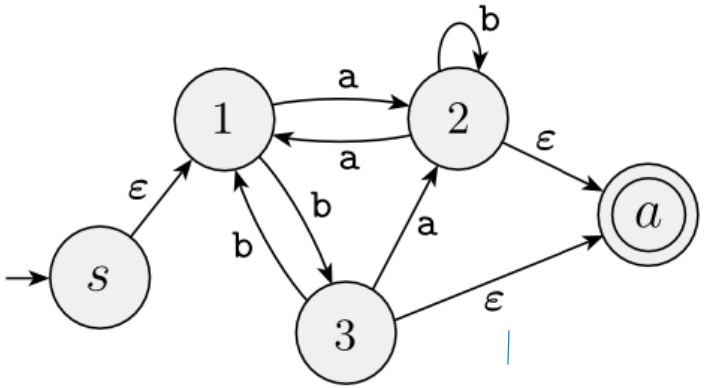


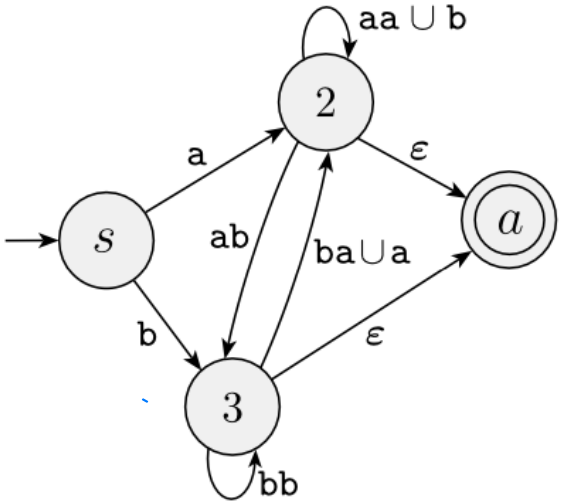


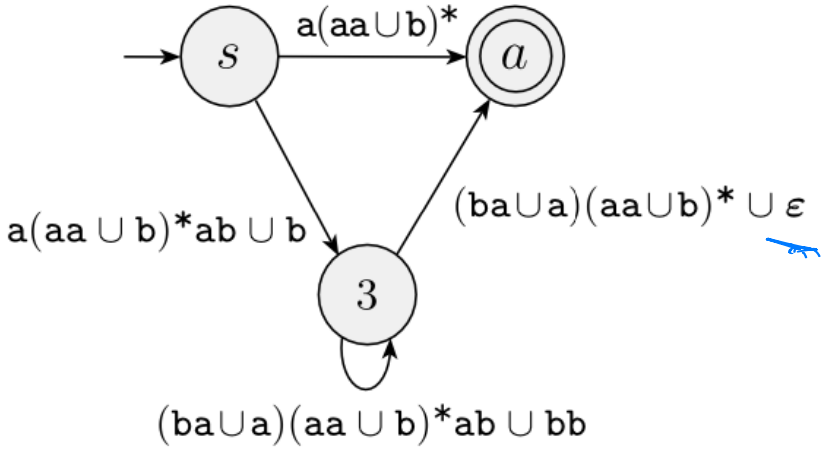


Example of DFA to RE conversion









(a(aa∪b)*ab∪b)((ba∪a)(aa∪b)*ab∪bb)*((ba∪a)(aa∪b)* ∪ε)∪a(aa∪b)*

RE $\iff$ FAs

This completes the two parts of a fairly complicated argument summarized again below.

Theorem

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

Proof.

$\Rightarrow$ Start with an NFA for A and convert into an equivalent GNFA. Remove all states one by one, except the unique start state and the unique accept state. The expression on the arc from the start to the accept state is the regular expression that describes A.

$\Leftarrow$ We showed this using the inductive definition of a regular expression $(a, \epsilon, \emptyset, R_1 \cup R_2, R_1 \circ R_2, R^*)$ to convert the given regular expression for A into an NFA for A. $\square$

What Makes a Language Regular

Consider the languages

- $L = \{0^*1^*\}$
- $L = \{01\}$
- $L = \{0011\}$
- $L = \{\epsilon, 01, 0011\}$
- $L = \{0^n1^n | n \geq 0\}$

Let's try to build FAs for them...

The Pumping Lemma (PL) for Regular Languages

0^*1^* $\quad p = 2$ $\quad 00$, 01, 11, 000

Theorem

*(PL) If A is a regular language, then there exists a number p (*the pumping length*) so that if s is any string in A of length at least p, then s may be divided into three pieces, $s = xyz$, satisfying the following conditions:*

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

$s = 00$ $\quad xyz$ $\quad i=0$ $\quad xy^0z = 0$

$i=1$ $\quad xyz = 00$

$xy^2z = 000$

The Pumping Lemma (PL) for Regular Languages

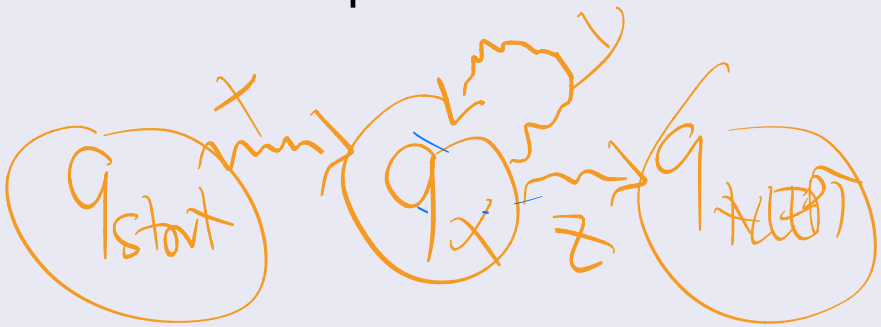
Theorem

If A is regular, then $\exists p$ so that if $s \in A$ and $|s| > p$, then s may be divided into three pieces, $s = xyz$, satisfying the following:

1. $xy^i z \in A$, *for each* $i \geq 0$,
2. $|y| > 0$, *and*
3. $|xy| \leq p$.

Proof.

Let $M = (Q, \Sigma, \delta, q_s, F)$ be a DFA for language A. Then we set $p = |Q|$. Consider some string $s \in A$ such that $|s| > p$. While computing, M must repeat some state (by the pigeonhole principle). Let q_x be the first state that is repeated. Then this picture completes the proof:



Proving a Language Non-Regular

- To prove a language is regular: build a FA or give a RE

Proving a Language Non-Regular

- To prove a language is regular: build a FA or give a RE
- To prove a language is non-regular, we use the PL

Proving a Language Non-Regular

- To prove a language is regular: build a FA or give a RE
- To prove a language is non-regular, we use the PL

Claim

$L = \{0^n 1^n | n \geq 0\}$ *is not a regular language*

Proof.

- Suppose L is regular. Then the PL applies.

Proving a Language Non-Regular

- To prove a language is regular: build a FA or give a RE
- To prove a language is non-regular, we use the PL

Claim

$L = \{0^n 1^n | n \geq 0\}$ *is not a regular language*

Proof.

- Suppose L is regular. Then the PL applies.
- Let p be the pumping length.

Proving a Language Non-Regular

- To prove a language is regular: build a FA or give a RE
- To prove a language is non-regular, we use the PL

Claim

$L = \{0^n 1^n | n \geq 0\}$ *is not a regular language*

Proof.

- Suppose L is regular. Then the PL applies.
- Let p be the pumping length.
- Let's pick a string $s = 0^p 1^p$ from L and longer than p.

Proving a Language Non-Regular

- To prove a language is regular: build a FA or give a RE
- To prove a language is non-regular, we use the PL

Claim

$L = \{0^n 1^n | n \geq 0\}$ *is not a regular language*

Proof.

- Suppose L is regular. Then the PL applies.
- Let p be the pumping length.
- Let's pick a string $s = 0^p 1^p$ from L and longer than p.
- Now we must consider all possible ways to break s into three parts $s = xyz$ subject to PL conditions $|y| > 0$ and $|xy| \leq p$.

Proving a Language Non-Regular

- To prove a language is regular: build a FA or give a RE
- To prove a language is non-regular, we use the PL

Claim

$L = \{0^n 1^n | n \geq 0\}$ *is not a regular language*

Proof.

- Suppose L is regular. Then the PL applies.
- Let p be the pumping length.
- Let's pick a string $s = 0^p 1^p$ from L and longer than p.
- Now we must consider all possible ways to break s into three parts $s = xyz$ subject to PL conditions $|y| > 0$ and $|xy| \leq p$.
- Clearly, y must contain one or more zeros (from conditions 2 and 3).

Proving a Language Non-Regular

- To prove a language is regular: build a FA or give a RE
- To prove a language is non-regular, we use the PL

Claim

$L = \{0^n 1^n | n \geq 0\}$ *is not a regular language*

Proof.

- Suppose L is regular. Then the PL applies.
- Let p be the pumping length.
- Let's pick a string $s = 0^p 1^p$ from L and longer than p.
- Now we must consider all possible ways to break s into three parts $s = xyz$ subject to PL conditions $|y| > 0$ and $|xy| \leq p$.
- Clearly, y must contain one or more zeros (from conditions 2 and 3).
- Condition 1 of the PL requires that $xy^i z \in L$ for each $i \geq 0$.

Proving a Language Non-Regular

- To prove a language is regular: build a FA or give a RE
- To prove a language is non-regular, we use the PL

Claim

$L = \{0^n 1^n | n \geq 0\}$ *is not a regular language*

Proof.

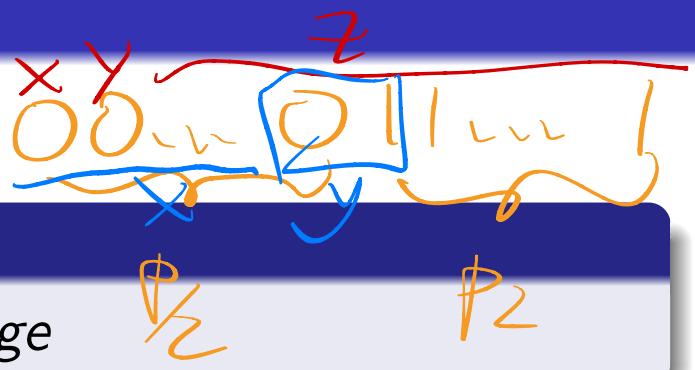
- Suppose L is regular. Then the PL applies.
- Let p be the pumping length.
- Let's pick a string $s = 0^p 1^p$ from L and longer than p.
- Now we must consider all possible ways to break s into three parts $s = xyz$ subject to PL conditions $|y| > 0$ and $|xy| \leq p$.
- Clearly, y must contain one or more zeros (from conditions 2 and 3).
- Condition 1 of the PL requires that $xy^i z \in L$ for each $i \geq 0$.
- But this is not possible: "pumping" y up or down results in strings that are not in L, $xz \notin L$ (fewer 0s than 1s), $xyyz \notin L$ (more 0s than 1s). □

Proving a Language Non-Regular

Use PL to prove a language is non-regular. Think of it as a game:

- I **give** you the language L
- I **give** you a value p
- You **choose** a string s longer than p
- Choose carefully in order to break the claim that s satisfies the 3 conditions of the PL
- Note that you **do not choose** how s is broken into 3 parts $s = xyz$
- You must **argue that no matter how I break** s into 3 parts there is always a contradiction.
- Choosing s poorly can make your job harder or impossible.

Proving a Language Non-Regular



Claim

$L = \{0^n 1^n | n \geq 0\}$ *is not a regular language*

Now, let's choose a different string s.

Proof.

Suppose L is regular. Then the PL applies. Let p be the pumping length. Then consider the string $s = 0^{\lceil p/2 \rceil} 1^{\lceil p/2 \rceil}$ and all possible ways to break s down into $s = xyz$ subject to the 3 PL conditions.

- y contains one or more 0s; then xy^2z contains more 0s than 1s
- y contains one or more 1s; then xy^2z contains more 1s than 0s
- y contains 0s and 1s; then xy^2z alternates b/n 0s and 1s more than once.

□

Administrivia

- Optional Exam #1: Wednesday October 29, 9:00-10:00
- Come to the lecture room a bit early, to find a seat
- Exam questions come from exercise sheets #1 and #2
- Reading: Chapters 1 and 2
- Last time: regular languages, non-regular languages, Pumping Lemma
- Today: non-regular languages, Pumping Lemma, grammars

The Pumping Lemma (PL) for Regular Languages

Theorem

*(PL) If A is a regular language, then there exists a number p (*the pumping length*) so that if s is any string in A of length at least p, then s may be divided into three pieces, $s = xyz$, satisfying the following conditions:*

1. $xy^i z \in A$, *for each* $i \geq 0$,
2. $|y| > 0$, *and*
3. $|xy| \leq p$.

Proving a Language Non-Regular

- To prove a language is regular: build a FA or give a RE
- To prove a language is non-regular, we use the PL

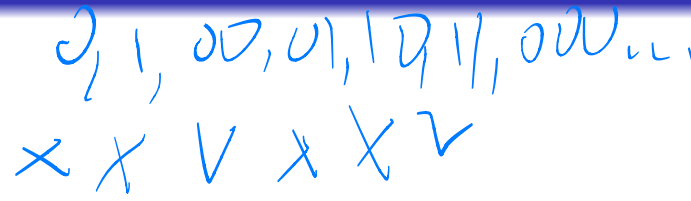
Claim

$L = \{0^n 1^n | n \geq 0\}$ *is not a regular language*

Proof.

- Suppose L is regular. Then the PL applies.
- Let p be the pumping length.
- Let's pick a string $s = 0^p 1^p$ from L and longer than p.
- Now we must consider all possible ways to break s into three parts $s = xyz$ subject to PL conditions $|y| > 0$ and $|xy| \leq p$.
- Clearly, y must contain one or more zeros (from conditions 2 and 3).
- Condition 1 of the PL requires that $xy^i z \in L$ for each $i \geq 0$.
- But this is not possible: "pumping" y up or down results in strings that are not in L, $xz \notin L$ (fewer 0s than 1s), $xyyz \notin L$ (more 0s than 1s). □

Proving a Language Non-Regular



Claim

$L = \{ww | w \in \{0,1\}^*\}$ *is not a regular language*

Proof.

Suppose L is regular. Then the PL applies. Let p be the pumping length. Then consider the string $s = 0^p10^p1$ and all possible ways to break s down into $s = xyz$ subject to the 3 PL conditions. Because of condition 3, y must contain one or more 0s. Then xy^2z has more 0s in on the left side of the first 1 than on the right, which means that $xy^2z \notin L$. □

Again note how the careful choice of s reduces the number of cases we need to consider.

some choices don't work at all 0^p0^p

Proving a Language Non-Regular

0, 1, 00, 01, 10, 11, 000, 001, 010, 011, ...

Claim

$L = \{0^i 1^j | i > j\}$ *is not a regular language*

(0 xyiz ∈ L ∀i = 0, 1, 2, ...

Proof.

Suppose L is regular. Then the PL applies. Let p be the pumping length. Then consider the string $s = 0^{p+1}1^p$ and all possible ways to break s down into $s = xyz$ subject to the 3 PL conditions.

Because of condition 3, y must contain one or more 0s.

What can we say about xy^2z and its membership in L?

Better to consider $xy^0z = xz$, as this string has less than or equal number of 0s than 1s, which means that $xy^0z \notin L$. □

Note that here "pumping up" didn't work, but "pumping down" does indeed work.

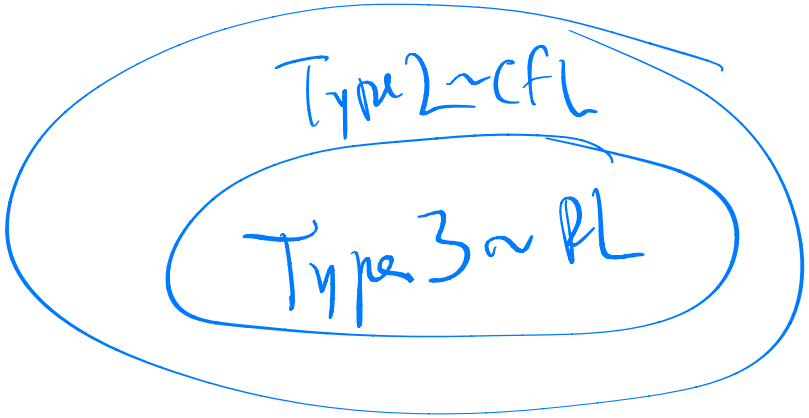
The Pumping Lemma (PL) for Regular Languages

Observe

- the PL is not a necessary and sufficient condition for regularity!
- that is, the pumping lemma is not a complete characterization of regular languages.
- PL: A regular $\Rightarrow p : s \in A$ and $|s| > p \ldots$
- The opposite is not necessarily true

Context Free Languages

- Regular languages are characterized by FAs and REs
- CFLs are characterized by pushdown automata (PDAs) and Context Free Grammars (CFGs)
- Example of a CFL: $L = \{0^n 1^n | n \geq 0\}$
- PDAs are machines with memory (stack)
- Unlike FAs, DPDAs and NPDAs are not equivalent in power
- CFGs are Type 2 grammars in Chomsky's hierarchy



Context Free Grammars

Definition

A context-free grammar is a 4-tuple (V, Σ, R, S), where:

1. V is a finite set of variables,
2. Σ is a finite set of terminals, $\Sigma \cap V = \emptyset$
3. R is a finite set of rules, $\alpha \to \beta$, where α is a variable and β a string of variables and terminals
4. $S \in V$ is the start variable

Example: $G = (\{S\}, \{0, 1\}, R, S)$, where the set of rules R is given by: $S \to 0S1|\epsilon$

Context Free Grammars

Definition

A context-free grammar is a 4-tuple (V, Σ, R, S), where:

1. V is a finite set of variables,
2. Σ is a finite set of terminals, $\Sigma \cap V = \emptyset$
3. R is a finite set of rules, $\alpha \to \beta$, where α is a variable and β a string of variables and terminals
4. $S \in V$ is the start variable

Example: $G = (\{S\}, \{0, 1\}, R, S)$, where the set of rules R is given by: $S \to 0S1|\epsilon$

And the language of G is $L = \{0^n 1^n | n \geq 0\}$

Context Free Grammars

Definition

If u, v, and w are strings of variables and terminals, and $A \to w$ is a rule of the grammar, we say that uAv **yields** uwv, written $uAv \Rightarrow uwv$. We say that u **derives** v, written $u \overset{*}{\Rightarrow} v$, if $u = v$ or if a sequence $u_1, u_2, \ldots, u_k$ exists for $k \geq 0$ and $u_1 \Rightarrow u_2 \Rightarrow \cdots \Rightarrow u_k \Rightarrow v$.

- The **language** of the grammar is $\{w \in \Sigma^* | S \overset{*}{\Rightarrow} w\}$.
- Any language that can be generated by some context-free grammar is called a **context-free language** (CFL).

Example: $S \to 0S1|\epsilon$

equivalent

$S \to 0S1$

$S \to \epsilon$

Context Free Grammars

Definition

If u, v, and w are strings of variables and terminals, and $A \to w$ is a rule of the grammar, we say that uAv **yields** uwv, written $uAv \Rightarrow uwv$. We say that u **derives** v, written $u \overset{*}{\Rightarrow} v$, if $u = v$ or if a sequence $u_1, u_2, \ldots, u_k$ exists for $k \geq 0$ and $u_1 \Rightarrow u_2 \Rightarrow \cdots \Rightarrow u_k \Rightarrow v$.

- The **language** of the grammar is $\{w \in \Sigma^* | S \overset{*}{\Rightarrow} w\}$.
- Any language that can be generated by some context-free grammar is called a **context-free language** (CFL).

Example: $S \to 0S1|\epsilon$

Parsing and parse trees: compilers and interpreters use parsers to extract the meaning of a program before generating the compiled code; the parser can be built, given a CFG.

Context-Free Languages and Grammars

$\langle SENTENCE \rangle \rightarrow \langle NOUN.PHRASE \rangle \langle VERB.PHRASE \rangle$

$\langle NOUN.PHRASE \rangle \rightarrow \langle CMPLX.NOUN \rangle | \langle CMPLX.NOUN \rangle \langle PREP.PHRASE \rangle$

$\langle VERB.PHRASE \rangle \rightarrow \langle CMPLX.VERB \rangle | \langle CMPLX.VERB \rangle \langle PREP.PHRASE \rangle$

$\langle PREP.PHRASE \rangle \rightarrow \langle PREP \rangle \langle CMPLX.NOUN \rangle$

$\langle CMPLX.NOUN \rangle \rightarrow \langle ARTICLE \rangle \langle NOUN \rangle$

$\langle CMPLX.VERB \rangle \rightarrow \langle VERB \rangle | \langle VERB \rangle \langle NOUN.PHRASE \rangle$

$\langle ARTICLE \rangle \rightarrow a | the$

$\langle NOUN \rangle \rightarrow boy | girl | flower$

$\langle VERB \rangle \rightarrow sees | likes$

$\langle PREP \rangle \rightarrow with$

Strings generated by this grammar include:

- The boy sees the girl
- The girl likes the boy with the flower
- A boy sees

$\langle SENTENCE \rangle \Rightarrow \langle NOUN.PHRASE \rangle \langle VERB.PHRASE \rangle \Rightarrow$

$\langle CMPLX.NOUN \rangle \langle VERB.PHRASE \rangle \Rightarrow \langle ARTICLE \rangle \langle NOUN \rangle \langle VERB.PHRASE \rangle \Rightarrow$

$a \langle NOUN \rangle \langle VERB.PHRASE \rangle \Rightarrow a\ boy \langle VERB.PHRASE \rangle \Rightarrow$

$a\ boy \langle CMPLX.VERB \rangle \Rightarrow a\ boy \langle VERB \rangle \Rightarrow a\ boy\ sees$

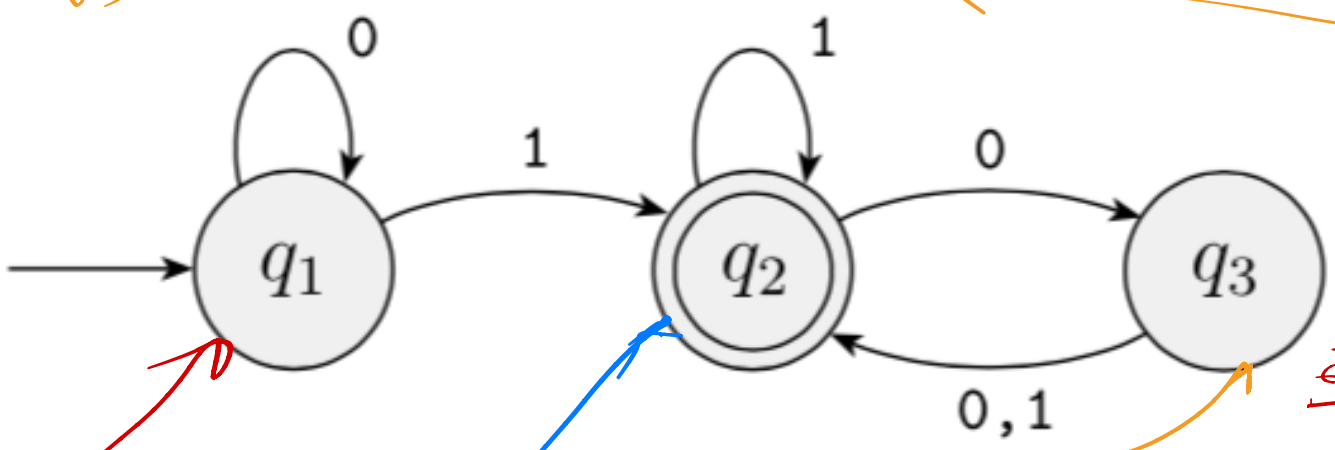
CFLs and CFGs

- Context-free grammars are used to specify the syntax of a language.
- Programming languages such as Python and Java have context-free grammars and they are used for parsing.
- The $O(n^3|G|)$ CYK algorithm uses dynamic programming to check whether a given string (program) can be generated by the context-free grammar for the language.

Grammars

- We have seen two different, though equivalent, methods of describing regular languages: FAs and REs
- A grammar provides yet another way to describe a language
- While an automaton *recognizes* a language, a grammar *generates* the language
- The grammars for regular languages are also known as *type 3 grammars*
- The grammars for context-free languages are context-free grammars, also known as *type 2 grammars*
- There exists a finite automaton for a language $L \iff$ there exists a type 3 grammar for L
- There exists a *pushdown automaton* for a language $L \iff$ there exists a type 2 grammar for L

Type 3 Grammars: Example



$S \to 0S$
$S \to 1A$
$A \to 1A$
$A \to 0B$
$A \to \epsilon$
$B \to 0A$
$B \to 1A$

$S \to 0S|1A$
$A \to 1A|0B|\epsilon$
$B \to 0A|1A$

Designing Grammars

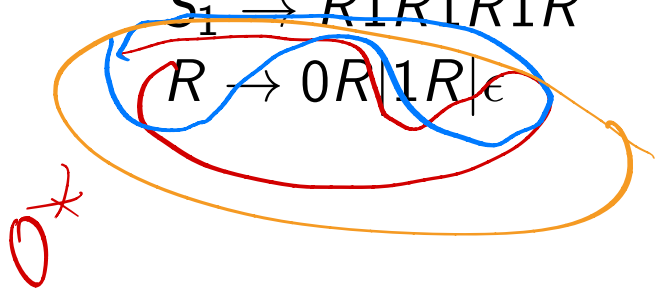
1. $L_1 = \{w | w \text{ contains at least three 1s}\}$

Designing Grammars

1. $L_1 = \{w | w \text{ contains at least three 1s}\}$

$$S_1 \rightarrow R1R1R1R$$

$$R \rightarrow 0R|1R|\epsilon$$

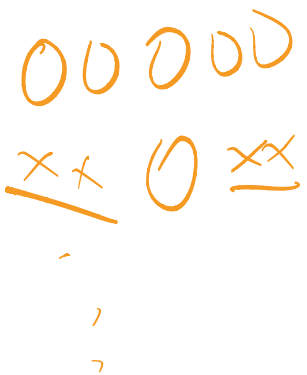


Designing Grammars

1. $L_1 = \{w | w \text{ contains at least three 1s}\}$

   $S_1 \rightarrow R1R1R1R$
   $R \rightarrow 0R|1R|\epsilon$

2. $L_2 = \{w |$ the length of w is odd and its middle symbol is a $0\}$



Designing Grammars

1. $L_1 = \{w | w \text{ contains at least three 1s}\}$

   $S_1 \rightarrow R1R1R1R$
   $R \rightarrow 0R|1R|\epsilon$

2. $L_2 = \{w |$ the length of w is odd and its middle symbol is a $0\}$

   $S_2 \rightarrow 0|0S0|0S1|1S0|1S1$

3. L_3 is strings over the alphabet $\{a, b\}$ with more a's than b's

Designing Grammars

1. $L_1 = \{w | w \text{ contains at least three 1s}\}$

   $S_1 \to R1R1R1R$
   $R \to 0R|1R|\epsilon$

2. $L_2 = \{w |$ the length of w is odd and its middle symbol is a $0\}$

   $S_2 \to 0|0S0|0S1|1S0|1S1$

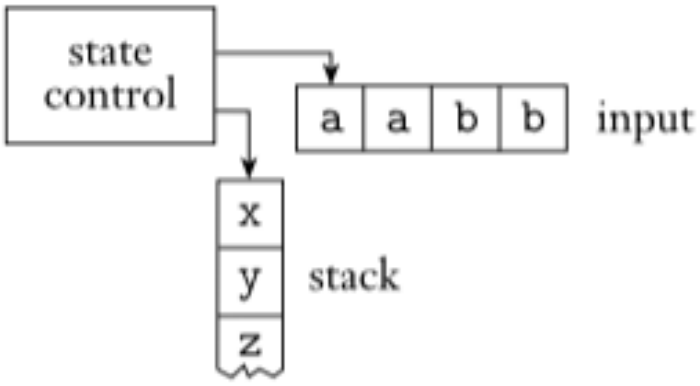
3. L_3 is strings over the alphabet $\{a, b\}$ with more a's than b's

   $S_3 \xrightarrow{1} TaT$
   $T \to TT|aTb|bTa|a|\epsilon$
   2 3 4 5 6

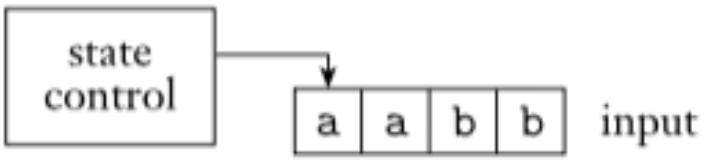
baa
$S_3 \xrightarrow{1} TaT \xrightarrow{4} bTaaT \to$
$\xrightarrow{6} baaT \xrightarrow{6} baa$

Pushdown Automata

Schematic Pushdown Automaton

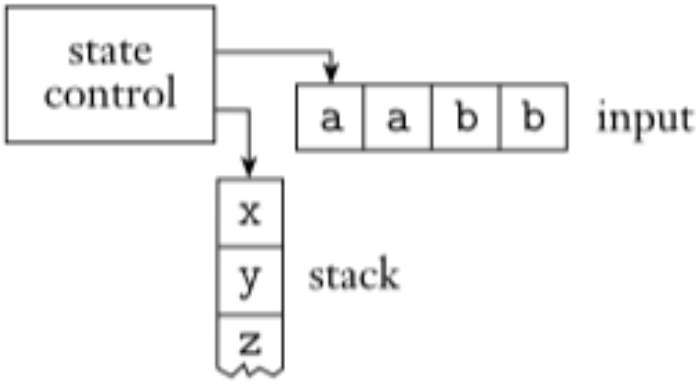


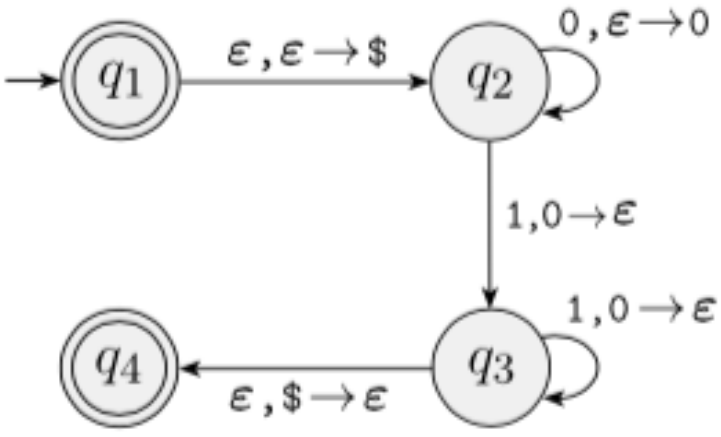
Schematic Finite Automaton



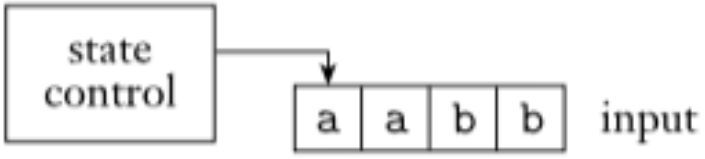
Pushdown Automata

Schematic Pushdown Automaton



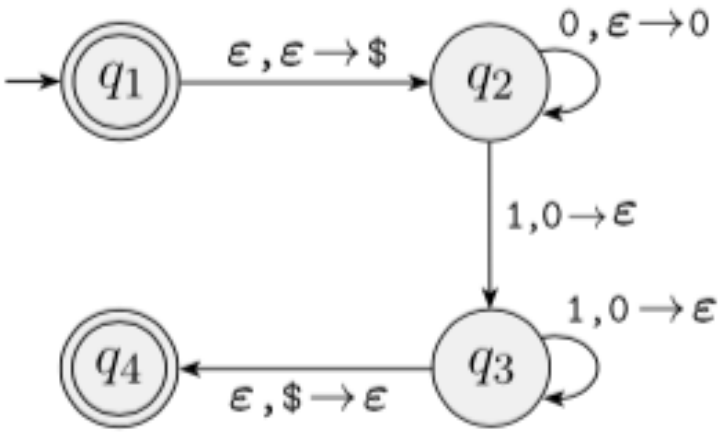


Schematic Finite Automaton



$\{0^n 1^n \mid n \geq 0\}$

PDA



A pushdown automaton is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_s, F)$, where

- Q is a finite set called the states,
- Σ is a finite input alphabet,
- Γ is a finite stack alphabet,
- $\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \to P(Q \times \Gamma_\epsilon)$ is the transition function,
- $q_s \in Q$ is the start state,
- $F \subseteq Q$ is the set of accept states.

Administrivia

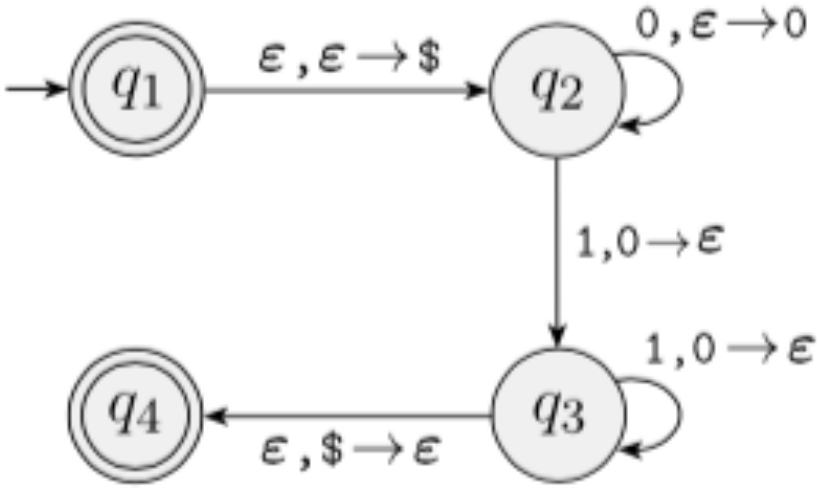
- Optional Exam #1: grades posted on Wednesday, 48 hours to review and comment
- Exercise sheet #3 assigned last Wednesday
- Reading: Chapter 2
- Last time: Pumping Lemma, Context Free Languages, Grammars
- Today: Push-down Automata, Grammars, Normal Forms

Context-Free Languages

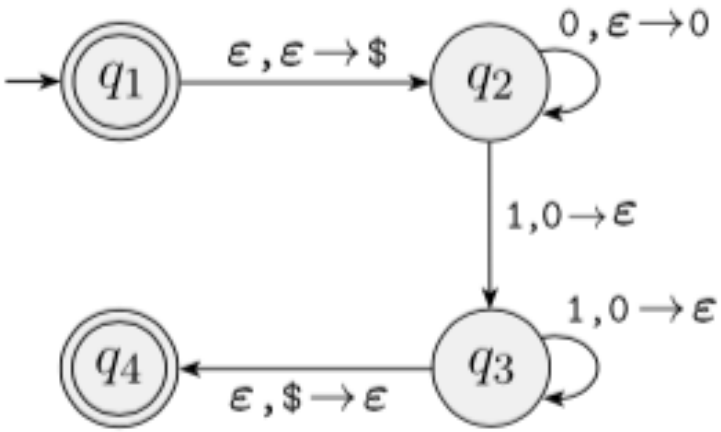
- Regular languages are characterized by FAs and REs
- CFLs are characterized by **Context-Free Grammars (CFGs)**
- CFLs are characterized by **pushdown automata (PDAs)**
- PDAs are machines with memory (stack)
- Example of a CFL: $L = \{0^n 1^n | n \geq 0\}$
- Unlike FAs, DPDAs and NPDAs are not equivalent in power
- CFGs are Type 2 grammars in Chomsky's hierarchy

$\{0^n 1^{2n} \mid n \geq 0\}$

for every 0, push 2 0s on stack



PDA



A pushdown automaton is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_s, F)$, where

- Q is a finite set called the states,
- Σ is a finite input alphabet,
- Γ is a finite stack alphabet,
- $\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \to P(Q \times \Gamma_\epsilon)$ is the transition function,
- $q_s \in Q$ is the start state,
- $F \subseteq Q$ is the set of accept states.

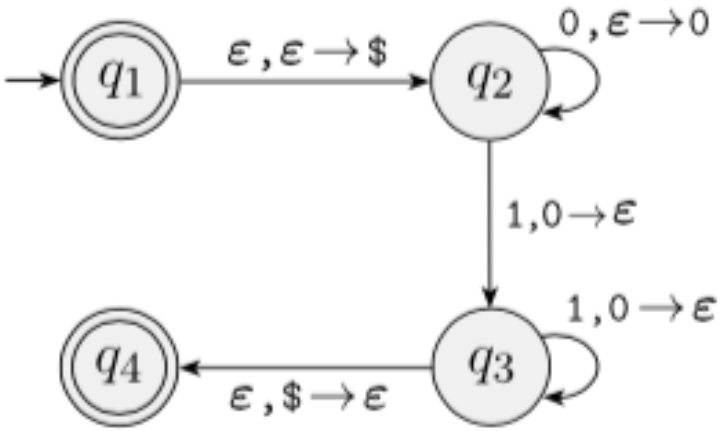
Computation of a Pushdown Automaton

Let $M = (Q, \Sigma, \Gamma, \delta, q_S, F)$ be a pushdown automaton and let $w = w_1 w_2 \dots w_n$ be a string where each w_i is a member of the alphabet Σ_ϵ. Then M accepts w if there exists a sequence of states $r_0, r_1, \dots, r_n$ in Q and a sequence of strings $s_0, s_1, \dots, s_n$ in Γ_ϵ that satisfy the three conditions:

- $r_0 = q_S$ and $s_0 = \epsilon$
- $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$, for $i = 0, ..., n-1$, where $s_i = at$ and $s_{i+1} = bt$ for some $a, b \in \Gamma_\epsilon$ and $t \in \Gamma^*$.
- $r_n \in F$.

A language is called a **context free language** if some PDA recognizes it.

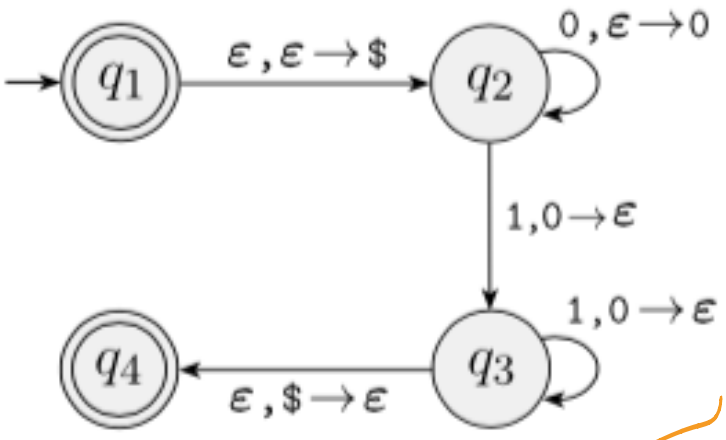
PDA Technicalities



The formal PDA definition has no explicit mechanism to test for:

- **Empty stack**. We accomplish this by initially placing a special symbol \$ on the stack. When we see this on top of the stack we know that the stack is empty. This gives us the ability to "test for an empty stack."
- **End of input string**. We assume that the accept state takes effect only when the machine is at the end of the input, i.e., if the PDA is an accept state but the input has not yet been completely processed, the PDA does not yet accept.

PDA Technicalities



"$q, b \to b$" on a, b b on top of stack, leave b on top of stack

input character
top of stack
write on top of stack

The many meanings of $a, b \to c$: on a from input, pop b, push c:

- $a = \epsilon$: don't read from input
- $b = \epsilon$: don't pop top of stack
- $c = \epsilon$: don't push anything on top of stack
- $a, b \to \epsilon$: on a, pop b, (don't push)
- $a, \epsilon \to c$: on a (don't pop), push c

Let's Build a PDA for a Language

Consider the language $L = \{a^i b^j c^k | i, j, k \geq 0 \text{ and } i = j \text{ or } i = k\}$

✓ ab

✗ aa

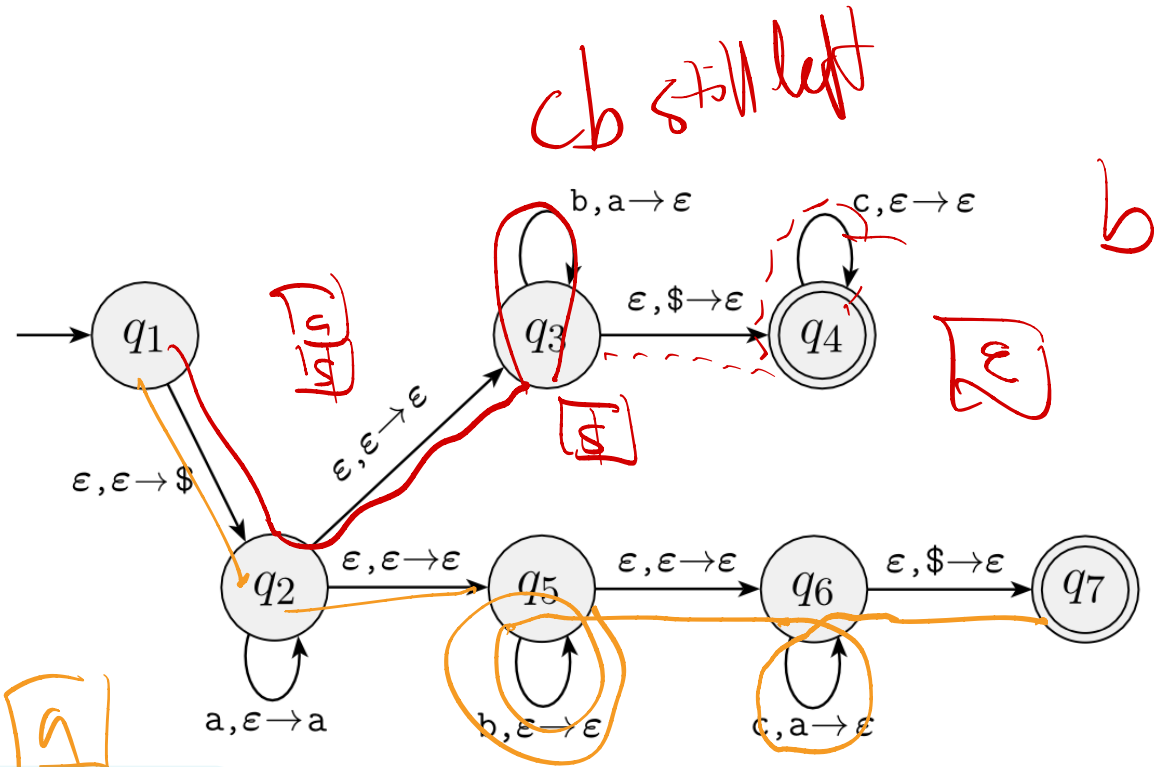
✓ ac

bc ✗

abc ✓✓

Let's Build a PDA for a Language

Consider the language $L = \{a^i b^j c^k | i, j, k \geq 0 \text{ and } i = j \text{ or } i = k\}$



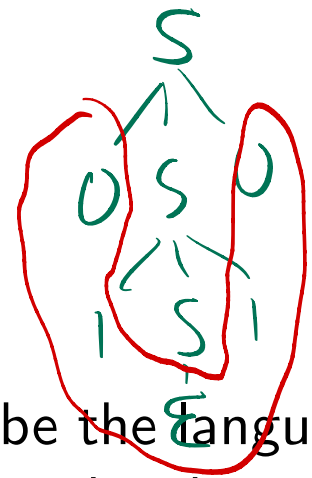
Recall Parse Trees

The parse trees for a grammar G are trees that must meet the following conditions:

$S \to 0S0 \to 01S10 \to 0110$

- each internal node v is labeled by a variable in $V \in G$
- each leaf node is labeled by either a terminal in Σ or ϵ; if a leaf is labeled with ϵ it is the only child of its parent
- if an internal node is labeled A and its children are $X_1, X_2, \ldots X_k$, then $A \to X_1 X_2 \ldots X_k$ is a rule in G

$S \to \epsilon|0|1$
$S \to 0S0|1S1$



Another way to describe the language of a grammar is as the set of yields of the parse trees that have the start symbol as root and a terminal string as the yield.

Context Free Grammars and Normal Forms

Definition

A context-free grammar is a 4-tuple (V, Σ, R, S), where:

1. V is a finite set of variables,
2. Σ is a finite set of terminals, $\Sigma \cap V = \emptyset$
3. R is a finite set of rules, $\alpha \to \beta$, where α is a variable and β a string of variables and terminals
4. $S \in V$ is the start variable

Example

1. $S \to ASA$
2. $S \to aB$
3. $A \to B$
4. $A \to S$
5. $B \to b$
6. $B \to \epsilon$

$S \xrightarrow{1} ASA \xrightarrow{1} AASAA \xrightarrow{3}$
$BASAA \xrightarrow{2} BBSAA \xrightarrow{3}$
$BBSBA \xrightarrow{3} BBSBB \xrightarrow{2}$
$BBaBBB \xrightarrow{6} BBaBB \xrightarrow{6}$
$BaBB \xrightarrow{6} BaB \xrightarrow{6} aB \xrightarrow{5} a$

Normal Forms for Type 2 Grammars

Simplified (standardized) rules such as those in the Chomsky Normal Form or the Greibach Normal Form, provide some nice properties of the derivations:

- the size of the generated string is proportional to the number of rules applied.
- Chomsky Normal Form (1959) makes it possible to use a polynomial time algorithm to decide whether a string can be generated by a grammar; see the Cocke-Younger-Kasami (CYK) algorithm.
- Greibach Normal Form (1965) makes it possible to prove that every CFG can be recognized by a PDA that works in real time (i.e., without ϵ transitions).

$A \to BC$
$A \to a$
$S \to \epsilon$ (if $\epsilon \in L$)

$A \to aA_1A_2 \ldots A_k, k \geq 0$
$S \to \epsilon$ (if $\epsilon \in L$)

Chomsky Normal Form

Definition

A context-free grammar is in Chomsky normal form if every rule is of the form:

$A \to BC$

$A \to a$

where a is any terminal and A, B, and C are any variables, except that B and C may not be the start variable. In addition, we permit the rule $S \to \epsilon$ if $\epsilon \in L$, where S is the start variable.

Clearly CNF and GNF are much more restricted than the initial definition of Context Free Grammars. So it is not obvious that they are as powerful (i.e., recognize the same class of languages)...

CFGs in CNF

Why do we care to force CFGs in a special form such as CNF?

- rules of the type $A \to B$ are confusing: are we making any progress towards generating terminals?
- there could be loops in the derivation: $A \to B$, $B \to C$, $C \to A$
- rules such as $A \to \epsilon$ allow us to generate very long strings of variables and then erase them all
- Without "unit production rules" and without ϵ rules, every step in the derivation makes demonstrable progress toward the terminal string: either the string gets longer or a terminal appears.
- as you'll see in the exercises, deriving a word length n in the language takes exactly $2n - 1$ steps.
- finally, the parse tree is binary tree!

Any CFL has a CFG in CNF

Theorem

Any context-free language can be generated by a context-free grammar in Chomsky normal form.

Proof.

Let L be a CFL. Then there exists a type 2 grammar G for it. We show how to convert G into Chomsky normal form. The conversion has several stages wherein rules that violate the conditions are replaced with equivalent ones that are satisfactory.

1. add a new start variable.
2. eliminate all ϵ-rules of the form $A \to \epsilon$.
3. eliminate all unit rules of the form $A \to B$.
4. convert the remaining rules into the proper form.

with extra wrinkles

□

CFG to CNF Example

Phase 1 (new start)

valid CFG

$S \to ASA|aB$
$A \to B|S$
$B \to b|\epsilon$

$\mathbf{S_0} \to \mathbf{S}$
$S \to ASA|aB$
$A \to B|S$
$B \to b|\epsilon$

Phase 2 (remove $B \to \epsilon$)

$S_0 \to S$
$S \to ASA|aB$
$A \to B|S$
$B \to b|\epsilon$

$S_0 \to S$
$S \to ASA|aB|\mathbf{a}$
$A \to B|S|\epsilon$
$B \to b$

Phase 2 (remove $A \to \epsilon$)

$S_0 \to S$
$S \to ASA|aB|a$
$A \to B|S|\epsilon$
$B \to b$

$S_0 \to S$
$S \to ASA|aB|a|\mathbf{SA}|\mathbf{AS}|\mathbf{S}$
$A \to B|S$
$B \to b$

CFG to CNF Example

Phase 3 (remove $S \to S$)

$S_0 \to S$
S $\to ASA|aB|a|SA|AS|S$
A $\to B|S$
B $\to b$

$S_0 \to S$
S $\to ASA|aB|a|SA|AS$
A $\to B|S$
B $\to b$

Phase 3 (remove $S_0 \to S$)

$S_0 \to S$
S $\to ASA|aB|a|SA|AS$
A $\to B|S$
B $\to b$

$S_0 \to \mathbf{ASA|aB|a|SA|AS}$
S $\to ASA|aB|a|SA|AS$
A $\to B|S$
B $\to b$

CFG to CNF Example

Phase 3 (remove $A \to B$)

$S_0 \to ASA|aB|a|SA|AS$
S $\to ASA|aB|a|SA|AS$
A $\to B|S$
B $\to b$

$S_0 \to ASA|aB|a|SA|AS$
S $\to ASA|aB|a|SA|AS$
A $\to S|\mathbf{b}$
B $\to b$

Phase 3 (remove $A \to S$)

$S_0 \to ASA|aB|a|SA|AS$
S $\to ASA|aB|a|SA|AS$
A $\to S|b$
B $\to b$

$S_0 \to ASA|aB|a|SA|AS$
S $\to ASA|aB|a|SA|AS$
A $\to b|\mathbf{ASA}|\mathbf{aB}|\mathbf{a}|\mathbf{SA}|\mathbf{AS}$
B $\to b$

CFG to CNF Example

Phase 4 (fix remaining rules)

$S_0 \to ASA|aB|a|SA|AS$
$S \to ASA|aB|a|SA|AS$
$A \to b|ASA|aB|a|SA|AS$
$B \to b$

$S_0 \to AA_1|UB|a|SA|AS$
$S \to AA_1|UB|a|SA|AS$
$A \to b|AA_1|UB|a|SA|AS$
$A_1 \to SA$
$U \to a$
$B \to b$

Ambiguity: Syntactic, Semantic, etc.

Context Free Languages can contain ambiguous sentences:

- Squad helps dog bite victim

Ambiguity: Syntactic, Semantic, etc.

Context Free Languages can contain ambiguous sentences:

- Squad helps dog bite victim
- MBA studies mushroom

Ambiguity: Syntactic, Semantic, etc.

Context Free Languages can contain ambiguous sentences:

- Squad helps dog bite victim
- MBA studies mushroom
- Enraged cow injures farmer with ax

Ambiguity: Syntactic, Semantic, etc.

Context Free Languages can contain ambiguous sentences:

- Squad helps dog bite victim
- MBA studies mushroom
- Enraged cow injures farmer with ax
- Kids make nutritious snacks

Ambiguity: Syntactic, Semantic, etc.

Context Free Languages can contain ambiguous sentences:

- Squad helps dog bite victim
- MBA studies mushroom
- Enraged cow injures farmer with ax
- Kids make nutritious snacks
- Hershey bars protest

Ambiguity: Syntactic, Semantic, etc.

Context Free Languages can contain ambiguous sentences:

- Squad helps dog bite victim
- MBA studies mushroom
- Enraged cow injures farmer with ax
- Kids make nutritious snacks
- Hershey bars protest
- Prostitutes appeal to pope

Ambiguity: Syntactic, Semantic, etc.

Context Free Languages can contain ambiguous sentences:

- Squad helps dog bite victim
- MBA studies mushroom
- Enraged cow injures farmer with ax
- Kids make nutritious snacks
- Hershey bars protest
- Prostitutes appeal to pope
- I never said she stole my money

syntactic

semantic

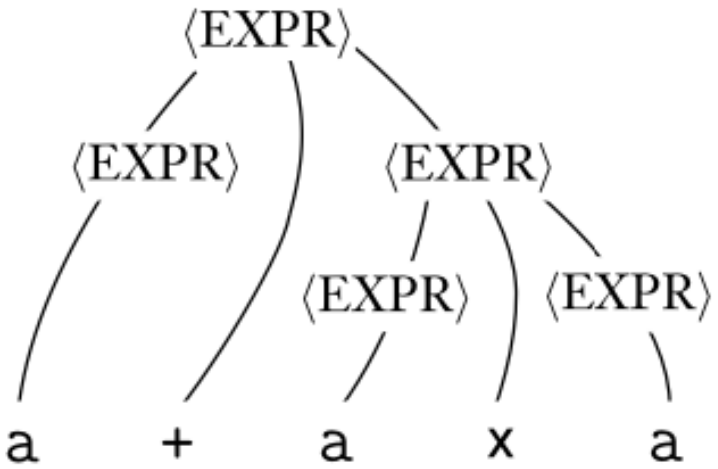
Ambiguity

Definition

A string w is derived ambiguously in context-free grammar G if it has two or more different leftmost derivations. Grammar G is ambiguous if it generates some string ambiguously.

Consider the following grammar:

$< EXPR > \rightarrow < EXPR > + < EXPR > \mid < EXPR > x < EXPR > \mid (< EXPR >) \mid a$



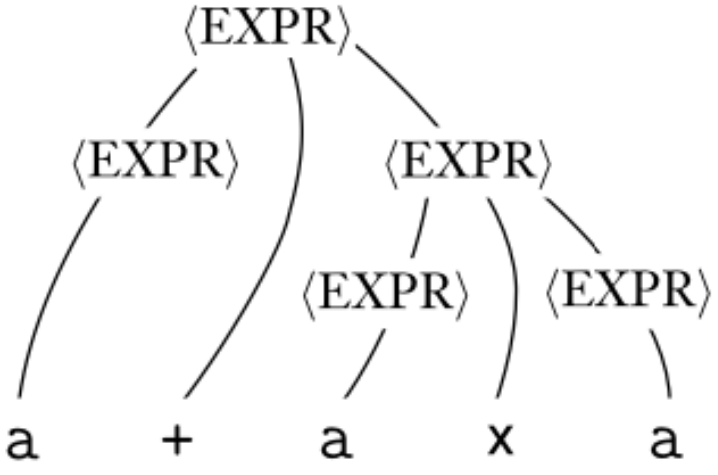
Ambiguity

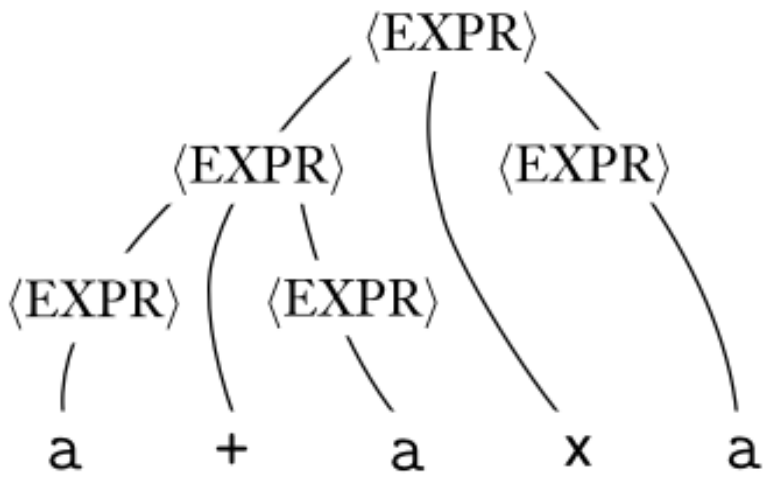
Definition

A string w is derived ambiguously in context-free grammar G if it has two or more different leftmost derivations. Grammar G is ambiguous if it generates some string ambiguously.

Consider the following grammar:

$< EXPR > \rightarrow < EXPR > + < EXPR > \mid < EXPR > x < EXPR > \mid (< EXPR >) \mid a$





Languages Recognized by PDAs and Generated by CFGs

Theorem

A language is context free if and only if some pushdown automaton recognizes it.

Proof.

Recall that by definition, a language is CF if some CFG generates it. Then we need to show that the class of CFLs is equivalent to the class of languages recognized by PDAs.

- $\Rightarrow$ If G is a CFG for language A, then there exists PDA P that recognizes A
- $\Leftarrow$ If PDA P recognizes a language A, then there exists a CFG G that generates A

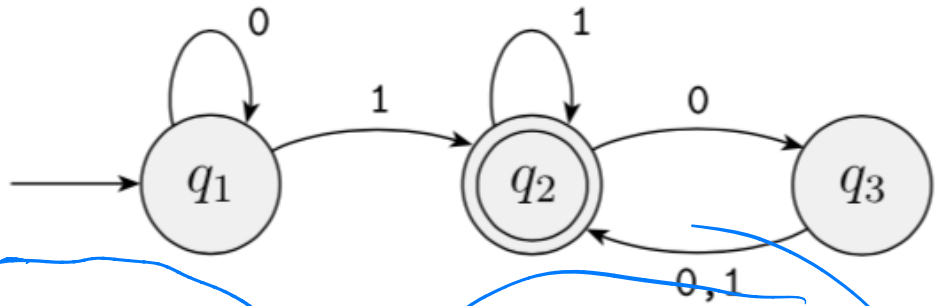
$\square$

Administrivia

- average 30.6
- pass 32.5

- Optional Exam #1: grades posted, comment by 6pm Thursday
- Exercise sheet #4 assigned today, due next Wednesday
- Optional Exam #2: Wednesday, November 12, 9:00-10:00
- Reading: Chapter 2, 3
- Last time: Context Free Languages, Push-down Automata, Grammars, Normal Forms
- Today: PDAs $\Longleftrightarrow$ CFGs, Chomsky hierarchy, non-CFL languages

Type 3 Grammars



$S \to 0S|1A$
$A \to 1A|0B|\epsilon$
$B \to 0A|1A$

Definition

A grammar for a regular language is a type3 right-linear grammar if every rule is of the form:

$A \to aB$

$A \to a$

where a is any terminal, and A and B are any variables. If ϵ is in the language than $S \to \epsilon$ is a rule, where S is the start variable.

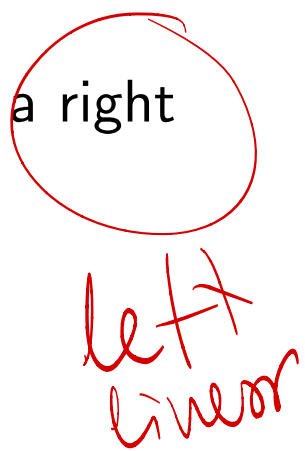
But what about the rule $A \to \epsilon$ in the grammar above?

Type 3 Grammars and Regular Languages

Consider the grammar:
$S \to aA|a$
$A \to aA|a|bB|b$
$B \to bB|b$

What makes this a type 3 grammar?

- Chomsky's hierarchy: type 0, type 1, type 2, type 3
- the most general rule is $\alpha \to \beta$, where α and β are strings of terminals and variables
- rule restrictions: $\alpha \in V$, $|\alpha| = 1$, $|\beta| \leq 2$, etc.
- e.g., the grammar above it not only type 3 but also a right linear grammar



Chomsky's Language Hierarchy (1956)

There are 4 languages (4 grammar types) in the hierarchy:

- Type 3 grammars (regular languages):
  $\alpha \to \beta$
  $\alpha \in V$ and $\beta \in \Sigma V \cup \Sigma$ ($S \to \epsilon$ if $\epsilon \in L$)
- Type 2 grammars (context-free languages):
  $\alpha \to \beta$
  $\alpha \in V$ and $\beta \in (\Sigma \cup V)^*$
- Type 1 grammars (context-sensitive languages):
  $\alpha A \beta \to \alpha \gamma \beta$
  $A \in V$, $\alpha, \beta \in (\Sigma \cup V)^*$, $\gamma \in (\Sigma \cup V)^+$
- Type 0 grammars (unrestricted languages):
  $\alpha \to \beta$
  $\alpha, \beta \in (\Sigma \cup V)^*$

Type 3 Grammars and Regular Languages

Theorem

A language is regular if and only if a Type3 grammar generates it.

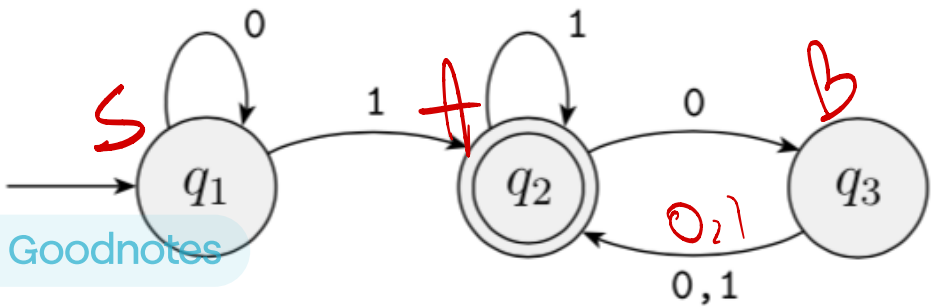
Proof.

$\Rightarrow$ Let $M = \{Q, \Sigma, \delta, q_S, F\}$ be the DFA for the language. We use M to construct the grammar $G = (V, \Sigma_G, R, S)$ as follows.

- $\Sigma_G = \Sigma$
- $V = Q$
- $S = q_0$
- $R = \begin{cases} q_i \to aq_j, & \delta(q_i, a) \to q_j \\ q \to \epsilon, & q \in F \end{cases}$

final answer

$S \to 1A \mid 0S \mid 1$
$A \to 0B \mid 1A \mid 1$
$B \to 0A \mid 1A \mid 0 \mid 1$



$S \to 1A \mid 0S$
$A \to 0B \mid 1A \mid \varepsilon$
$B \to 0A \mid 1A$

Type 3 Grammars and Regular Languages

Theorem

A language is regular if and only if a Type3 grammar generates it.

Proof.

$\Leftarrow$ Let $G = (V, \Sigma_G, R, S)$ be a Type3 grammar. We construct equivalent NFA $N = \{Q, \Sigma, \delta, q_S, F\}$ from it as follows.

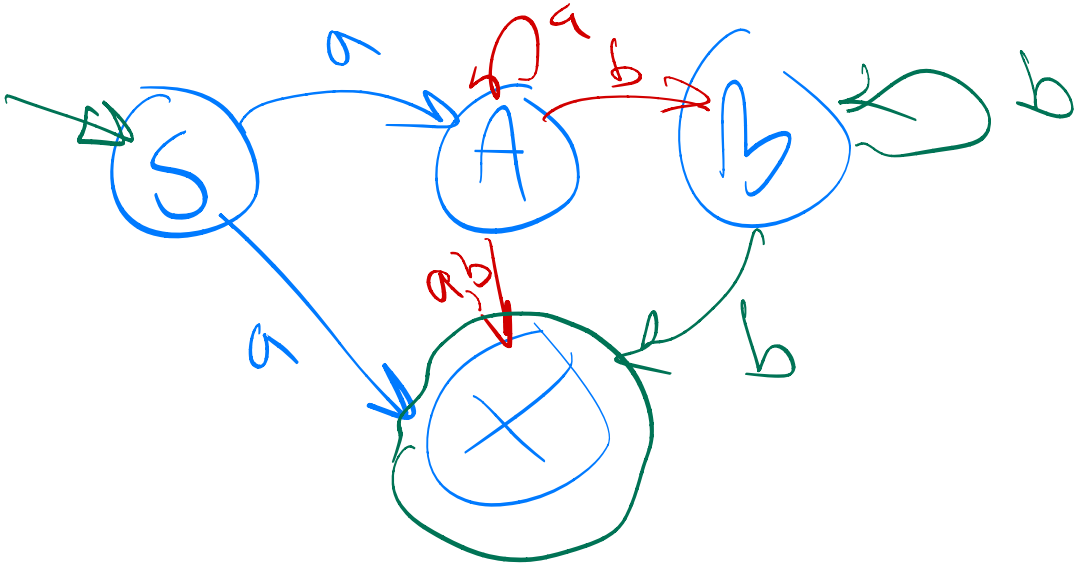
- $\Sigma = \Sigma_G$
- $Q = V \cup X$, where X is not already in V
- $q_0 = S$
- $F = \{X\}$ (or $F = \{X \cup S\}$ when $S \to \epsilon$ is a rule of G)
- $\delta \begin{cases} B \in \delta(A, a), & \text{if } A \to aB \in R \\ X \in \delta(A, a), & \text{if } A \to a \in R \end{cases}$

□

Example grammar to NFA conversion:

Example grammar to NFA conversion

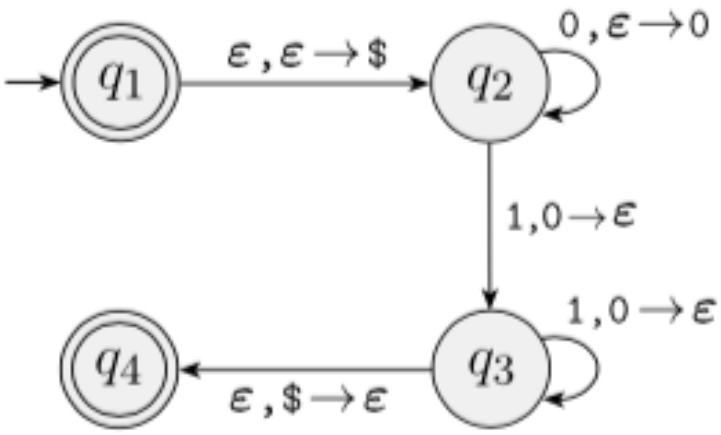
$\Sigma = \{a, b\}$



$$S \to aA|a$$
$$A \to aA|a|bB|b$$
$$B \to bB|b$$

$$\delta \begin{cases} B \in \delta(A, a), & \text{if } A \to aB \in R \\ X \in \delta(A, a), & \text{if } A \to a \in R \end{cases}$$

PDA $\iff$ CFG



$S \to 0S1|\epsilon$

Languages Recognized by PDAs and Generated by CFGs

Theorem

A language is context free if and only if some pushdown automaton recognizes it.

Proof.

Recall that by definition, a language is CF if some CFG generates it. Then we need to show that the class of CFLs is equivalent to the class of languages recognized by PDAs.

- $\Rightarrow$ If G is a CFG for language A, then there exists PDA P that recognizes A
- $\Leftarrow$ If PDA P recognizes a language A, then there exists a CFG G that generates A

$\square$

CFG $G \Rightarrow$ PDA P

Lemma

Let G be a CFG for language A. Then there exists PDA P that recognizes A.

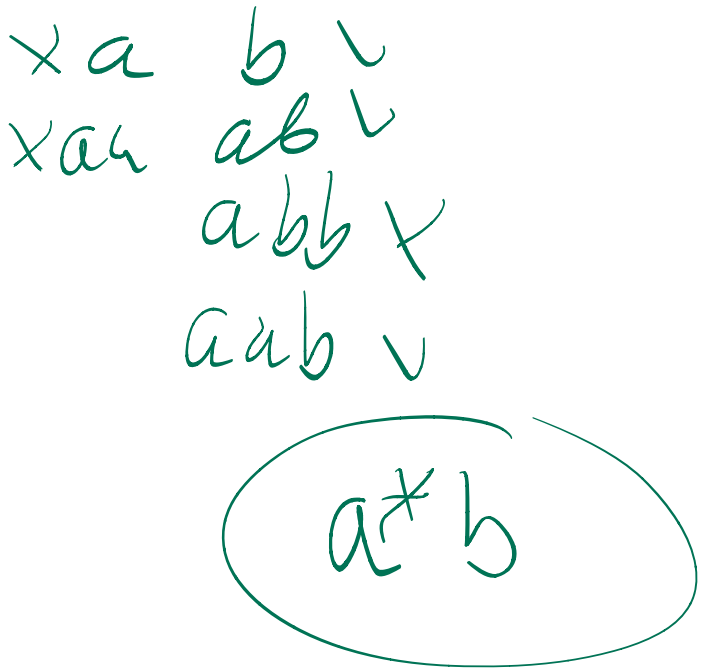
Proof.

(Rough sketch) We construct a PDA P that accepts its input w, if G generates that input, by determining whether there exists a valid derivation for w in G. A derivation is a sequence of substitutions made as a grammar generates a string. Each step of the derivation yields an intermediate string of variables and terminals. P determines whether some series of substitutions using the rules of G can lead from the start variable to w. But which of many possible derivations do we try? The PDA's nondeterminism allows it to "guess" the sequence of correct substitutions. □

CFG $G \Rightarrow$ PDA P

Consider the CFG G below; what is $L(G)$?

$S \rightarrow aTb|b$
$T \rightarrow Ta|\epsilon$



CFG $G \Rightarrow$ PDA P

Consider the CFG G below; what is $L(G)$?

$S \rightarrow aTb|b$
$T \rightarrow Ta|\epsilon$

Create a PDA using the idea from the proof sketch:

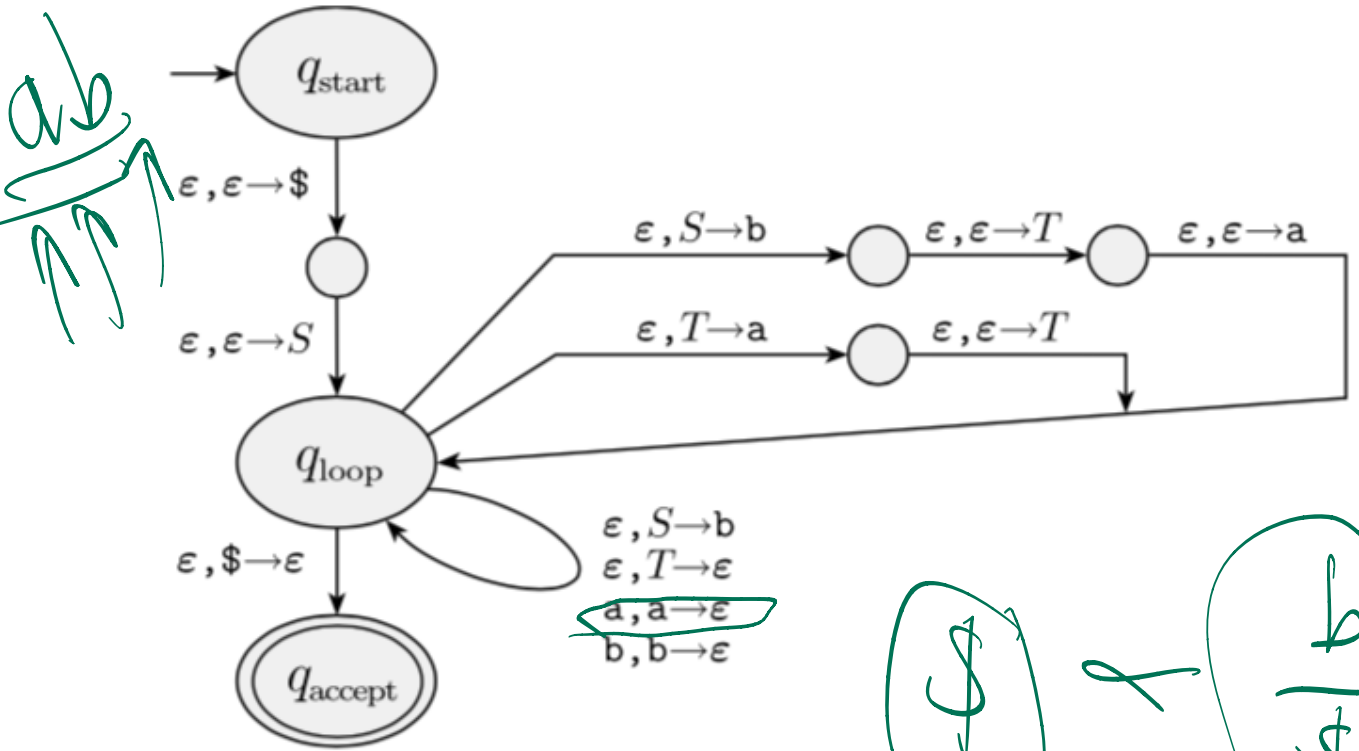
CFG $G \Rightarrow$ PDA P

Consider the CFG G below; what is $L(G)$?

$S \to aTb \mid b$

$T \to Ta \mid \epsilon$

Create a PDA using the idea from the proof sketch:

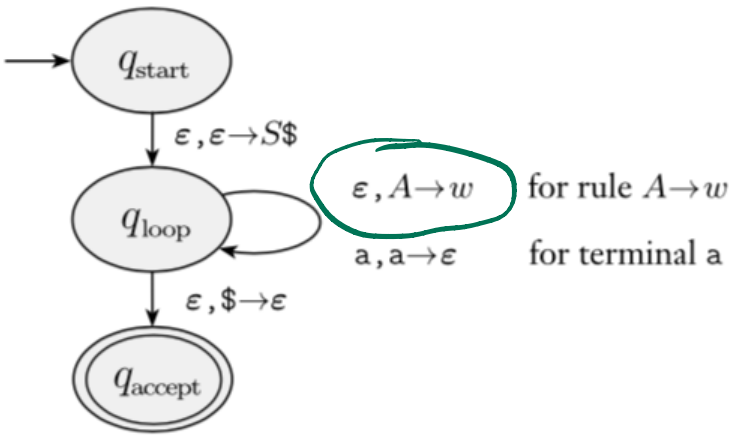


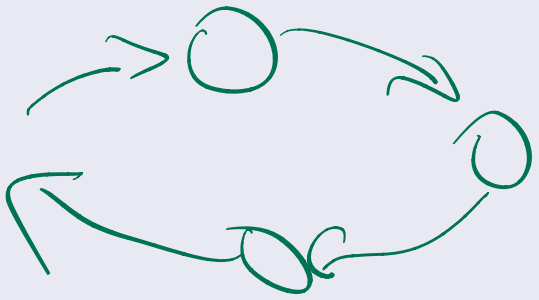
CFG $G \Rightarrow$ PDA P

Proof.

We construct $P = (Q, \Sigma, \Gamma, \delta, q_s, F)$ from G as follows. Let $(r, u) \in \delta(q, a, s)$ mean that when P is in state q, with a as the next input symbol and symbol s on top of the stack, P reads a, pops s, pushes the string u onto the stack and goes to state r. Then $Q = \{q_s, q_{loop}, q_{accept}\} \cup E$, where E is the set of states needed for the $(r, u) \in \delta(q, a, s)$ trick.

The transition function is shown in this figure:





□

PDA $P \Rightarrow$ CFG G

Lemma

Let PDA P recognize a language A. Then there exists a CFG G that generates A.

Proof.

(Rough sketch) We design G such that G generates a string if that string causes the PDA to go from its start state to an accept state. For each pair of states p and q in P, the grammar will have a variable A_{pq}. This variable generates all the strings that can take P from p with an empty stack to q with an empty stack. We further simplify P to have the following three features.

1. P has a single accept state.
2. P empties its stack before accepting.
3. Each transition either pushes a symbol onto the stack (a push move) or pops one off the stack (a pop move), but it does not do both at the same time.

PDA $P \Rightarrow$ CFG G

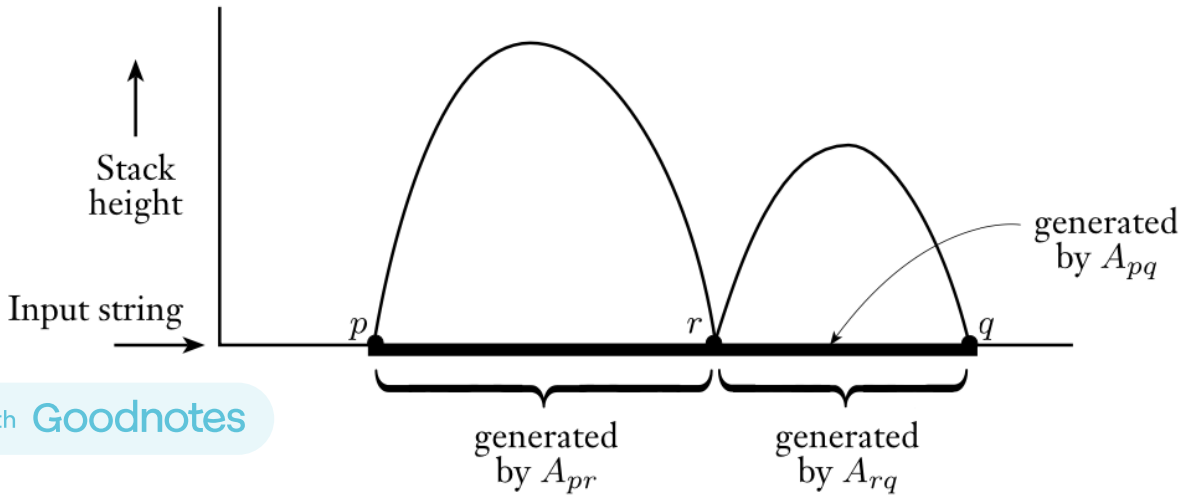
Lemma

Let PDA P recognize a language A. Then there exists a CFG G that generates A.

Proof.

Let $P = (Q, \Sigma, \Gamma, \delta, q_s, q_a)$. We construct G from P as follows.
The variables of G are $\{A_{pq} | p, q \in Q\}$. The start variable is A_{q_s, q_a} and the rules of G are:

1. For each $p, q, r, s \in Q$, $u \in \Gamma$, and $a, b \in \Sigma_\epsilon$, if $\delta(p, a, \epsilon)$ contains (r, u) and $\delta(s, b, u)$ contains (q, ϵ), put the rule $A_{pq} \rightarrow aA_{rs}b$ in G



PDA $P \Rightarrow$ CFG G

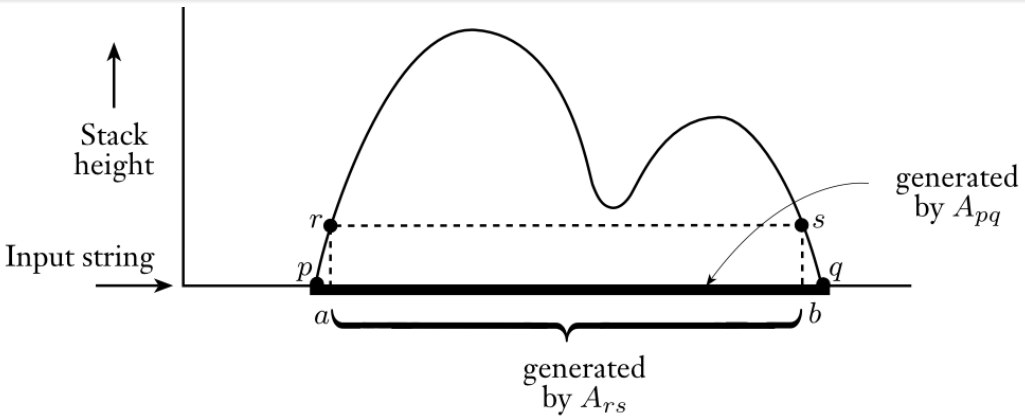
Lemma

Let PDA P recognize a language A. Then there exists a CFG G that generates A.

Proof.

Let $P = (Q, \Sigma, \Gamma, \delta, q_s, q_a)$. We construct G from P as follows.
The variables of G are $\{A_{pq} | p, q \in Q\}$. The start variable is A_{q_s, q_a} and the rules of G are:

1. For each $p, q, r, s \in Q$, $u \in \Gamma$, and $a, b \in \Sigma_\epsilon$, if $\delta(p, a, \epsilon)$ contains (r, u) and $\delta(s, b, u)$ contains (q, ϵ), put the rule $A_{pq} \rightarrow aA_{rs}b$ in G
2. For each $p, q, r \in Q$, put the rule $A_{pq} \rightarrow A_{pr}A_{rq}$ in G



PDA $P \Rightarrow$ CFG G

Lemma

Let PDA P recognize a language A. Then there exists a CFG G that generates A.

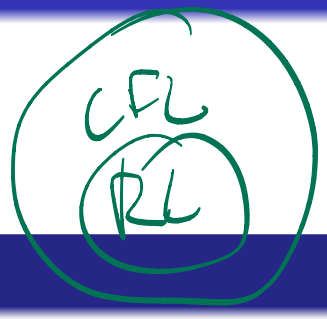
Proof.

Let $P = (Q, \Sigma, \Gamma, \delta, q_s, q_a)$. We construct G from P as follows.
The variables of G are $\{A_{pq} | p, q \in Q\}$. The start variable is A_{q_s, q_a} and the rules of G are:

1. For each $p, q, r, s \in Q, u \in \Gamma$, and $a, b \in \Sigma_\epsilon$, if $\delta(p, a, \epsilon)$ contains (r, u) and $\delta(s, b, u)$ contains (q, ϵ), put the rule $A_{pq} \rightarrow aA_{rs}b$ in G
2. For each $p, q, r \in Q$, put the rule $A_{pq} \rightarrow A_{pr}A_{rq}$ in G
3. For each $p \in Q$, put the rule $A_{pp} \rightarrow \epsilon$ in G

Then we need a very careful proof by induction that if A_{pq} generates x, then x can bring P from p with empty stack to q with empty stack.

RL $\subset$ *CFL*



Theorem

Every regular language is also a context free language.

Proof.

- By definition, a language is context free if there exists a PDA for it.
- By definition, a language is regular if there exists an NFA for it.
- And an NFA is a (special type of) PDA that does not use its stack.

□

The Pumping Lemma (PL) for Context Free Languages

Theorem

*(PL) If A is a context free language, then there exists a number p (*the pumping length*) so that if s is any string in A of length at least p, then s may be divided into five pieces, $s = uvxyz$, satisfying the following conditions:*

1. $uv^i xy^i z \in A$, *for each* $i \geq 0$,
2. $|vy| > 0$, *and*
3. $|vxy| \leq p$.

The Pumping Lemma (PL) for Context Free Languages

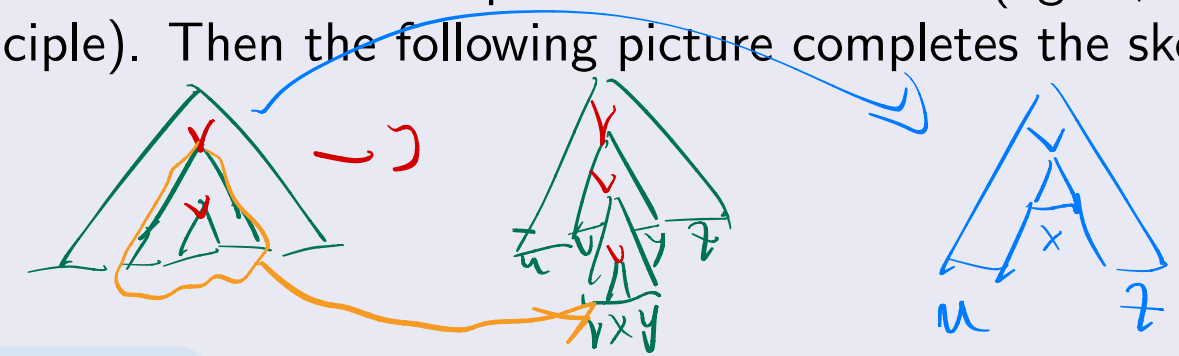
Theorem

If A is context free, then $\exists p$ so that if $s \in A$ and $|s| > p$, then s may be divided into five pieces, $s = uvxyz$, to satisfy:

1. $uv^i xy^i z \in A$, *for each* $i \geq 0$,
2. $|vy| > 0$, *and*
3. $|vxy| \leq p$.

Proof.

(Sketch) Let G be a CFG for CFL A. Then we set $p = |V|$, where V are the variables of G. Consider some long string $s \in A$: long enough that in its derivation we must repeat some variable R (again, by the pigeonhole principle). Then the following picture completes the sketch:



□

The Pumping Lemma (PL) for Context Free Languages

Proof.

Let G be a CFG for CFL A. Let $b \geq 2$ be the maximum number of RHS symbols in any rule. Any node in a parse tree has $\leq b$ children. The depth of the tree is proportional to this branching factor b: At depth d from the root, there are at most b^d leaves, i.e., the longest string that can be generated after d rules of the grammar has size $\leq b^d$. Then if a generated string is at least $b^d + 1$ long, its parse tree must be at least $d + 1$ deep.

If $|V|$ is the number of variables in G, let $p = b^{|V|+1}$. If $s \in A$ and $|s| \geq p$, then its parse tree must have depth $\geq |V| + 1$, as $b^{|V|+1} \geq b^{|V|} + 1$.

Now consider a parse tree T for s (if more than one, choose the one with fewest nodes). T must be at least $|V| + 1$ deep, so its longest root-to-leaf path has length at least $|V| + 1$. But this path has at least $|V| + 2$ nodes and only the leaf is a terminal, with the others variables. Then some variable R appears more than once on that path. If more than one repeat, choose R to be a variable that repeats among the lowest $|V| + 1$ variables on this path. □

Proving a Language Non-Context-Free

- To prove a language is context-free we build a PDA, or a CFG
- To prove a language is not context-free, we use the PL

Claim

$L = \{0^n 1^n 0^n | n \geq 0\}$ *is not a context-free language*

First, let's get some intuition:

Proving a Language Non-Context-Free

- To prove a language is context-free we build a PDA, or a CFG
- To prove a language is not context-free, we use the PL

Claim

$L = \{0^n 1^n 0^n | n \geq 0\}$ *is not a context-free language*

First, let's get some intuition:

- Can we use the stack to match 0s and 1s and then 1s and 0s?

Proving a Language Non-Context-Free

- To prove a language is context-free we build a PDA, or a CFG
- To prove a language is not context-free, we use the PL

Claim

$L = \{0^n 1^n 0^n | n \geq 0\}$ *is not a context-free language*

First, let's get some intuition:

- Can we use the stack to match 0s and 1s and then 1s and 0s?
- How about we push two 0s for each 0 and then match the first n 0s to n 1s and then the remaining 0s to the trailing 0s?

Proving a Language Non-Context-Free

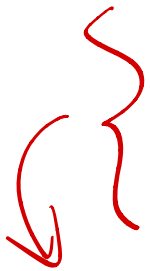
- To prove a language is context-free we build a PDA, or a CFG
- To prove a language is not context-free, we use the PL

Claim

$L = \{0^n 1^n 0^n | n \geq 0\}$ *is not a context-free language*

First, let's get some intuition:

- Can we use the stack to match 0s and 1s and then 1s and 0s?
- How about we push two 0s for each 0 and then match the first n 0s to n 1s and then the remaining 0s to the trailing 0s?
- What language would such a PDA recognize?



Proving a Language Non-Context-Free

Claim

$L = \{0^n 1^n 0^n | n \geq 0\}$ *is not a context-free language*

Proof.

Suppose L is context-free. Then the PL applies. Let p be the pumping length. Then consider the string $s = 0^p 1^p 0^p$ and all possible ways to break s down into $s = uvxyz$ subject to the 3 PL conditions.

- Consider all cases depending on what v and y might contain
- Show that in all of these cases, pumping up results in a string that is not in L.

□

Proving a Language Non-Context-Free

Claim

$L = \{ww | w \in \{0,1\}^*\}$ is not a context-free language.

- Recall that $L = \{ww^R | w \in \{0,1\}^*\}$ is indeed a CFL
- The PDA memory restriction makes it easy to recognize ww^R
- Why not ww?

Proof.

- Let's try to use the PL to show that L is not a CFL

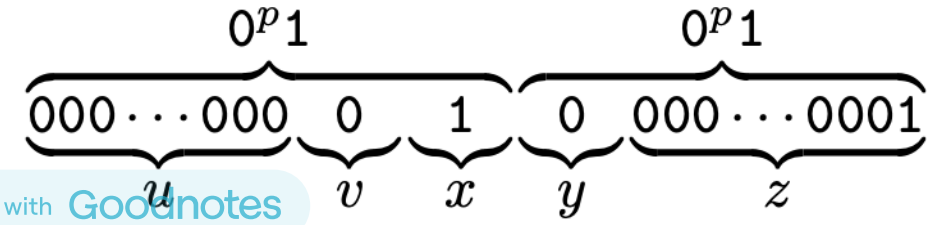
Proving a Language Non-Context-Free

Claim

$L = \{ww | w \in \{0,1\}^*\}$ is not a context-free language.

Proof.

- Assume L is a CFL and obtain a contradiction with the PL
- Let p be the pumping length given by the pumping lemma
- We now get to choose a string $s \in L$ of length $\geq p$
- How about $s = 0^p 1 0^p 1$?
- Clearly, $s \in L$ and has length greater than p, so it appears to be a good candidate
- However, s **can** be pumped by dividing it as follows!



Proving a Language Non-Context-Free

Claim

$L = \{ww | w \in \{0,1\}^*\}$ is not a context-free language.

Proof.

- Assume L is a CFL and obtain a contradiction with the PL
- Let p be the pumping length given by the pumping lemma
- We choose $s = 0^p1^p0^p1^p$
- Clearly, $s \in L$ and has length greater than p
- Condition 3 of the PL restricts the ways that s can be divided ($s = uvxyz$, where $|vxy| \leq p$)
- if the substring xyz occurs only in the first 0s of s, pumping s up to uv^2xy^2z creates more 0s on the left than on the right
- if the substring xyz occurs only in the first 1s of s, pumping s up to uv^2xy^2z creates more 1s on the left than on the right
- symmetrically if substring xyz is on the right with only 0s or only 1s

Proving a Language Non-Context-Free

Claim

$L = \{ww | w \in \{0,1\}^*\}$ *is not a context-free language.*

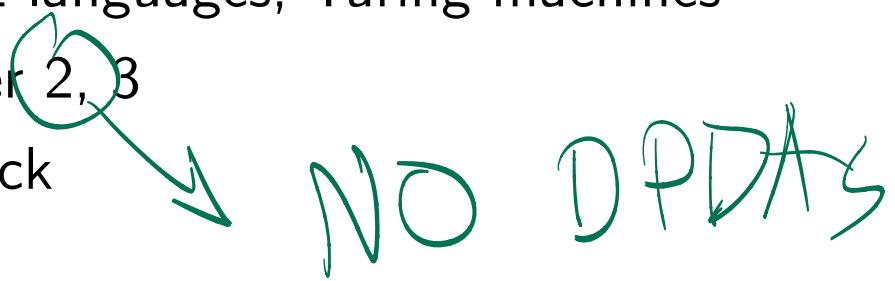
Proof.

- if the substring xyz contains 0s and 1s in the first half of s, pumping s up to uv^2xy^2z creates a different string on the left than on the right (not ww)
- if the substring xyz contains 1s and 0s and straddles the middle of s, then uv^2xy^2z creates a different number of 0s and 1s in the left and right parts

□

Administrivia

- Optional Exam #2: Wednesday, November 12, 9:00-10:00
- Exam will cover exercise sheets #3 and #4
- Last time: CFLs, Type2 and Type3 grammars, PDAs, pumping lemma for CFLs
- Today: Non-CFL languages, Turing machines
- Reading: Chapter 2, 3
- Feedback feedback



The Pumping Lemma (PL) for Context Free Languages

Theorem

*(PL) If A is a context free language, then there exists a number p (*the pumping length*) so that if s is any string in A of length at least p, then s may be divided into five pieces, $s = uvxyz$, satisfying the following conditions:*

1. $uv^i xy^i z \in A$, *for each* $i \geq 0$,
2. $|vy| > 0$, *and*
3. $|vxy| \leq p$.

Proving a Language Non-Context-Free

Claim

$L = \{0^n 1^n 0^n | n \geq 0\}$ *is not a context-free language*

Claim

$L = \{ww | w \in \{0,1\}^*\}$ *is not a context-free language.*

- Recall that $L = \{ww^R | w \in \{0,1\}^*\}$ is indeed a CFL
- The PDA memory restriction makes it easy to recognize ww^R
- Why not ww?

001 000 ✗
001 001 ✓
✗ 00000
✗ 001100

Proof.

- Let's try to use the PL to show that L is not a CFL

Proving a Language Non-Context-Free

Claim

$L = \{ww | w \in \{0,1\}^*\}$ is not a context-free language.

Proof.

- Assume L is a CFL and obtain a contradiction with the PL
- Let p be the pumping length given by the pumping lemma
- We now get to choose a string $s \in L$ of length $\geq p$
- How about $s = 0^p 10^p 1$?
- Clearly, $s \in L$ and is longer than p, so it seems a good candidate

Proving a Language Non-Context-Free

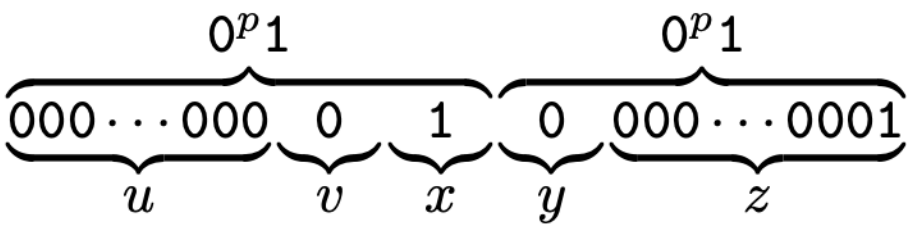
Claim

$L = \{ww | w \in \{0,1\}^*\}$ *is not a context-free language.*

Proof.

- Assume L is a CFL and obtain a contradiction with the PL
- Let p be the pumping length given by the pumping lemma
- We now get to choose a string $s \in L$ of length $\geq p$
- How about $s = 0^p 10^p 1$?
- Clearly, $s \in L$ and is longer than p, so it seems a good candidate
- However, s **can** be pumped by dividing it as follows!

BAD CHOICE FOR S

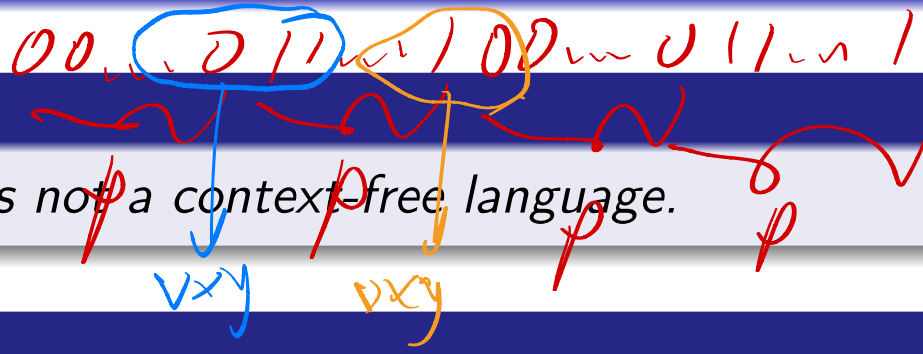


$uv^3xy^3z = 0^{p+2}1 0^{p+2} \notin L$

$uv^0xy^0z = 0^{p-1}1 0^{p-1} \notin L$

$uv^2xy^2z = 0^{p+1}1 0^{p+1} \notin L$

Proving a Language Non-Context-Free



Claim

$L = \{ww | w \in \{0,1\}^*\}$ *is not a context-free language.*

Proof.

- Assume L is a CFL and obtain a contradiction with the PL
- Let p be the pumping length given by the pumping lemma
- We choose $s = 0^p1^p0^p1^p$
- Clearly, $s \in L$ and has length greater than p
- Condition 3 of the PL restricts the ways that s can be divided ($s = uvxyz$, where $|vxy| \leq p$)
- if the substring xyz occurs only in the first 0s of s, pumping s up to uv^2xy^2z creates more 0s on the left than on the right
- if the substring xyz occurs only in the first 1s of s, pumping s up to uv^2xy^2z creates more 1s on the left than on the right
- symmetrically if substring xyz is on the right with only 0s or only 1s

Proving a Language Non-Context-Free

Claim

$L = \{ww | w \in \{0,1\}^*\}$ *is not a context-free language.*

Proof.

- if the substring xyz contains 0s and 1s in the first half of s, pumping s up to uv^2xy^2z creates a different string on the left than on the right (not ww)
- if the substring xyz contains 1s and 0s and straddles the middle of s, then uv^2xy^2z creates a different number of 0s and 1s in the left and right parts

□

What Do We Know?

Automata (FA, PDA), grammars (type3, type2), and languages (regular, context-free)

- Regular languages, e.g., 0^*1^*, numbers divisible by 2,3,...,
- Context-free languages, e.g., 0^n1^n, ww^R, programming languages
- not context-free, e.g., $0^n1^n0^n$, ww, ...

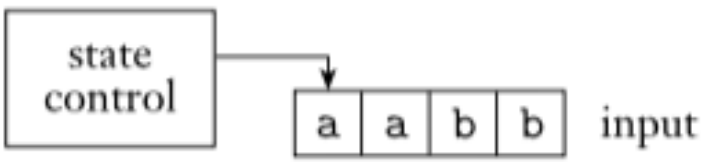
What is next?

- Turing machines
- Date back to 1930s, before any computers
- Naturally, Alan Turing didn't call them "Turing Machines" but "automatic machines"
- A spectacular paper ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO THE ENTSCHEIDUNGSPROBLEM

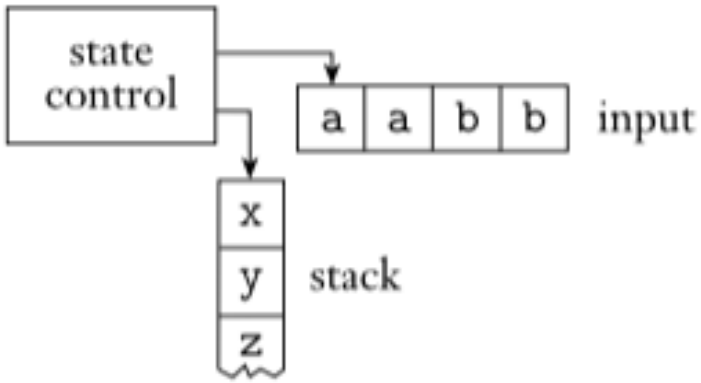
Automata Schematics

Recall the first two types of automata:

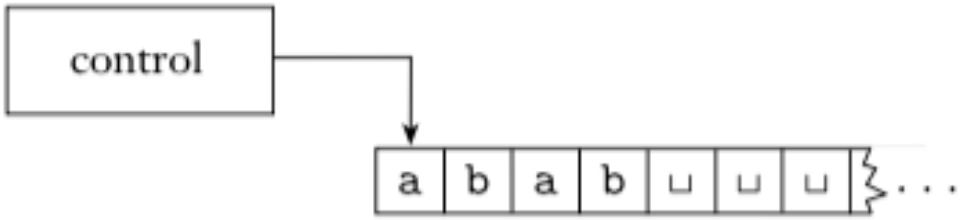
Schematic Finite Automaton



Schematic Pushdown Automaton



Schematic Turing Machine

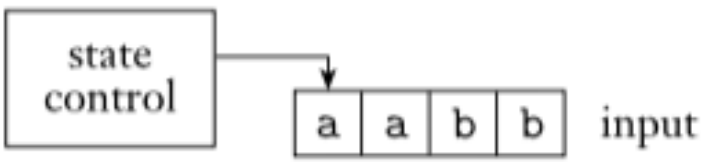


What is different?

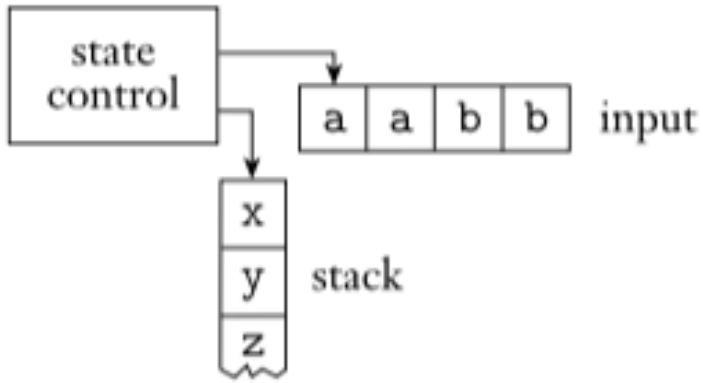
Automata Schematics

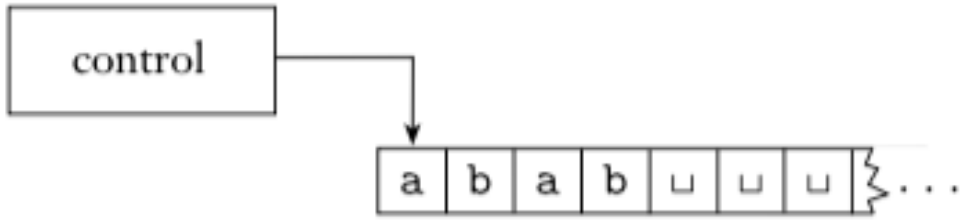
Recall the first two types of automata:

Schematic Finite Automaton



Schematic Pushdown Automaton





Schematic Turing Machine

What is different? unrestricted memory

Definition of a Turing Machine

Definition

A Turing machine is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_s, q_A, q_R)$, where

- Q is a finite set called the states,
- Σ is a finite input alphabet, $\sqcup \notin \Sigma$
- Γ is a finite tape alphabet, $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function,
- $q_s \in Q$ is the start state,
- $q_A \in Q$ is the accept state,
- $q_R \in Q$ is the reject state.

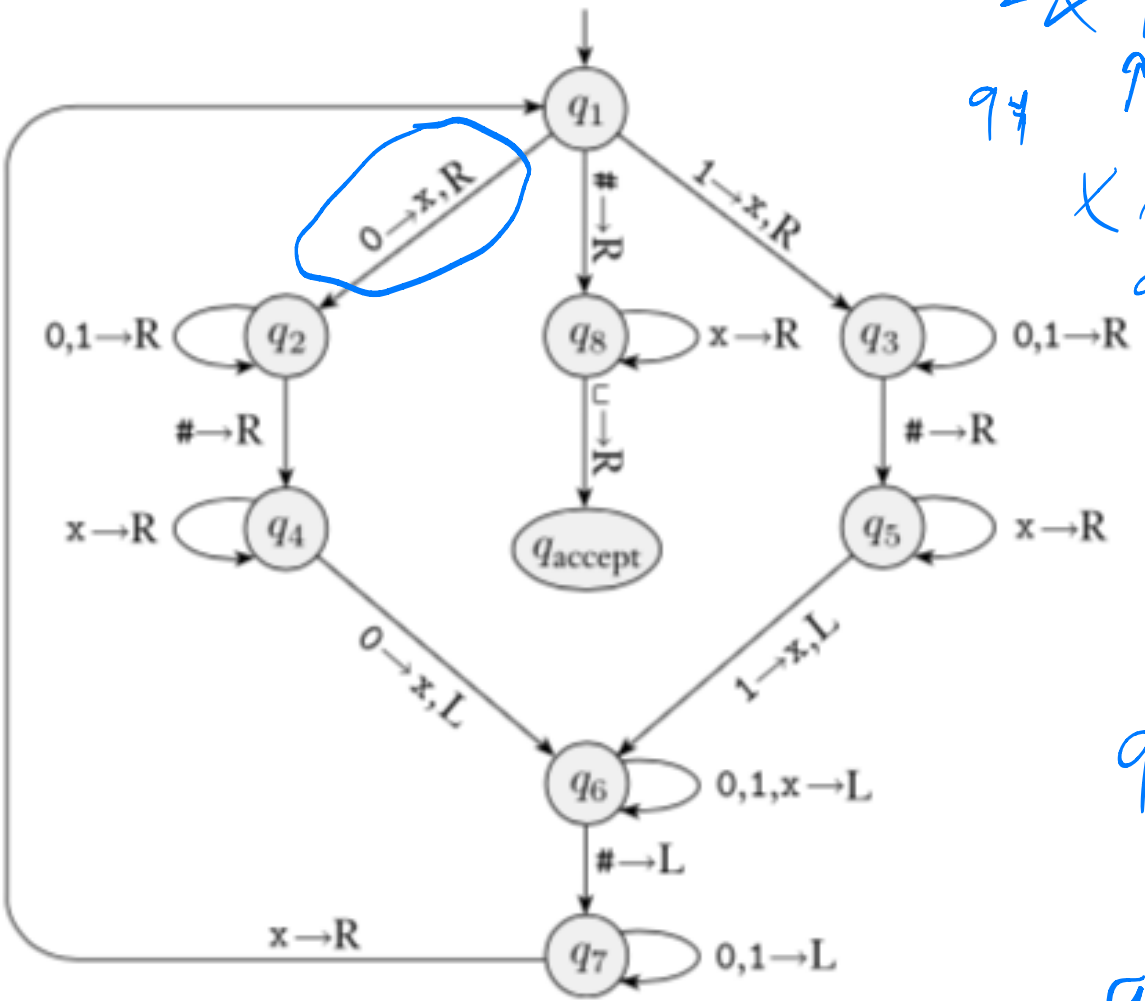
Computation of a Turing Machine

A Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_s, q_A, q_R)$, given input $w = w_1 w_2 \ldots w_n \in \Sigma^*$ on the leftmost n cells of the tape (the rest of the cells blank), computes as follows.

- The head starts on the leftmost square of the tape.
- The first blank on the tape marks the end of the input.
- The computation follows M's transition function.
- If M ever tries to move its head to the left off the left-hand end of the tape, the head stays in the same place.
- The computation continues until it enters either the accept or reject states, at which point it halts.
- If neither occurs, M goes on forever.

Turing Machine Example

$w\#w \mid w \in \{0,1\}^*$



q_1 # 0 1 # 0 1

q_2 x 1 # 0 1

q_4 x 1 # 0 1

x 1 # 0 1

q_6 x 1 # x 1

q_7 x 1 # x 1

q_1 x 1 # x 1

x x # x 1

q_3

Turing Machine Example

$L = \{w\#w | w \in \{0,1\}^*\}$
TM for L will use a simple strategy ~~strategy~~: repeatedly match (cross off) symbols on the two sides of the #.
$M =$ "On input string w:

1. Zig-zag across the tape to corresponding positions on either side of the # symbol to check whether these positions contain the same symbol. If they do not, or if no # is found, reject. Cross off symbols as they are checked to keep track of which symbols correspond.
2. When all symbols to the left of the # have been crossed off, check for any remaining symbols to the right of the #. If any symbols remain, reject; otherwise, accept."

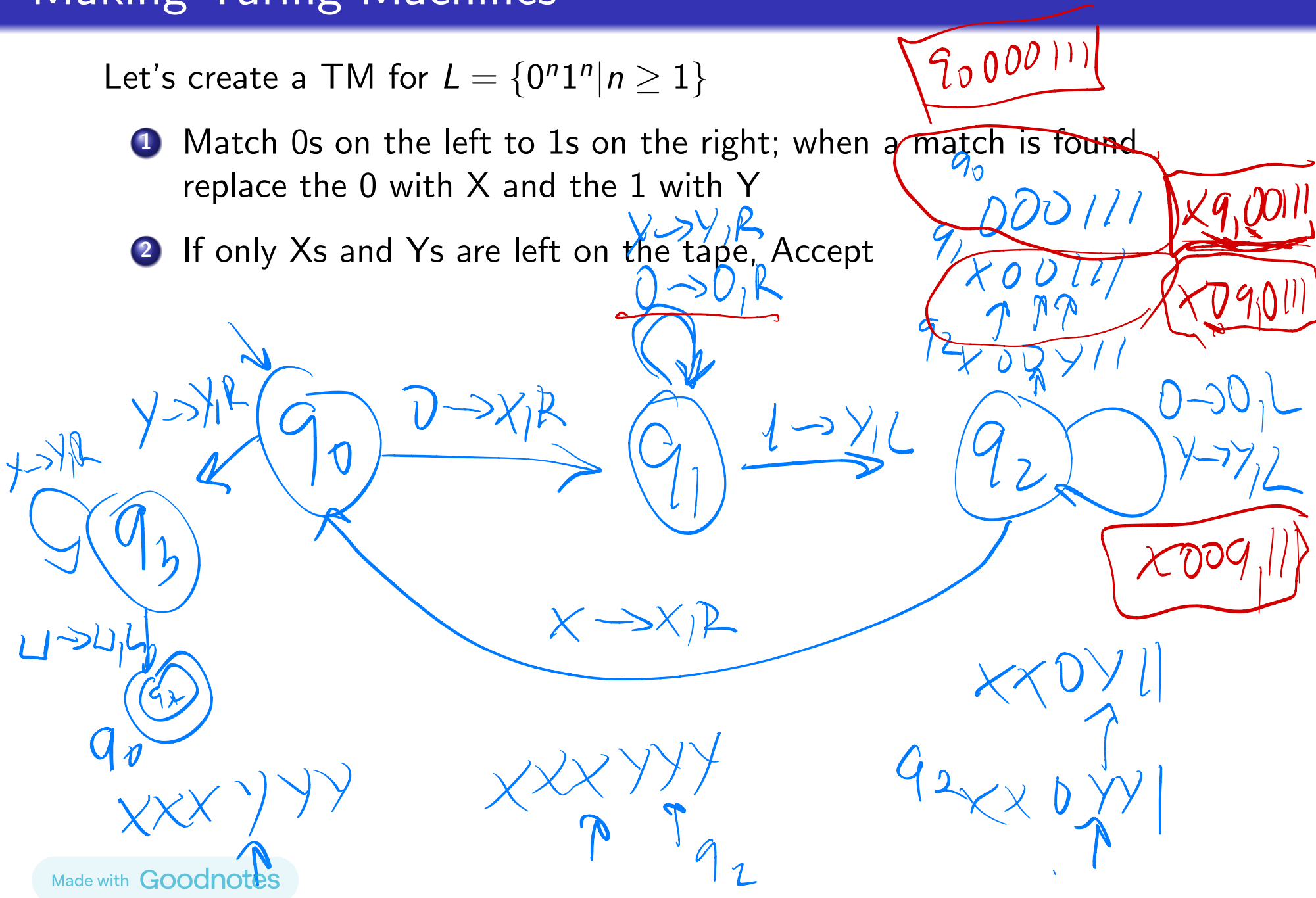
Turing Machine descriptions

- A **formal description** that spells out in full the Turing machine's states, transition function, and so on. It is the lowest, most detailed level of description.
- An **implementation-level description**, in which we use English to describe the way that the Turing machine moves its head and the way that it stores data on its tape. At this level we do not give details of states or transition function.
- A **high-level description**, wherein we use English to describe an algorithm, ignoring the implementation details. At this level we do not need to mention how the machine manages its tape or head.

Making Turing Machines

Let's create a TM for $L = \{0^n 1^n | n \geq 1\}$

1. Match 0s on the left to 1s on the right; when a match is found replace the 0 with X and the 1 with Y
2. If only Xs and Ys are left on the tape, Accept



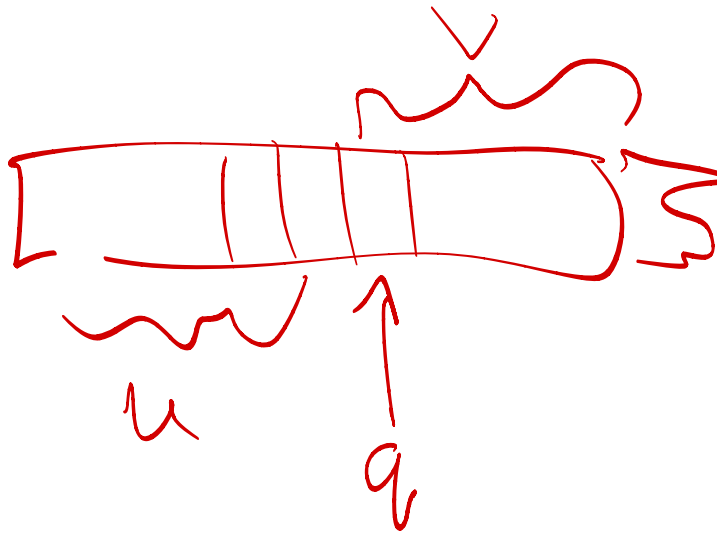
Turing Machine Configurations

As a Turing machine computes, changes occur in:

- the current state,
- the current tape contents,
- the current head location.

A setting of these three items is a **TM configuration**
Configurations are represented as follows:

- $< uqv >$ indicates that M is in state q with string u to the left of the tape head and string v to the right.
- the tape contents are uv
- M's head is over the first symbol of v
- the rest of the tape contains blanks



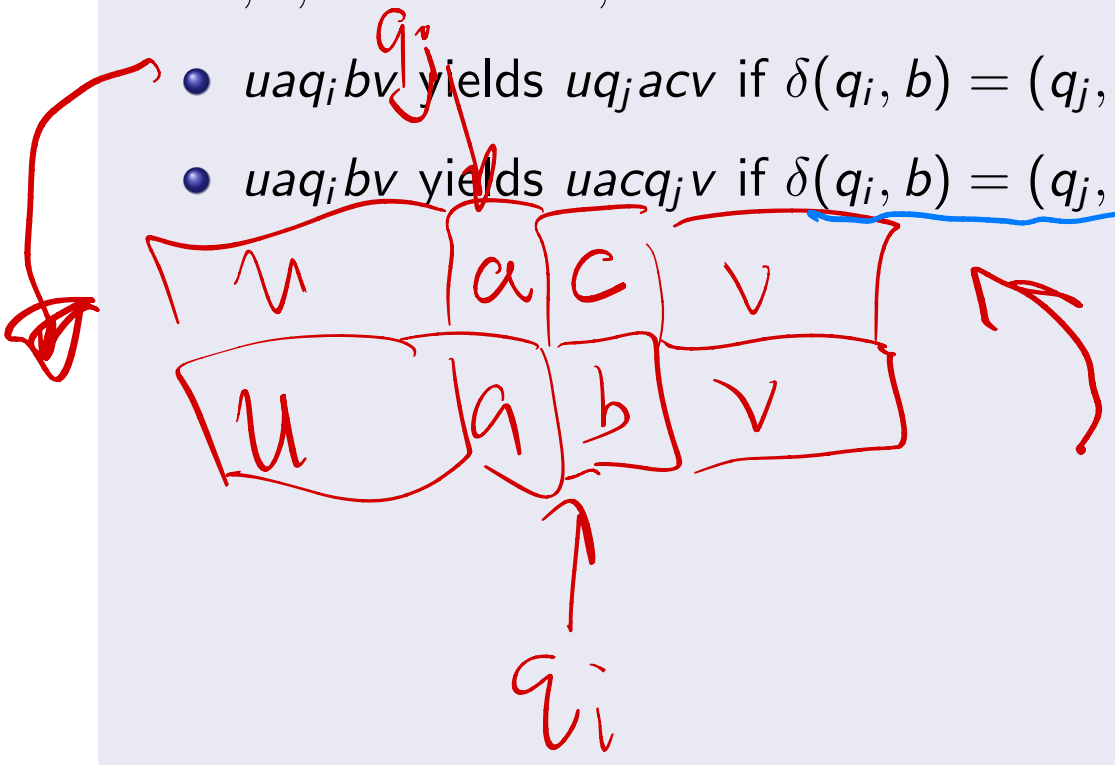
Back to Turing Machine Computation

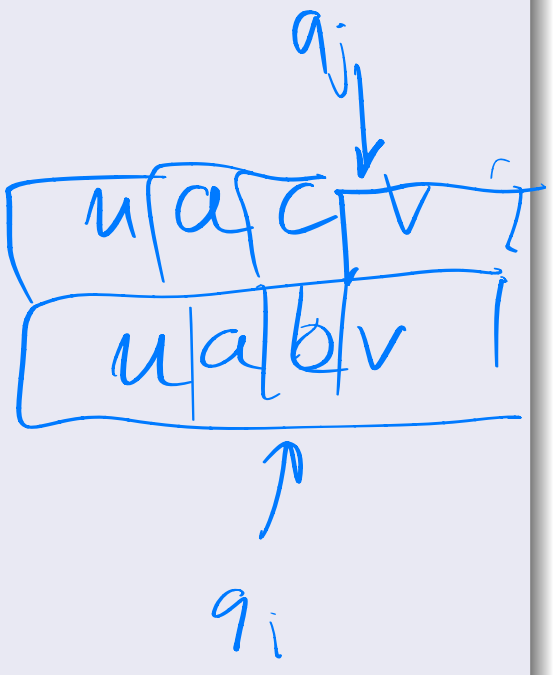
Configuration C_1 yields configuration C_2 if the TM can transition from C_1 to C_2 in a single step, defined formally as follows:

Definition

Let $a, b, c \in \Gamma$ and $u, v \in \Gamma^*$. Then:

- uaq_ibv yields uq_jacv if $\delta(q_i, b) = (q_j, c, L)$.
- uaq_ibv yields $uacq_jv$ if $\delta(q_i, b) = (q_j, c, R)$.





Back to Turing Machine Computation

Configuration C_1 yields configuration C_2 if the TM can transition from C_1 to C_2 in a single step, defined formally as follows:

Definition

Let $a, b, c \in \Gamma$ and $u, v \in \Gamma^*$. Then:

- uaq_ibv yields uq_jacv if $\delta(q_i, b) = (q_j, c, L)$.
- uaq_ibv yields $uacq_jv$ if $\delta(q_i, b) = (q_j, c, R)$.

The **start configuration** of M on input w is q_Sw. In an **accepting configuration**, the state of the configuration is q_A. In a **rejecting configuration**, the state of the configuration is q_R. Accepting and rejecting configurations are **halting configurations**. A TM M accepts input w if a sequence of configurations $C_1, C_2 \ldots C_k$ exists, where:

1. C_1 is the start configuration of M on input w,
2. each C_i yields C_{i+1}, and
3. C_k is an accepting configuration.

Language of Turing Machines

Definition

The collection of strings accepted by some TM M is the language recognized by M, denoted $L(M)$.

Definition

a TM that halts on all inputs is called a **decider**. A decider that recognizes some language also is said to **decide** that language.

Definition

Call a language **Turing-decidable** or just **decidable** if some Turing machine decides it. This is also known as a **recursive** language.

Deciders vs Recognizers

Recall that given some input a TM can **accept**, or **reject**, or **loop** (not enter a halting state) on a given input. With this in mind:

Definition

a TM that halts in an accept state on all $w \in L$ is called a **recognizer** for language L. Note that if $w \notin L$ a recognizer might not terminate.

Definition

A language **Turing-recognizable** if some TM recognizes it. This is also known as a **recursively enumerable** language.

Another TM Example

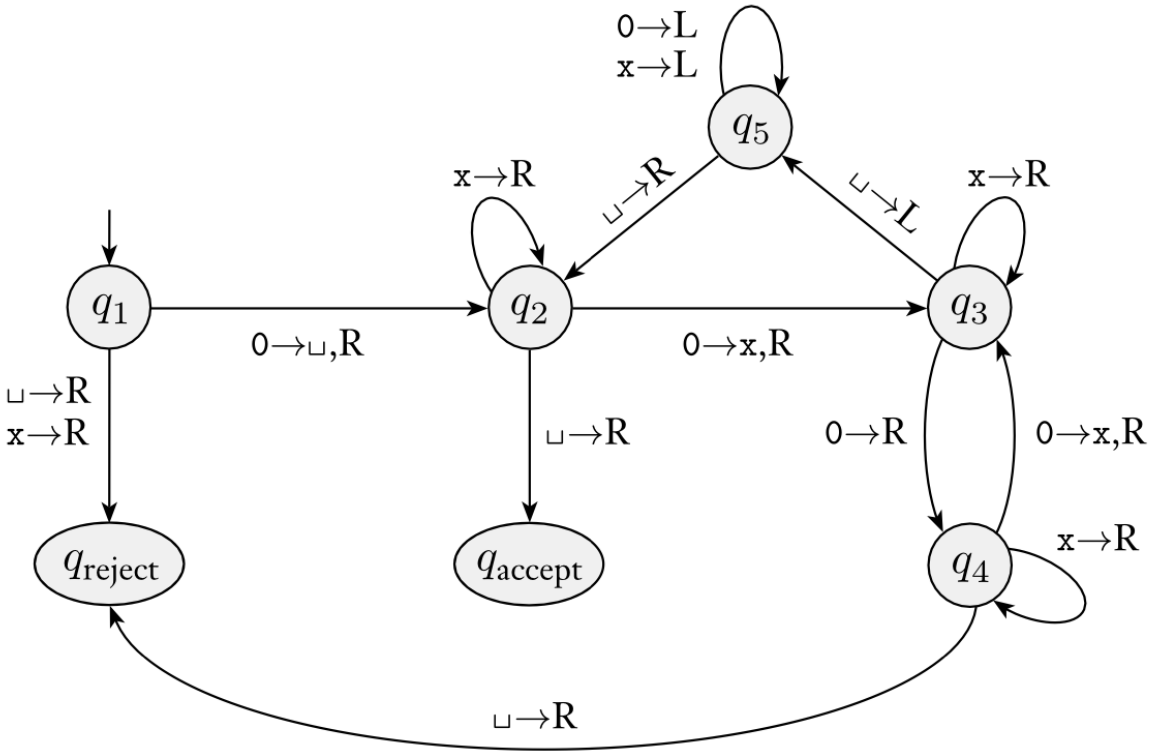
Consider the language $A = \{0^{2^n} | n \geq 0\}$, the language consisting of all strings of 0s whose length is a power of 2.

Strategy: If there's exactly one 0 accept. Otherwise, repeatedly cut the number of 0s in half keeping track of whether the number of 0s remaining are odd or even. If an odd number remains, reject; otherwise if all 0s are gone, accept.

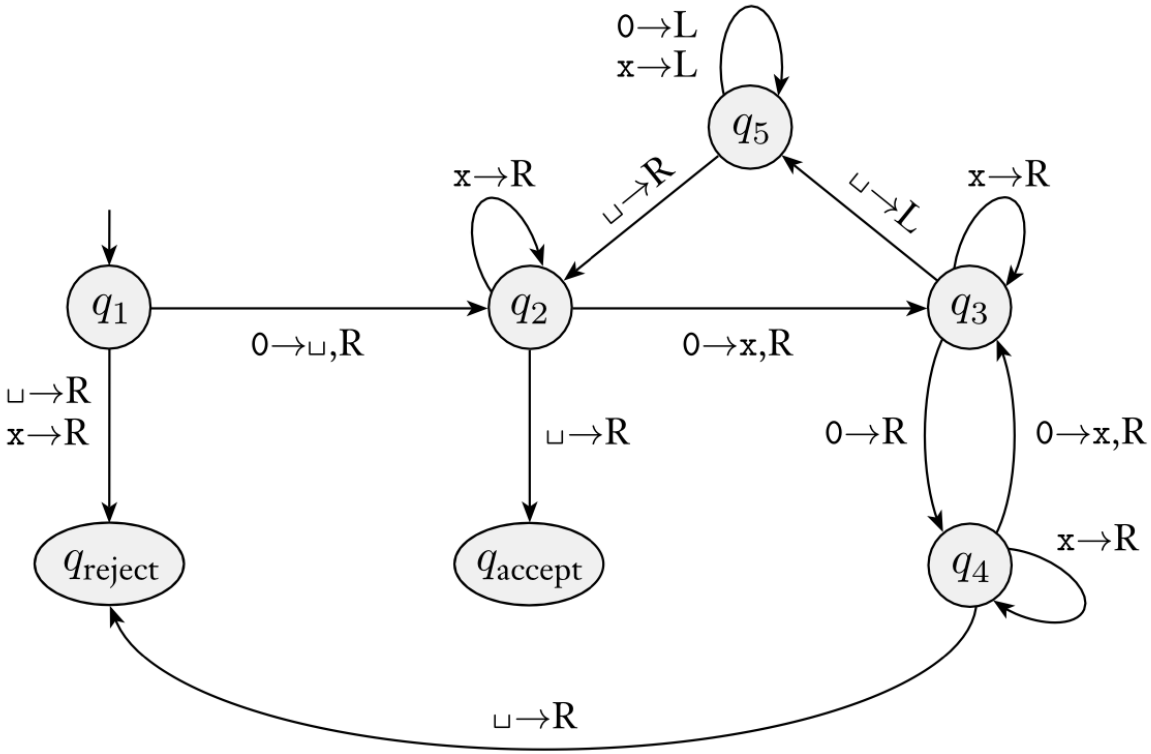
M = "On input string w:

1. Sweep left to right across the tape, crossing off every other 0.
2. If in stage 1 the tape contained a single 0, accept.
3. If in stage 1 the tape contained more than a single 0 and the number of 0s was odd, reject.
4. Return the head to the left-hand end of the tape.
5. Go to stage 1."

Another TM Example



Another TM Example



$q_1$0000
␣$q_2$000
␣x$q_3$00
␣x0$q_4$0
␣x0xq_3␣
␣x0q_5x␣
␣x$q_5$0x␣
␣q_5x0x␣
q_5␣x0x␣
␣q_2x0x␣
␣x$q_2$0x␣
␣xxq_3x␣
␣xxxq_3␣
␣xxq_5x␣
␣x$q_5$xx␣
␣$q_5$xxx␣
q_5␣xxx␣
␣$q_2$xxx␣
␣x$q_2$xx␣
␣xxq_2x␣
␣xxxq_2␣
␣xxx␣q_{accept}