Automata Theory (aka Foundations of CS) CISC 303

Instructor:	Prof. Ilya Safro, 430 Smith Hall
Office hours:	Wednesdays 9am-10am or by appointment
TA:	Sristy Sangskriti, Connor Nagle

Course Structure

What	How many	Time	Points
Homework	**12-14**	**1 week**	**30**
Midterm exam I	**1**	**1.25 hours**	**20**
Midterm exam II	**1**	**1.25 hours**	**20**
Final exam	**1**	**1.25 hours**	**20**
Pop-up or online quizzes	**10-15**	**5-7 min**	**10**
Total			**100.00**

Points	Grade
≥ 89.00	**A**
≥ 79.00	**B**
≥ 65.00	**C**
≥ 50.00	**D**
≥ 0	**F**

Bonuses

Work in class, extra work in homework exercises, etc. - up to 10 points. We do not want to miss the next Turing, Fields and Nobel laureates, so any submitted conference/journal paper written during and as a result of this course - 100 points, and new interesting ideas - up to 100 points (both are based on instructor's subjective judgment).

NO ELECTRONIC DEVICES IN THIS CLASSROOM!

All slides will be available online after each class

Average over all midterms ≥ 89.00

+

Average over all homework assignments ≥ 89.00

+

Average over all quizzes ≥ 89.00

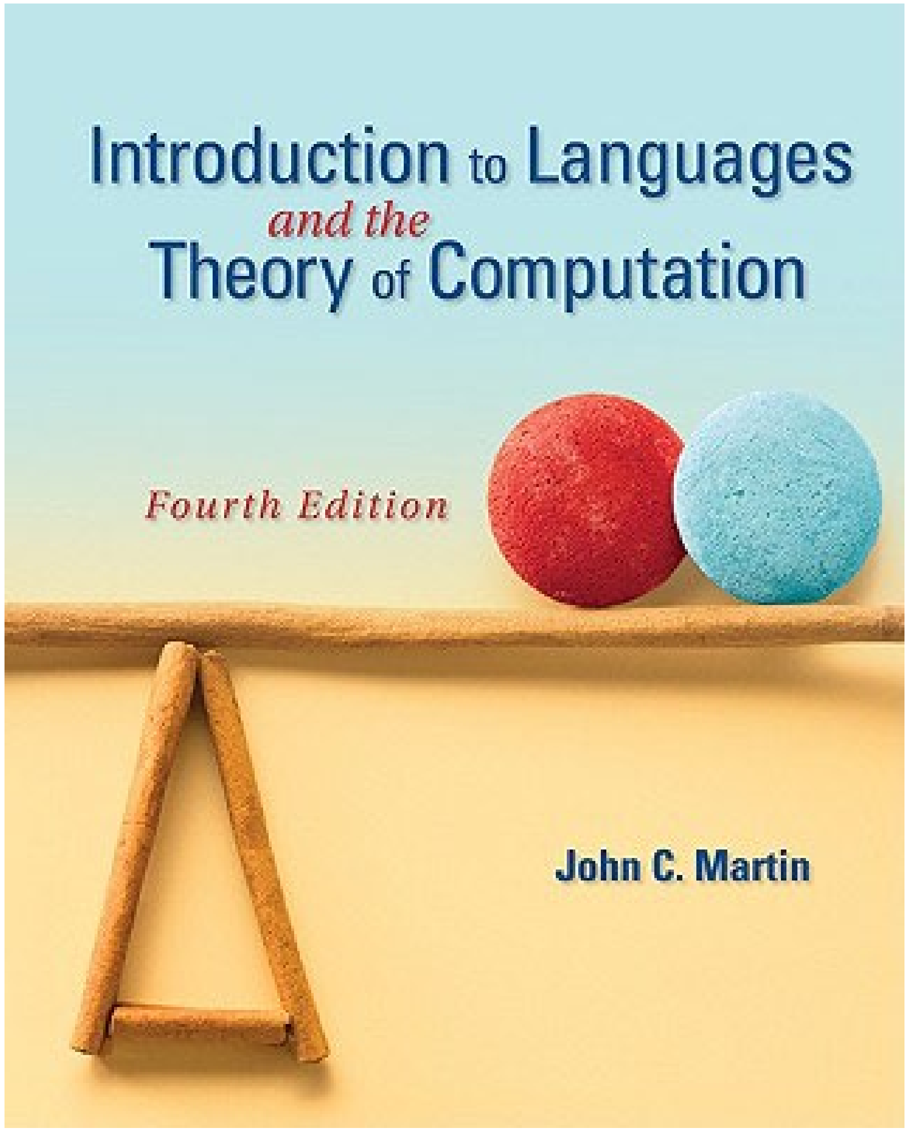
(which also means the attendance)

=

No final exam

- **Pop-up or online quizzes: no unexcused absences are allowed; unexcused absences get counted as zeros**
- **<u>If</u> any curve will be used, a minimum score of 40 is still required to pass this course**
- **No curves will be given in the middle of the semester. Your grade reflects your knowledge.**
- **In the end of the semester a curve of at most N required points will be given to satisfy two conditions (if they are not satisfied without the curve)**
  - **Top 15% of students will have an A**
  - **The last student in the top 70% will have a C**
- **No curve will be applied to students who missed midterm or final exams.**

Recommended Book



Assumption:
You have all prerequisites and you know

- **mathematical induction**
- basics of set theory (sets, inclusion, difference, union, **proofs of equality**, etc.)
- basics of mathematical logic (operators and/or/not/$\rightarrow$, **proofs "if and only if"**, etc.)

You can find this material in Chapter I.

- Grimaldi “Discrete and Combinatorial Mathematics: An Applied Introduction “ (very good introductory book to cover math that you may need)
- Linz “Formal Languages and Automata” (not easy but very good)
- Hopcroft, Motwani, Ullman “Introduction to Automata Theory, Languages, and Computation” (one of the best classical textbooks)
- Sipser “Introduction to the Theory of Computation” (one of the best classical textbooks)
- Meduna “Automata and Languages”
- Sudkamp “Languages and Machines”
- Arora, Barak “Computational Complexity: A Modern Approach” (recommended to students who are interested in algorithms and theory; especially if you plan to continue for MSc or PhD)

- https://en.wikipedia.org/wiki/JFLAP JFLAP is a software for experiments with languages, finite automata and Turing machines. Use it when we will start with finite automata (in 1-2 weeks)

Important

- In some homework assignments, new definitions, principles, and algorithms will be introduced. Exams can include them! All exams are cumulative over any and all previous and current material.
- Exams will be closed book, closed notes and closed any other aids. A score of 0 will be given to anyone not present at the beginning of the exam.

Very important

- You cannot copy-paste solutions from the Internet, books, friends, etc. If you don’t solve them by yourself, this will be the best way to fail the exams.
- Solve as many exercises from the textbook as you can. Try to solve more than what you get in the assignments. **This is not a passive learning course.**
- **Common mistake: you are sure that you understand some chapter (which is easy) but you did not solve 30-40 problems from that chapter by yourself (which is hard).**
- All chapters are cumulative. Do not neglect any material.
- There are no dumb questions. Certainly, not in such a challenging course. You can ask anything. It is normal if you don’t understand something. It is not normal if you don’t understand something and immediately start working on it at home.

Triviality

Sometimes I will say, or you will read in books that something is *trivial*. It does not mean that I and the book author are smart, and if you do not understand then you are not.

Triviality in math refers to: the mathematically most simple case; any result which requires little or no effort to derive or prove; very simple object structure; something that directly follows from the definition.

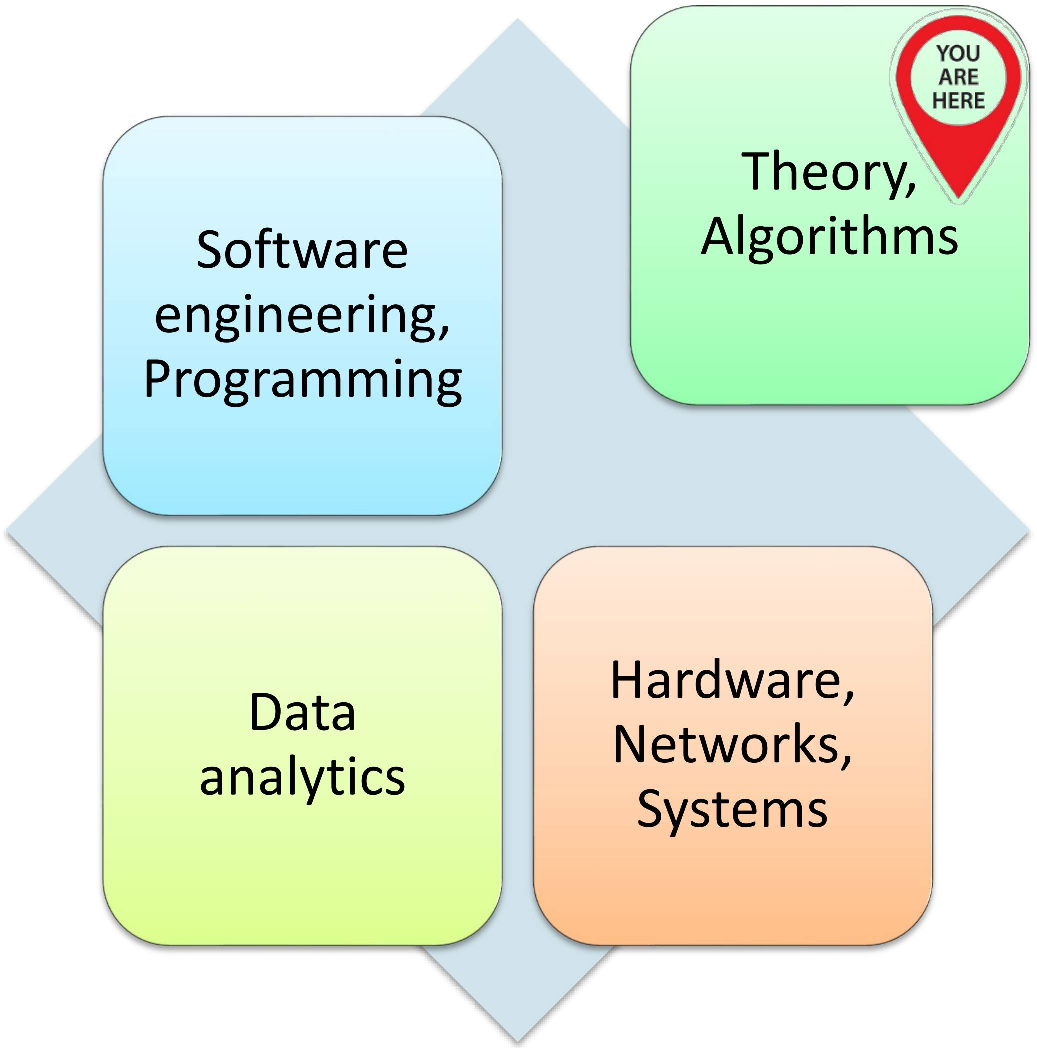
There are several mathematical definitions of triviality that depend on the context.

Example: For $10x+5y-9z = 0$ $x=y=z=0$ is called a trivial solution

See https://en.wikipedia.org/wiki/Triviality_(mathematics) or many books in mathematical reasoning.

Major reasons for D and F in this course

- Some students start preparing for midterms or finals one week in advance. You must work during the semester.
- Copy-pasting homework solutions
- Students don’t understand theorems and proofs. Just memorizing them is not the best idea.
- **Students start this course without knowing mathematical induction**
- Student don’t read and solve additional examples in the textbook
- Not attending lectures
- Not taking quizzes

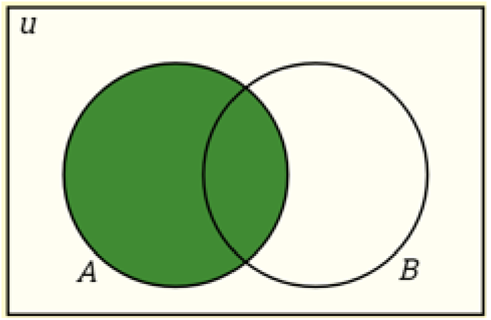

Software engineering, Programming
Theory, Algorithms
YOU ARE HERE
Data analytics
Hardware, Networks, Systems

You will

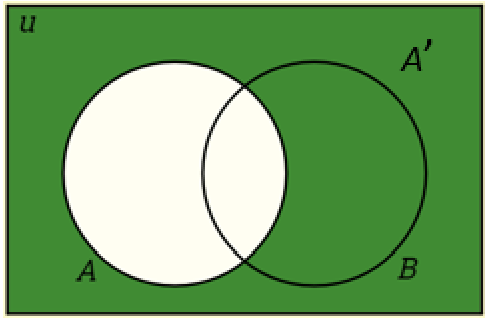
- Discuss computer science in the language of mathematics
- Think about programming using mathematical objects
- Define the notion of an **algorithm**
- Compare algorithms
- Learn computational models
- Understand their fundamental limitations and advantages
- Study ideas behind compilers
- Distinguish between solvable and unsolvable problems (no matter what computers you use)
- Learn fundamentals of computing that explain what we can expect to solve with reasonable resources
- Use rigorous proofs to confirm the conclusions where we can
- Discuss various complexity classes and the P vs NP millennium problem
- Solve a lot of brain teasers
- See some counterintuitive things

Sets

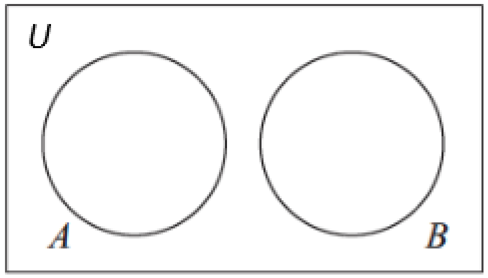
Set Operations and Venn Diagrams



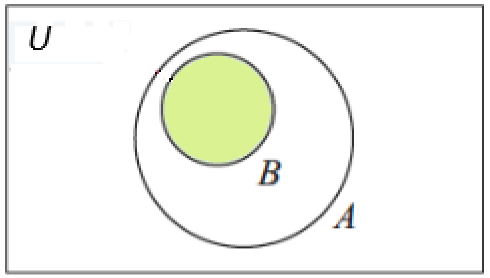
Set *A*



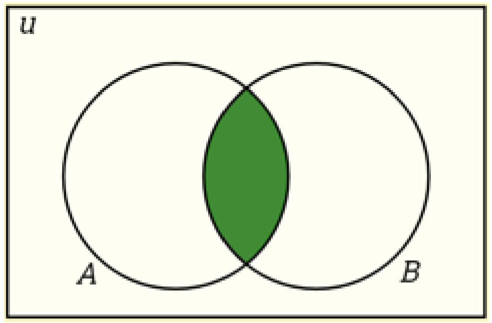
A' the complement of *A*



A and *B* are disjoint sets

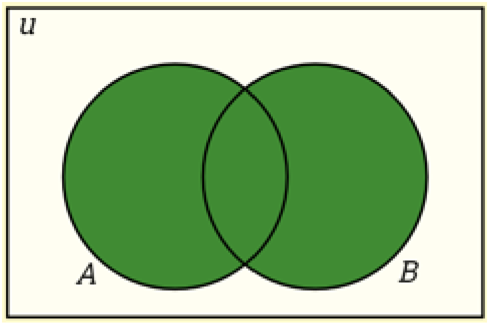


B is proper subset of *A* $B \subset A$



Both A and B $A \cap B$

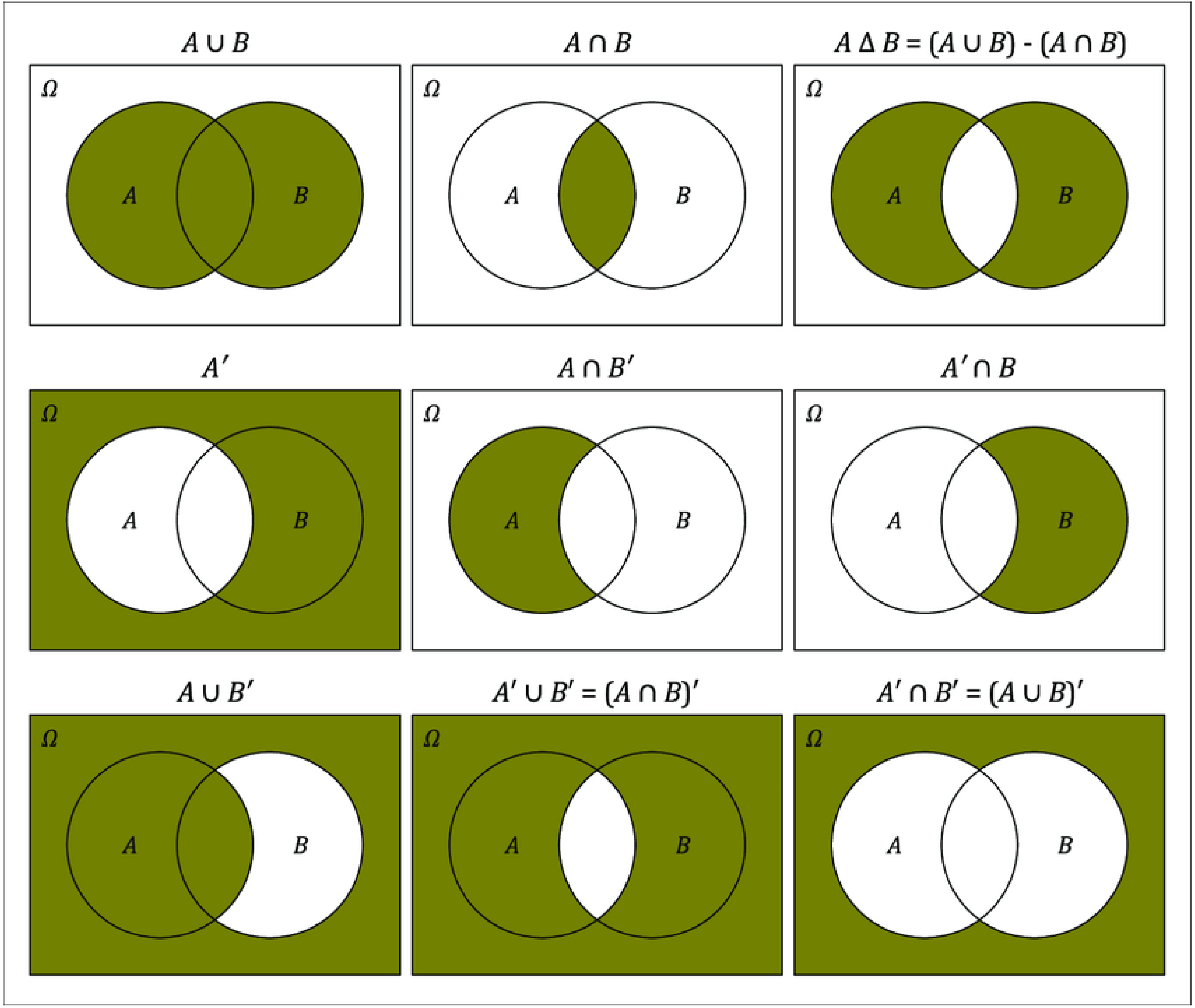
A intersect *B*



Either A or B $A \cup B$

A union *B*

https://www.onlinemathlearning.com/venn-diagrams.html


$A \cup B$
$A \cap B$
$A \Delta B = (A \cup B) - (A \cap B)$
A'
$A \cap B'$
$A' \cap B$
$A \cup B'$
$A' \cup B' = (A \cap B)'$
$A' \cap B' = (A \cup B)'$
Ω
A
B

Cardinal "Sets, Graphs, and Things We Can See: A Formal Combinatorial Ontology for Empirical Intra-Site Analysis" 13

How to prove that sets A and B are equal?

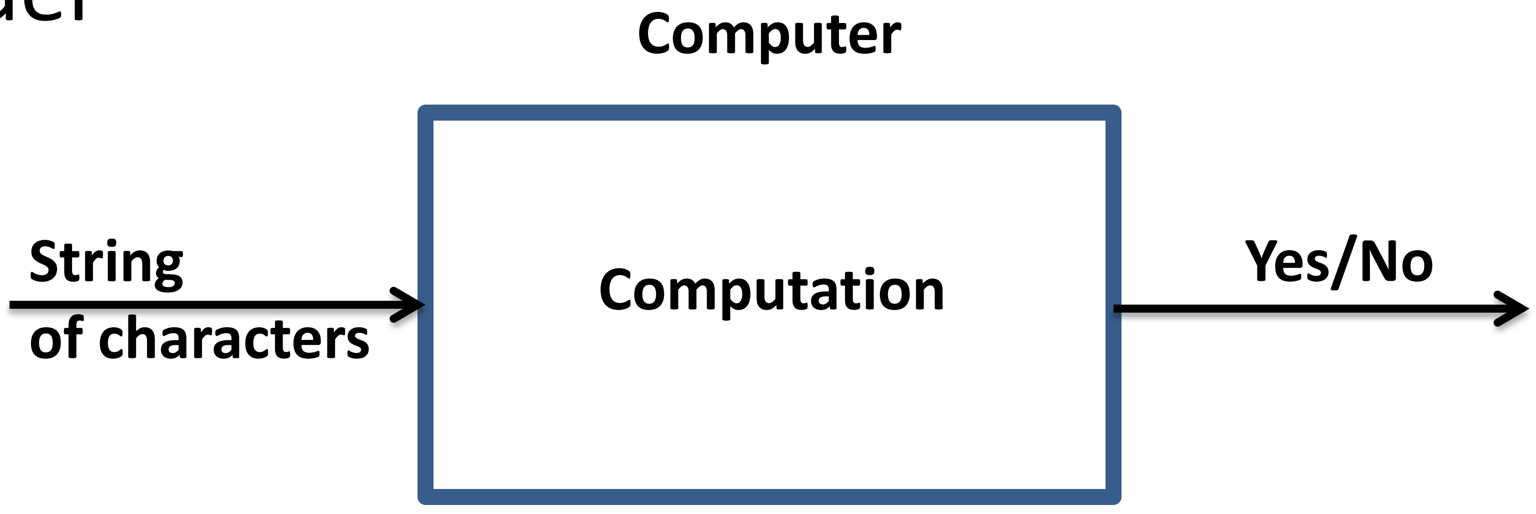
In order to prove that $A = B$ you need to prove that

- $A \subseteq B$
- $B \subseteq A$

How to prove that $A \subseteq B$? Without loss of generality take any element $x \in A$ and prove that it is also an element of B.

Languages

Model



This computer plays a role of a language acceptor

Examples:

- We enter a string and ask whether this string is a legal algebraic expression or not.

  a+b/c → the answer is YES

  aa+++b--- → the answer is NO

- We enter a string and ask whether this string includes exactly 3 characters *a*, and 5 characters *b* or not.

Definition (Alphabet)

- An alphabet (usually denoted by $\sum$) is a finite set of symbols, e.g., $\{a, b\}$, $\{0, 1\}$, $\{A, B, C, \dots, Z\}$, or $\{\spadesuit, \bigstar, \heartsuit\}$.
- A string over $\sum$ is a finite sequence of symbols in $\sum$. For a string x, $|x|$ stands for the length of x. Example: $|\heartsuit\spadesuit\bigstar\heartsuit| = 4$.
- $n_\sigma(x)$ is the number of occurrences of the symbol σ in the string x. Example: $n_a(abbba) = 2$.
- The null string Λ is a string over $\sum$, no matter what the alphabet $\sum$ is. By definition, $|\Lambda| = 0$.

Definition (Alphabet)

- The set of all strings over Σ will be written as Σ^*. For $\Sigma = \{a, b\}$, we have

$$\{a, b\}^* = \{\Lambda, a, b, aa, ab, ba, bb, aaa, aab, \ldots\}$$

canonical order

Remark: There exist infinite alphabets but not in this course.

Canonical order, the order in which shorter strings precede longer strings and strings of the same length appear alphabetically. Canonical order is different from *lexicographic*, or strictly alphabetical order, in which *aa* precedes *b*.

Definition A language over Σ is a subset of Σ^*.

Examples:

empty set

- The empty language $\emptyset$.
- $\{\Lambda, a, aab\}$
- The language PAL of palindromes over $\{a, b\}$ (strings such as *aba* or *baab* that are unchanged when the order of the symbols is reversed).
- $\{x \in \{a,b\}^* \mid n_a(x) > n_b(x)\}$
- $\{x \in \{a,b\}^* \mid |x| \geq 2$ and x begins and ends with $b\}$

Definition A language over Σ is a subset of Σ^*.

Examples:

empty set

- The empty language $\emptyset$.
- $\{\Lambda, a, aab\}$
- The language PAL of palindromes over $\{a, b\}$ (strings such as aba or baab that are unchanged when the order of the symbols is reversed).
- $\{x \in \{a, b\}^* \mid n_a(x) > n_b(x)\}$
- $\{x \in \{a, b\}^* \mid |x| \geq 2 \text{ and } x \text{ begins and ends with } b\}$

Λ is always an element of Σ^*, but other languages over Σ may or may not contain it. More real examples:

- The language of numeric literals in Java such as 0.3 and 5.0E-3.
- The language of legal Java programs. $\Sigma = \{\text{numbers,letters}, \ldots\}$.

- Concatenation: if x and y are two strings, the concatenation of x and y is written xy and consists of the symbols of x followed by those of y.

  Example: if $x = ab$ and $y = bab$ then $xy = abbab$ and $yx = babab$.
  For every string x, $x\Lambda = \Lambda x = x$.

- If s is a string and

$$s = tuv$$

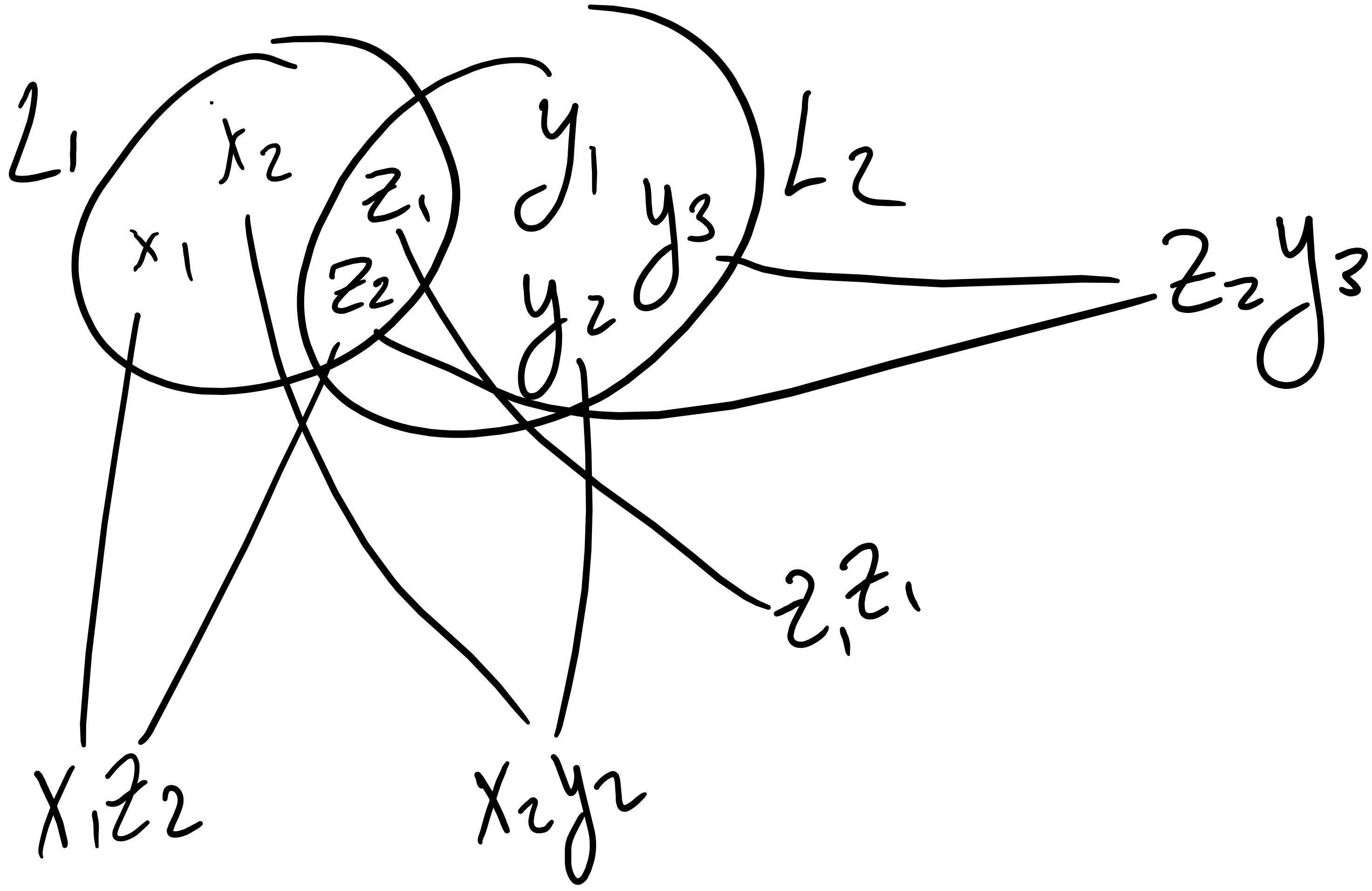
  for three strings t , u, and v, then t is a prefix of s, v is a suffix of s, and u is a substring of s.

- Each of them is not necessarily non-empty string.

- Concatenation of languages L_1 and L_2 is the language

$$L_1L_2 = \{xy \mid x \in L_1 \text{ and } y \in L_2\}$$

Example: $\{a, aa\}\{\Lambda, b, ab\} = \{a, ab, aab, aa, aaab\}$.



Exponents:

- If $a \in \sum$

$$a^k = aa \dots a \quad (k \text{ times})$$

- If $x \in L$

$$x^k = xx \dots x$$

- We define L^k for a language L

$$L^k = \{x_1 x_2 \dots x_{k-1} x_k \mid x_i \in L,\ 1 \leq i \leq k\}$$

- For pairs of alphabets and languages we can define union, intersection, and difference operations $(\cup, \cap, -)$

$$\begin{aligned}
\Sigma_1 \cup \Sigma_2 &= \{a \mid a \in \Sigma_1 \text{ or } a \in \Sigma_2\} \\
\Sigma_1 \cap \Sigma_2 &= \{a \mid a \in \Sigma_1 \text{ and } a \in \Sigma_2\} \\
\Sigma_1 - \Sigma_2 &= \{a \mid a \in \Sigma_1 \text{ and } a \notin \Sigma_2\}
\end{aligned}$$

Same for languages L_1, and L_2.

- The Kleene star (or Kleene closure) operation on a language L

$$L^* = L^0 \cup L^1 \cup L^2 \cup L^3 \cdots = \bigcup \{L^k \mid k \in \mathbb{N}\},$$

where $L^0 = \{\Lambda\}$

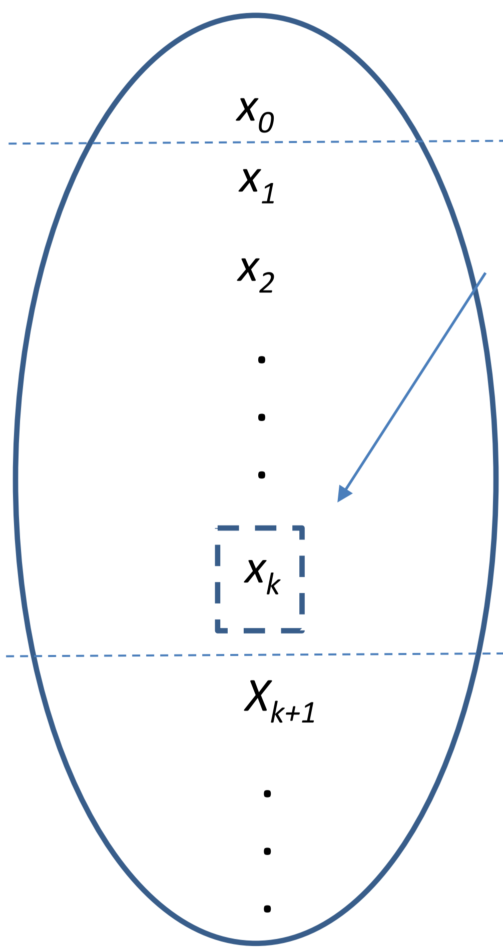
- Precedence rules are similar to the algebraic rules. Example:

$$L_1 \cup L_2 L_3^* = L_1 \cup (L_2(L_3^*))$$

Mathematical Induction

Simple (or regular) induction

Strong induction



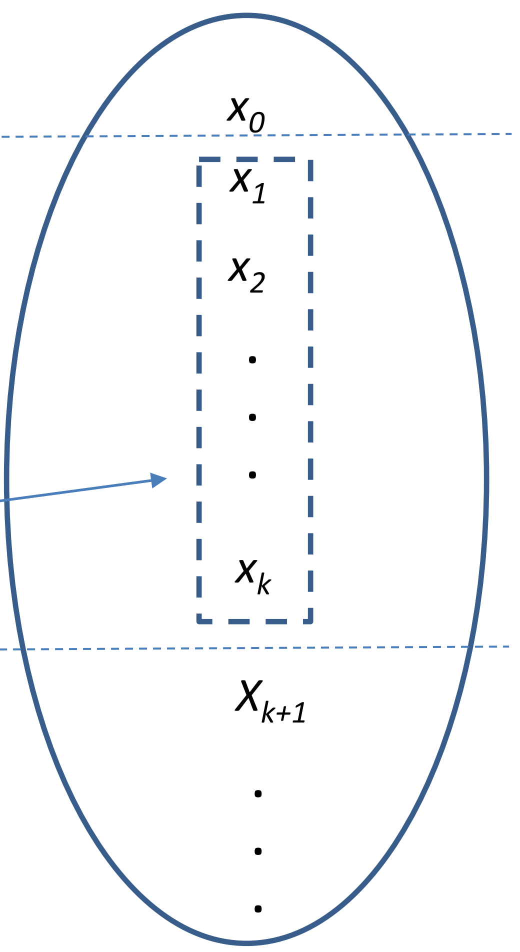
Basis step

1. Prove $P(x_0)$ for the smallest relevant element in S

2. Assume $P(x_k)$ *is true* for x_k

Hypothesis step

2. Assume $P(x_k)$ *is true* for all elements up to x_k



Induction step

3. Prove $P(x_{k+1})$ for the next relevant element in S

Prove by induction on n that

$$\sum_{i=1}^{n} i = \frac{n \cdot (n+1)}{2}$$

- Basis step: $n = 1$, $1 = 1(1+1)/2$

me $n = k$, • Hypothesis: Suppose the claim is true for so

$$\sum_{i=1}^{k} i = k(k+1)/2$$

ıe claim for $n = k+1$ • Induction step: We need to prove tl

$$k+1)/2 + (k+1) = (k+1)(k+2)/2 \quad \sum_{i=1}^{k+1} i = \sum_{i=1}^{k} i + (k+1) = k($$

true for all n. Thus the claim is

Proofs by contradiction

For every proposition p, p is equivalent to the conditional proposition

$$true \to p,$$

whose contrapositive is

$$\neg p \to false.$$

X	Y	X → Y	˥Y	˥Y→X
T	T	T	F	T
T	F	F	T	T
F	T	T	F	T
F	F	T	T	F

A proof of p by contradiction means

1. assuming that p is false and
2. deriving a contradiction (i.e., deriving false statement).

Proofs by contradiction

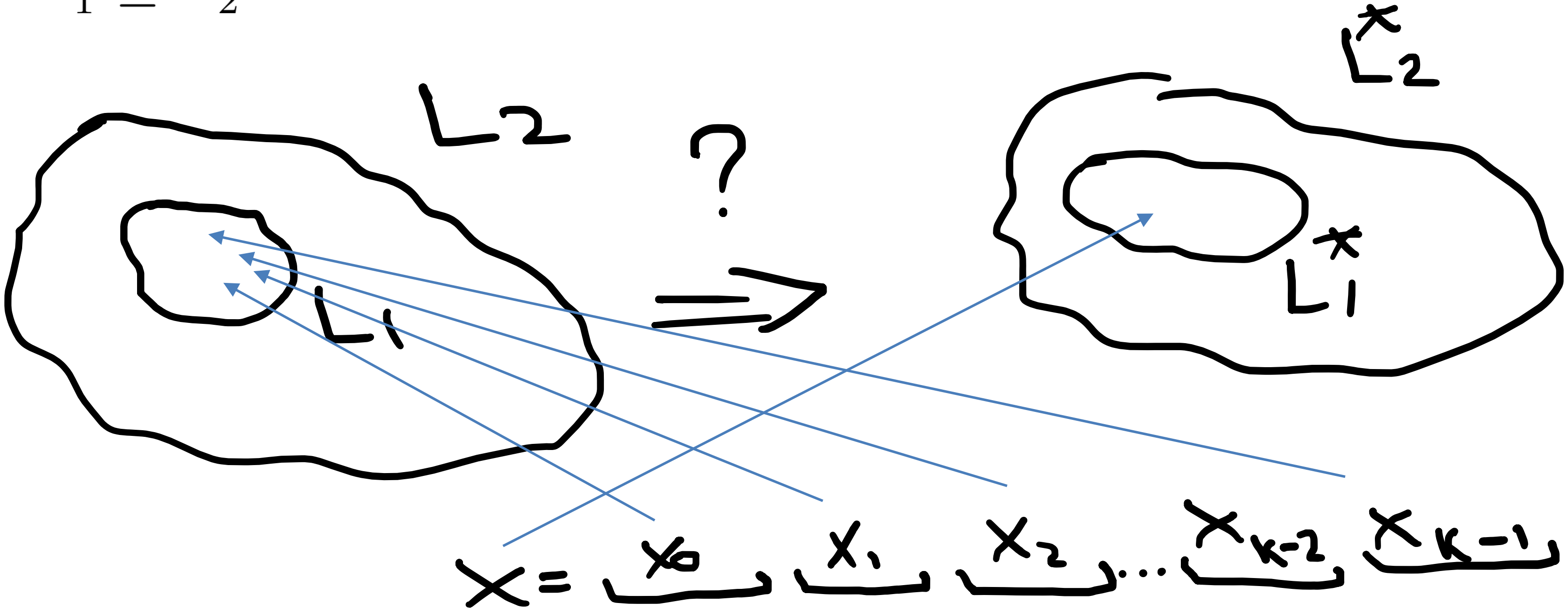
Example of a proposition with proof by contradiction:

There is no smallest positive real number (SPRN).

Proof by contradiction:

- Suppose that x is SPRN.
- Then $x>0$ because it is given that x is positive.
- But if we take $0<\frac{1}{2}<1$, and multiply by x we obtain $0<\frac{1}{2}x<x$.
- $\frac{1}{2}x$ is smaller than x, so this is a contradiction to the assumption
- Hence, there is no SPRN

Claim. *Let L_1 and L_2 be subsets of $\{a,b\}^*$. Prove that if $L_1 \subseteq L_2$, then $L_1^* \subseteq L_2^*$.*



$x = \underbrace{\underbrace{x_0}\ \underbrace{x_1}}\ \underbrace{x_2} \dots \underbrace{x_{k-2}}\ \underbrace{x_{k-1}}$

$x \in L_1^* \Rightarrow x \in L_2^*$ Basis for 1 string

Hypothesis for $k-1$ substrings $\in L_2^*$

$x = \underbrace{x_0 \dots x_{k-2}}_{y}$

Induction step $x = \underset{L_2^*}{y}\ \underset{L_2}{x_{k-1}} \Longrightarrow x \in L_2^*$

Claim. *Let L_1 and L_2 be subsets of $\{a, b\}^*$. Prove that if $L_1 \subseteq L_2$, then $L_1^* \subseteq L_2^*$.*

- We will prove this claim by contradiction.
- The claim says: if $L_1 \subseteq L_2$, then $L_1^* \subseteq L_2^*$, i.e., given the condition $L_1 \subseteq L_2$, we need to prove that all elements of L_1^* are also elements of L_2^*.
- A contradiction to it is when we assume that there exist something that contradicts the statement, and then we'll show that this is not true.
- Contradiction: let's assume that $\exists x \in L_1^*$ such that $x \notin L_2^*$.

This is what we will prove by induction, i.e., all three induction steps will be contradictions.

If you need to prove that $A \subseteq B$, you need to prove that every $x \in A$ is also in B.

Claim. *Let L_1 and L_2 be subsets of $\{a, b\}^*$. Prove that if $L_1 \subseteq L_2$, then $L_1^* \subseteq L_2^*$.*

Proof sketch. Let's assume that $\exists x \in L_1^*$ such that $x \notin L_2^*$.

By definition of L_1^*, x is a concatenation of k strings

$x_i \in L_1,\ 0 \leq i \leq k-1,$ i.e.,

Λ. $x = x_0 x_1 \ldots x_{k-1}$ or $x = \Lambda$

k, always showing the con- We prove the statement by induction on k ob

tradiction.

- Basis step:

he claim is correct because $\Lambda \in L_2^*$ by $- k = 0 \Rightarrow x = \Lambda$ - th

the definition of $*$;

$x_0 \neq \Lambda$ then $x \in L_1 \Rightarrow x \in L_2 \Rightarrow x \in L_2^*$. $- k = 1 \Rightarrow x = x$

Claim. *Let L_1 and L_2 be subsets of $\{a, b\}^*$. Prove that if $L_1 \subseteq L_2$, then $L_1^* \subseteq L_2^*$.*

Proof sketch (cont).

- Hypothesis: The claim is correct for $k - 1$ strings (i.e., the contradiction is wrong).
- Induction step:
  - $x = x_0 x_1 \cdots x_{k-2} x_{k-1}$ then $x = y x_{k-1}$, where y is a concatenation of $k - 1$ strings,
  - By induction hypothesis $y \in L_2^*$.
  - However, $x_{k-1} \in L_1$ and since $L_1 \subseteq L_2$ then $x_{k-1} \in L_2$.
  - Then, $x \in L_2^*$ because $x = y x_{k-1}$, where $y \in L_2^*$ and $x_{k-1} \in L_2$.

□

Claim. *Let L_1 and L_2 be subsets of $\{a, b\}^*$. Prove that*

$$L_1^* \cup L_2^* \subseteq (L_1 \cup L_2)^*.$$

Proof sketch.

$$\begin{aligned} L_1 \subseteq L_1 \cup L_2 \ &\Rightarrow & L_1^* \subseteq (L_1 \cup L_2)^* \\ L_2 \subseteq L_1 \cup L_2 \ &\Rightarrow & L_2^* \subseteq (L_1 \cup L_2)^* \\ &\Rightarrow & L_1^* \cup L_2^* \subseteq (L_1 \cup L_2)^*. \end{aligned}$$

Recursive definitions

- A recursive definition of a set begins with a *basis statement* that specifies one or more elements in the set. The *recursive part* of the definition involves one or more operations that can be applied to elements already known to be in the set, so as to produce new elements of the set.

Example: let $AnBn$ be the language over $\Sigma = \{a, b\}$ defined as $AnBn = \{a^n b^n \mid n \in \mathbb{N}\}$. Its recursive definition is

1. $\Lambda \in AnBn$
2. For every $x \in AnBn$, $axb \in AnBn$.

Example: recursive definition of PAL over $\Sigma = \{a, b\}$

Recursive definitions

- A recursive definition of a set begins with a *basis statement* that specifies one or more elements in the set. The *recursive part* of the definition involves one or more operations that can be applied to elements already known to be in the set, so as to produce new elements of the set.

Example: let $AnBn$ be the language over $\Sigma = \{a, b\}$ defined as $AnBn = \{a^n b^n \mid n \in \mathbb{N}\}$. Its recursive definition is

1. $\Lambda \in AnBn$
2. For every $x \in AnBn$, $axb \in AnBn$.

Example: recursive definition of PAL over $\Sigma = \{a, b\}$

1. $\Lambda, a, b \in PAL$
2. For every $x \in PAL$, $axa \in PAL$ and $bxb \in PAL$.

Recursive definitions of functions

Example 1: factorial function $n! = n \cdot (n-1) \cdot (n-2) \cdot ... \cdot 1$

$$f(0) = 1;\ \text{for every } n \in \mathbb{N},\ f(n+1) = (n+1) \cdot f(n)$$

Different notation

$$f(n) = \begin{cases} 1 & n = 0 \\ nf(n-1) & \text{otherwise} \end{cases}$$

Example 2

The function $f(n) = 2n + 1$ for natural numbers n can be defined recursively

$$f(0) = 1;\ \text{for every } n \in \mathbb{N},\ f(n+1) = f(n) + 2$$

Example: $EXPR$ is a language of legal algebraic expressions involving the identifier a, the binary operations $+$ and $*$, and parentheses,i.e.,

$$\Sigma = \{a, +, *, (,)\}.$$

Some of the strings in the language are

$$a, \ a + a * a, \ \text{and} \ (a + a * (a + a)).$$

Its recursive definition:

Example: $EXPR$ is a language of legal algebraic expressions involving the identifier a, the binary operations $+$ and $*$, and parentheses,i.e.,

$$\Sigma = \{a, +, *, (,)\}.$$

Some of the strings in the language are

$$a,\ \ a + a * a,\ \ \text{and}\ (a + a * (a + a)).$$

Its recursive definition:

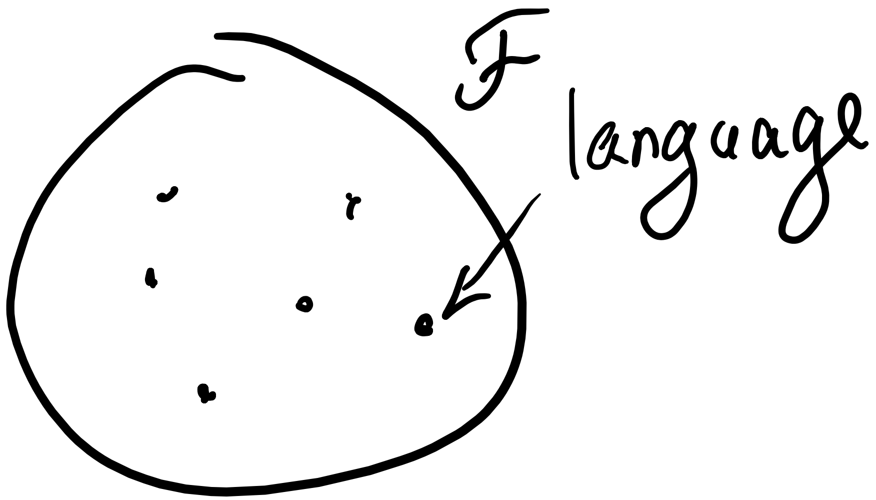
These are just strings, you don't need to compute them

1. $a \in EXPR$
2. $\forall x, y \in EXPR$, $x + y$ and $x * y \in EXPR$
3. $\forall x \in EXPR$, $(x) \in EXPR$

Example: a recursive definition of a set of languages over $\{a, b\}$. We denote by $\mathcal{F}$ the set $2^{\{a,b\}^*}$ (the set of languages over $\{a, b\}$) defined as follows:

set of subsets

1. $\emptyset$, $\{\Lambda\}$, $\{a\}$, and $\{b\}$ are elements of $\mathcal{F}$
2. $\forall L_1, L_2 \in \mathcal{F}$, $L_1 \cup L_2 \in \mathcal{F}$
3. $\forall L_1, L_2 \in \mathcal{F}$, $L_1 L_2 \in F$



Given language *L,* typical questions that one can ask are to (dis)prove that:

- string *x* belongs to *L*
- all (some) *x* in *L* have some property
- another language is a subset of L, etc.
- L* can contain certain strings
- recursive definition of L is …

(Example: compiler + source code.)

Mathematical induction with integers	Recursive definition of a language (or set)	Structural induction always works with recursive definition
Basis: Prove the proposition for the smallest integer(s).	**Basis:** Starts with the shortest string(s) that initiate generation of set. These strings cannot be decomposed or reduced.	**Basis:** Prove the proposition for the all the basis cases (shortst strings) of the recursive definition.
Hypothesis: Assume that the proposition is true for all integers up to a certain *k.*	**Condition**: Define strings that will participate in generation of new strings (or elements of set)	**Hypothesis**: Assume that the proposition is true for all strings generated until some step.
Induction: Prove the proposition for the next integer *k+1* using assumption about smaller integers.	**Recursive part**: Define rules to generate new strings (or elements of set) using the strings in condition,	**Induction**: Prove the proposition for the newly generated strings using rules in the recursive part.

1. $a \in EXPR$
2. $\forall x, y \in EXPR$, $x + y$ and $x * y \in EXPR$
3. $\forall x \in EXPR$, $(x) \in EXPR$

Suppose we want to prove that every string $x \in EXPR$ satisfies the statement $P(x)$.

The principle of **structural induction** uses recursive definition of language. It says that in order to show that $P(x)$ is true for every $x \in EXPR$, it is sufficient to show:

1. $P(a)$ is true
2. $\forall x, y \in EXPR$, if $P(x)$ and $P(y)$ are true, then $P(x + y)$ and $P(x * y)$ are true.
3. $\forall x \in EXPR$, if $P(x)$ is true, then $P((x))$ is true.

Induction hypothesis

operations on *x*, *y* generate strings for which P() is true length of *x*

Claim. *For every element x of $EXPR$, $|x|$ is odd.*

Proof.

- **Basis step.** We wish to show that $|a|$ is odd. This is true because $|a| = 1$.
- **Induction hypothesis.** $x, y \in EXPR$, and $|x|$,$|y|$ are odd.
- **Statement to be proved in the induction step.**

$$|x + y|,\ \ |x * y|,\ \ \text{and } |(x)| \text{ are odd.}$$

length of *x*

Claim. *For every element x of $EXPR$, $|x|$ is odd.*

Proof.

Proof of induction step.

- The lengths
$$|x + y| \text{ and } |x * y|$$
are both $|x|+|y|+1$, because the symbols of $x+y$ include those in x, those in y, and the additional “operator” symbol.
- The length $|(x)|$ is $|x| + 2$, because two parentheses have been added to the symbols of x.
- The first number is odd because the induction hypothesis implies that it is the sum of two odd numbers plus 1, and the second number is odd because the induction hypothesis implies that it is an odd number plus 2.

$\square$

Sometimes stronger statements are easier to prove

$EXPR$ is a language of legal algebraic expressions.

1. $a \in EXPR$
2. $\forall x, y \in EXPR$, $x + y$ and $x * y \in EXPR$
3. $\forall x \in EXPR$, $(x) \in EXPR$

Prove that no string in $EXPR$ can contain the substring $++$.

No problem with basis step, hypothesis, and concluding about $x*y$, and (x). Proving that $x+y$ doesn't contain $++$ is more difficult. Neither x nor y contains $++$ as a substring, but if x ended with $+$ or y started with $+$, then $++$ would occur in the concatenation.

Stronger statement: $\forall x \in EXPR$, x doesn't begin or end with $+$ **and** doesn't contain $++$. Complete proof at home.

Defining functions on sets defined recursively

- For $\Sigma = \{a, b\}$, we have $\{a, b\}^* = \{\Lambda, a, b, aa, ab, ba, bb, aaa, aab, \ldots\}$
- Recursive definition of set $S = \{a, b\}^*$ is

$$\Lambda \in S; \ \forall x \in S \ xa, xb \in S$$

- Example: let us consider the **reverse function** $r : \Sigma^* \to \Sigma^*$ that is defined recursively by referring to the recursive definition of Σ^*

$$r(x) = \begin{cases} \Lambda & x = \Lambda \\ ar(y) & x = ya, \ y \in \Sigma^* \\ br(y) & x = yb, \ y \in \Sigma^* \end{cases}$$

output of *r* if input of *r* is …

Reverse function *r(x)* reverses input string *x*. Example: *x = abac, r(x) = caba*

To illustrate the close relationship between the recursive definition of *{a, b}**, the recursive definition of *r*, and the principle of structural induction, we prove the following fact about the reverse function.

Claim. $\forall x, y \in \{a, b\}^*$, $r(xy) = r(y)r(x)$.

Note that we have two variables x, and y, i.e., "$\forall x, y$" is translated to "$\forall x$, and $\forall y$". In such cases the quantifiers are nested; we can write the statement in the form $\forall y\ P(y)$, where $P(y)$ is itself a quantified statement, $\forall x(...)$, and so we can attempt to use structural induction on y.

We prove $\forall y \in \Sigma^*$, $P(y)$ is true, where $P(y)$ is the statement "$\forall x \in \Sigma^*$, $r(xy) = r(y)r(x)$".

Proof. We prove $\forall y \in \Sigma^*$, $P(y)$ is true, where $P(y)$ is the statement "$\forall x \in \Sigma^*$, $r(xy) = r(y)r(x)$".

i.e., y is an empty string

Basic step. We need to prove the statement "$\forall x \in \Sigma^*$, $r(x\overset{\downarrow}{\Lambda}) = r(\Lambda)r(x)$." (complete at home)

Induction hypothesis. $y \in \Sigma^*$, and $\forall x \in \Sigma^*$, $r(xy) = r(y)r(x)$.

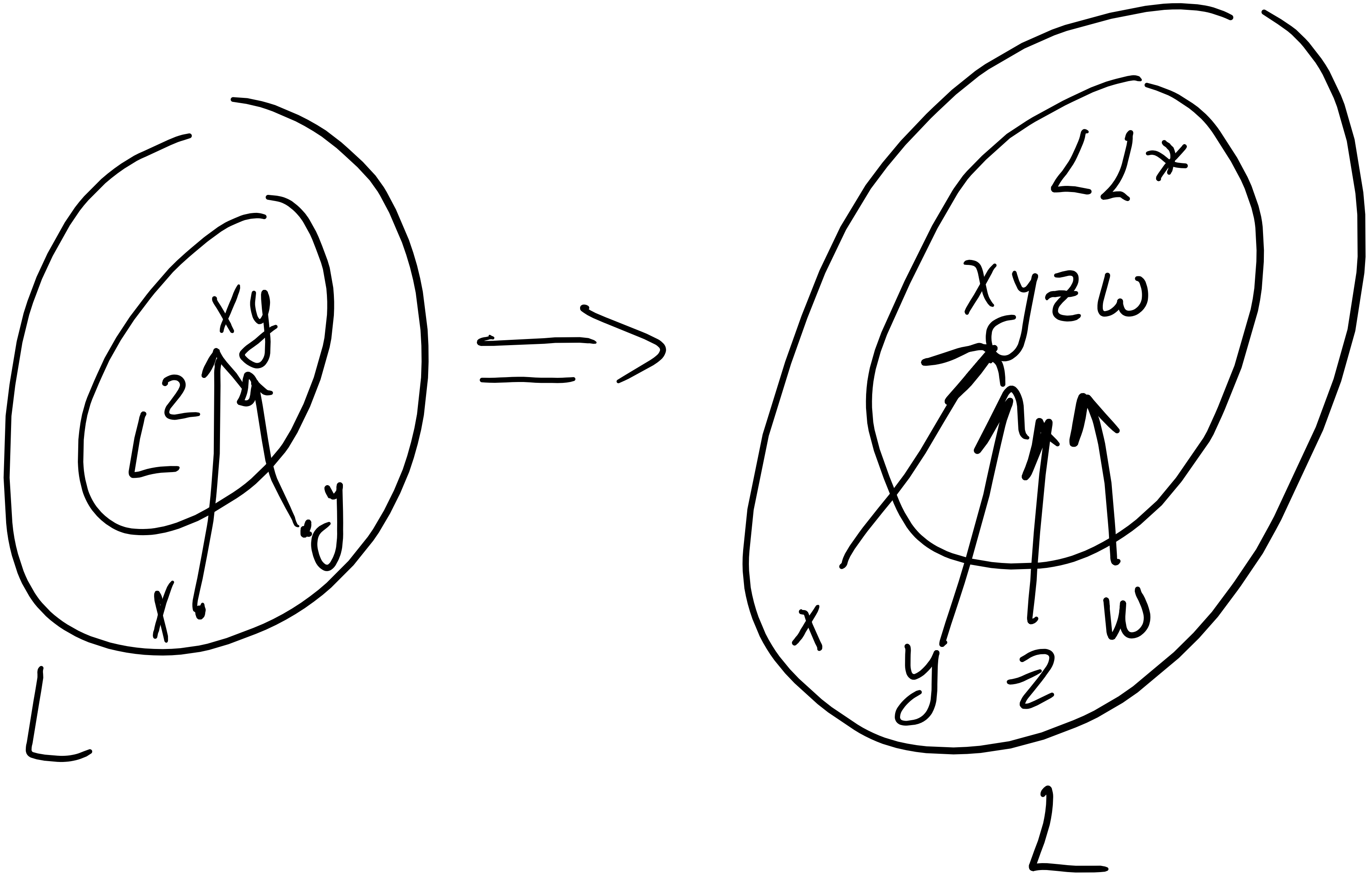
Statement to be proved in induction step. $\forall x \in \Sigma^*$, $r(x(ya)) = r(ya)r(x)$, and $r(x(yb)) = r(yb)r(x)$.

$$\begin{aligned}
r(x(ya)) &= r((xy)a) && \text{concatenation is associative} \\
&= a(r(xy)) && \text{by the definition of } r \\
&= a(r(y)r(x)) && \text{by the induction hypothesis} \\
&= (ar(y))r(x) && \text{concatenation is associative} \\
&= r(ya)r(x) && \text{by the definition of } r.
\end{aligned}$$

The second part of the proof is similar. Complete it in the assignment! □

Claim. *Prove that for every non-empty language $L \subseteq \{a,b\}^*$,*

$$if \quad L^2 \subseteq L, \quad then \quad LL^* \subseteq L.$$



Claim. *Prove that for every non-empty language $L \subseteq \{a, b\}^*$,*

$$if \quad L^2 \subseteq L, \quad then \quad LL^* \subseteq L.$$

Proof sketch. If $x \in LL^*$ then $x = x_1x_2$, where $x_1 \in L$, and $x_2 \in L^*$.

- if $x_2 = \Lambda$ - trivial; if $x = \Lambda$ - trivial
- otherwise $\exists y_1, \cdots, y_k \in L$ $s.t.$ $x = y_1y_2 \cdots y_k$, where $\forall y_i \neq \Lambda, 1 \leq i \leq k$.
- We prove by induction on k (the number of strings y_i)
  - Basis: k=1 - trivial
  - Hypothesis: the statement is true for $k - 1$
  - Induction: $x = y_1y_2 \cdots y_k = zy_k$, where $z = y_1y_2 \cdots y_{k-1}$. By the hypothesis $z \in L$, then $zy_k \in L^2$, and then $x \in L$, because $L^2 \subseteq L$.

You can also prove it by induction on L^k without splitting into substrings.

Claim. *Suppose $L \subseteq \{a, b\}^*$ is defined recursively as follows:*

$$\Lambda \in L; \forall x \in L \ ax, axb \in L.$$

Show that $L = L_0$, where $L_0 = \{a^i b^j | i \geq j\}$.

Proof sketch. Proving $L = L_0$ means that we need to prove both

$$\begin{tikzcd} L \subseteq L_0, & \text{and} & L_0 \subseteq L. \\ \forall x \in L \quad x \in L_0 \arrow[u] & & \forall x \in L_0 \quad x \in L \arrow[u] \end{tikzcd}$$

Intuition:

1. In *L* strings are generated with more a's than b's
2. In L_0 strings look like $\underbrace{aaaa\ldots aaaa}_{i} \underset{\geq}{} \underbrace{bb\ldots bb}_{j}$

Claim. *Suppose $L \subseteq \{a, b\}^*$ is defined recursively as follows:*

$$\Lambda \in L; \text{ for every } x \in L, \text{ both } ax \text{ and } axb \text{ are in } L.$$

Show that $L = L_0$, where $L_0 = \{a^i b^j | i \geq j\}$.

Proof sketch.

Part 2 ($L_0 \subseteq L$). We show by induction on n

$$\forall n \in \mathbb{N}, \text{ if } y \in L_0, \text{ and } |y| = n, \text{ then } y \in L.$$

Basis step: $y \in L_0$, and $|y| = 0 \Rightarrow y \in L$. This is true because if $|y| = 0$ then $y = \Lambda$, and $\Lambda \in L$.

Induction Hypothesis: $k \in \mathbb{N}, \forall y \in L_0$ such that

$$|y| \leq k \Rightarrow y \in L$$

Claim. *Suppose $L \subseteq \{a,b\}^*$ is defined recursively as follows:*

$$\Lambda \in L; \text{ for every } x \in L, \text{ both } ax \text{ and } axb \text{ are in } L.$$

Show that $L = L_0$, where $L_0 = \{a^i b^j | i \geq j\}$.

Proof sketch (cont).

IS: $y \in L_0$, $|y| = k+1 \Rightarrow y = a^i b^j$, and $i + j = k + 1$.

- $y \neq \Lambda$ because $k \geq 0$ and $|y| = k+1 \Rightarrow$ we must show that $y = ax$ or $y = axb$ for some $x \in L$.

Claim. *Suppose $L \subseteq \{a,b\}^*$ is defined recursively as follows:*

$$\Lambda \in L; \textit{ for every } x \in L, \textit{ both } ax \textit{ and } axb \textit{ are in } L.$$

Show that $L = L_0$, where $L_0 = \{a^i b^j | i \geq j\}$.

Proof sketch (cont).

IS: $y \in L_0$, $|y| = k+1 \Rightarrow y = a^i b^j$, and $i + j = k + 1$.

- $y \neq \Lambda$ because $k \geq 0$ and $|y| = k+1 \Rightarrow$ we must show that $y = ax$ or $y = axb$ for some $x \in L$.
- We know that $i + j > 0$, and $i \geq j \Rightarrow i > 0$, i.e., $y = ax$ for some x; if $j > 0$ then $y = axb$ for some x.

Claim. *Suppose $L \subseteq \{a,b\}^*$ is defined recursively as follows:*

$$\Lambda \in L; \text{ for every } x \in L, \text{ both } ax \text{ and } axb \text{ are in } L.$$

Show that $L = L_0$, where $L_0 = \{a^i b^j | i \geq j\}$.

Proof sketch (cont).

IS: $y \in L_0$, $|y| = k+1 \Rightarrow y = a^i b^j$, and $i + j = k + 1$.

- $y \neq \Lambda$ because $k \geq 0$ and $|y| = k+1 \Rightarrow$ we must show that $y = ax$ or $y = axb$ for some $x \in L$.
- We know that $i + j > 0$, and $i \geq j \Rightarrow i > 0$, i.e., $y = ax$ for some x; if $j > 0$ then $y = axb$ for some x.
- If $j > 0$ then $y = axb$, where $x = a^{i-1}b^{j-1}$, and $i - 1 \geq j - 1$, i.e., $x \in L_0$ and by IH $x \in L$ and then also $y \in L$. Case $j = 0$ is similar.

>>> Part 1 is easy: complete a home.

$\square$

Summary

- In typical proofs by mathematical induction, we choose an integer that is
  - The length of string
  - The number of substrings whose concatenation gives string *x*
  - The exponent of the language in the * of some expression

- Typical proofs by structural induction (SI):
  - SI doesn’t work without recursive definition (RD) of the language
  - The basis of SI corresponds to the basis of the RD
  - Formulate induction hypothesis on all input elements of the recursive rules
  - Prove induction step on all recursive rules in RD
  - Break down the problem you need to prove in induction step to easier problems in which you can apply the hypothesis, basis statement, etc.

Claim. *Suppose that $x, y \in \{a, b\}^*$ and neither Λ. Show that*

$xy = yx \;\Rightarrow\; \exists z \in \{a, b\}^*$, and $i, j \in \mathbb{N}$, such that $x = z^i$, and $y = z^j$.

Proof sketch. Let d be the greatest common divisor of $|x|$, and $|y|$. We rewrite x and y as

$$x = x_1 x_2 \cdots x_p, \text{ and } y = y_1 y_2 \cdots y_q,$$

where all $|x_i| = |y_j| = d$ for all i, j.

Since $xy = yx$ then $x^q y^p = y^p x^q$ (begin with $x^q y^p$, and run repeated transpositions, i.e., switch x and y).

$$xx \ldots xyy \ldots y = xx \ldots yxy \ldots y = \cdots = yy \ldots yxx \ldots x$$

Claim. *Suppose that $x, y \in \{a,b\}^*$ and neither Λ. Show that*

$$xy = yx \;\Rightarrow\; \exists z \in \{a,b\}^*, \text{ and } i, j \in \mathbb{N}, \text{ such that } x = z^i, \text{ and } y = z^j.$$

Proof sketch.

Both x^qy^p, and y^px^q have the same length $2pqd$,e.g.,

$$|x^qy^p| = |x_1 \cdots x_p \cdots q \text{ times } \cdots x_1 \cdots x_p y^p| = \cdots = 2pqd$$

- In both cases, prefixes x^q (of x^qy^p), and y^p (of y^px^q) have the same length and thus are equal.
- If $x^q = (x_1 \cdots x_p)^q$ then x_1 appears in positions 1, $pd+1$, $2pd+1$, ... , $(q-1)pd+1$.
- In y^p, the substring y_{r_i} of length d can be found at $ipd+1$, where $r_i = ip+1 \mod q$. Since p and q have no common factors, all r_i are different. Then, it follows that all y_i are equal (z). Same for x_i.