Why study formal language theory?

For some: for the beauty of theory

Understand tools used in practice
regular expressions/automata for string search
grammars: design programming or domain-specific languages

Understand complexity and limits of computation

Improve ability to analyze and solve problems

Example: Computer Viruses

A virus can be formally defined as a program that replicates itself.

Is it possible to build the perfect antivirus? (that detects any virus, with no false positives/negatives)

Fred Cohen showed in 1987 that the problem was undecidable.
The program $P$ under scrutiny could invoke any proposed decision procedure $D$ and infect other programs only if $D$ determines that $P$ is not a virus.

Example: Optimizing compilers

Is it possible to build the perfect optimizing compiler? (compiles a program to the shortest/simplest code)

If so, the optimized program should just
(1) read (part of) the input
(2) halt, or loop forever
thus it would have solved the halting problem!

Many fundamentally interesting problems in program analysis/testing/verification are undecidable for the same reason.

Selective outline

- Deterministic / nondeterministic automata, regular expressions
- Context-free grammars / pushdown automata
- Turing machines
- (Un)decidability, reducibility
- Complexity
  - time
  - space: polynomial, logarithmic
  - impact of nondeterminism
- More advanced topics
  - alternation, games, circuit complexity

Review: Sets

- Sets: unordered, unique elements
  - multisets, if number of occurrences matters \{2, 2, 3, 7, 7\}
  - finite vs. infinite sets (results may differ)
  - power set \(P(A) = \text{set of subsets of } A\ (2^{|A|}\text{ subsets})\)
  - Cartesian product / cross product:
    \[A \times B = \{(x, y) \mid x \in A, y \in B\}\]

Big Picture: Automata, Computability and Complexity

- Complexity Theory
  - What makes some problems computationally hard and others easy?
    Classification scheme according to computational difficulty
    Coping with complexity / change problem for easier solution
- Computability Theory
  - Limits of what can be computed
  - Limits of what can be proved
- Automata Theory
  - Formally define simple models of computation
    finite automata
    context-free grammars / pushdown automata

Logistics and Grading

- Homework: 30% (six homeworks)
- Midterm 1: 20% (Thu Feb 21, 7 pm, ILC S131)
- Midterm 2: 20% (Wed Apr 10, 7 pm, ILC S131)
- Final: 25% (Thu May 9, 10:30 am, Goessmann 20)
- Moodle quizzes: 5% (throughout semester)

Functions (mappings)

\[f : A \to B \quad A = \text{domain, } B = \text{range}\]

- injective: \(x_1 \neq x_2 \rightarrow f(x_1) \neq f(x_2)\)
- surjective (onto): \(\forall y \in B \exists x \in A : f(x) = y\)
- bijective (one-to-one correspondence): injective and surjective
Relations

Set of tuples \( \subseteq A_1 \times A_2 \times \ldots \times A_k \)

(k-ary relation, k-tuples)

A relation defines a predicate over \( A_1 \times A_2 \times \ldots \times A_k \)
true if tuple in relation, false if not

Binary relation: set of pairs \( \subseteq A \times B \)
Binary relation on a set \( A \): subset of \( A \times A \)

Equivalence relation on a set:

reflexive: \( \forall x : R(x, x) \)

symmetric: \( \forall x, y : R(x, y) \rightarrow R(y, x) \)

transitive: \( \forall x, y, z : R(x, y) \land R(y, z) \rightarrow R(x, z) \)

An equivalence relation partitions a set into disjoint subsets
  e.g., remainders \( \text{mod } n \), strongly connected components

Strings and Languages

Alphabet (usually \( \Sigma \)): any nonempty finite set

String: finite sequence of symbols from the alphabet
\( \Sigma^* \): set of all (finite) strings over \( \Sigma \), incl. empty string (denoted \( \epsilon \)).

Important: distinguish between:
finite: all strings in \( \Sigma^* \) are finite
bounded/unbounded: strings can be arbitrarily long
infinite: infinite strings are not in \( \Sigma^* \)

lexicographic order (dictionary order)
shortlex order: \( \epsilon, 0, 1, 00, 01, 10, 11, 000, \ldots \)
useful for enumerating strings

A language is an arbitrary subset of strings (of \( \Sigma^* \))

Preview: Decidability

A program can be viewed (encoded) as a string.

A (decision) problem can be viewed as a set of strings (inputs) for which the answer is YES.

Cantor’s theorem says there is no one-to-one mapping (bijection) between a set and its powerset.
  the powerset has “more” elements

Thus, there can be no one-to-one mapping between \( \Sigma^* \) (a superset of all programs) and \( P(\Sigma^*) \) (the set of all problems).
  \( \Rightarrow \) some problems must be undecidable

Graphs

\( G = (V, E) \): nodes/vertices, edges
directed / undirected: edges are ordered / unordered pairs
degree/in-/outdegree: number of edges (total/entering/leaving) node
path: sequence of nodes connected by edges (incl. empty)
strongly connected: a directed path between any two nodes

Eulerian path/cycle: contains each edge exactly once
Hamiltonian path/cycle: contains each vertex exactly once

Logic and proofs

Proof
informally: a convincing logical argument that a statement is true
formally: a sequence of true statements, which are either axioms, hypotheses, or are obtained from previous statements using deduction rules

Book gives clear convincing arguments without excessive formalism

But we must still practice being careful that all our steps are correct.

How to prove it ?

▶ Be patient
▶ Come back to it
let it sink in, let ideas develop
▶ Be neat
define notions clearly, be explicit about any deductions
▶ Be concise
the most beautiful proofs are often short and simple
**Proof by construction**

*Def.*: A graph is \( k \)-regular if every node has degree \( k \).

*Proof*: For any even \( n > 2 \) there is a 3-regular graph with \( n \) nodes.

Idea: try a “regular” construction: from each node, an edge going “left”, “middle”, and “right”.

Label nodes \( V = \{0, 1, \ldots, n - 1\} \). Visualize on a circle.

Construct edges:
- \((i - 1, i)\) (for \( i > 0 \)) and \((n - 1, 0)\), (left/right)
- and edges \((i, i + n/2)\) for \( i < n/2 \) (middle, to opposite nodes).

**Proof exercise: Ramsey’s Theorem**

*Def.*: A clique in a graph \( G \) is a subgraph in which any two nodes are connected by an edge.

An anti-clique (independent set) is a subgraph in which any two nodes are not connected by an edge.

*Proof*: Any graph with \( n \) nodes contains either a clique or an anti-clique with at least \( \frac{1}{4} \log_2 n \) nodes.

*Intuition*: \( \log_2 n \) suggests we need to successively halve the number of nodes.

<table>
<thead>
<tr>
<th>Proof by contradiction</th>
</tr>
</thead>
<tbody>
<tr>
<td>To prove ( P ), assume ( \neg P ) and derive a contradiction.</td>
</tr>
<tr>
<td>Closely related: <em>proof by contrapositive (indirect proof)</em></td>
</tr>
<tr>
<td>Contrapositive of ( P \to Q ) is ( \neg Q \to \neg P ).</td>
</tr>
<tr>
<td>The two are equivalent.</td>
</tr>
<tr>
<td>We could show ( \neg Q \to \neg P ) (by direct proof) and can then claim ( P \to Q ).</td>
</tr>
<tr>
<td>We also have ( \neg(P \to Q) = P \land \neg Q ).</td>
</tr>
<tr>
<td>Thus, if we assume ( P ) and ( \neg Q ) and derive a contradiction, we have proved ( P \to Q ).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Proof by induction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prove that all elements of an <em>infinite</em> set have a given property typically a predicate ( P(n) ) over naturals</td>
</tr>
<tr>
<td><strong>Ordinary induction</strong></td>
</tr>
<tr>
<td>If ( P(n_0) ) holds (typically ( n_0 = 0 ), or 1, etc.) and ( \forall n \geq n_0 : P(n) \to P(n+1) )</td>
</tr>
<tr>
<td>then ( \forall n \geq n_0 : P(n) ).</td>
</tr>
<tr>
<td><strong>Strong induction</strong></td>
</tr>
<tr>
<td>allows us to assume ( P(n) ) for all smaller values, not just previous:</td>
</tr>
<tr>
<td>If ( P(n_0) ) holds and ( (\forall k : n_0 \leq k \leq n \to P(k)) \to P(n+1) )</td>
</tr>
<tr>
<td>then ( \forall n \geq n_0 : P(n) ).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Where is the fallacy?</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Proof</em>: In any set of ( n ) horses, all have the same color.</td>
</tr>
<tr>
<td><em>Base case</em>: ( n = 1 ). One horse, clearly the same color.</td>
</tr>
<tr>
<td><em>Inductive step</em>: Assume true for ( n \geq 1 ), prove for ( n + 1 ).</td>
</tr>
<tr>
<td>Remove horse ( x ) from set of ( n + 1 ) horses. Remaining set has ( n ) horses, all of same color.</td>
</tr>
<tr>
<td>Now add back ( x ) and remove a different horse ( y ). Again ( n ) horses, all of same color.</td>
</tr>
<tr>
<td>Thus, adding ( y ) back we get ( n + 1 ) horses of same color, q.e.d.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Proof by construction (2)</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Proof</em>: There exist irrational numbers ( x ) and ( y ) so that ( xy ) is rational.</td>
</tr>
<tr>
<td>Finding a pair of numbers is non-obvious.</td>
</tr>
<tr>
<td>The first irrational number we usually learned of is ( \sqrt{2} ).</td>
</tr>
<tr>
<td>Why is it irrational?</td>
</tr>
<tr>
<td>Is ( \sqrt{2 \sqrt{2}} ) rational? Don’t know.</td>
</tr>
<tr>
<td>If it were (case 1), we have our ( x ) and ( y ).</td>
</tr>
<tr>
<td>Now assume ( \sqrt{2 \sqrt{2}} ) irrational. We have ( \sqrt{2 \sqrt{2}} = \sqrt{2} \sqrt{2} = 2 ) (rational),</td>
</tr>
<tr>
<td>so again we have our two numbers: ( \sqrt{2 \sqrt{2}} ) and ( \sqrt{2} ).</td>
</tr>
<tr>
<td>We’ve shown ( x ) and ( y ) must exist, without concretely finding them.</td>
</tr>
</tbody>
</table>
For next time

- Review notions in intro chapter
  (be familiar with terms in one-page glossary)
- Revisit any less familiar notions from CS 250 and 311
- Sign up for Piazza and Gradescope
- Review knowledge on finite automata