Formal Languages

- **Alphabet**: a finite set of symbols
- **String**: a finite sequence of symbols
- **Language**: a set of strings
- **String length**: number of symbols in it
- **String concatenation**: $w_1w_2$
- **Empty string**: $\varepsilon$ or $∅$
- **Language concatenation**: $L_1L_2 = \{w_1w_2 \mid w_1 \in L_1, w_2 \in L_2 \}$
- **String exponentiation**: $w^k = ww\ldots w$ (k times)
- **Language exponentiation**: $L^k = LL\ldots L$ (k times)

### Examples
- $S = \{a, b\}$
- $L = \{a, aa, aaa, \ldots\}$
- $\text{ababbaaab}$
- $|\text{aba}| = 3$
- $\text{ab} \cdot \text{ba} = \text{abba}$
- $\forall w \ w \cdot \varepsilon = \varepsilon \cdot w = w$
- $\{1, 2\} \cdot \{a, aa, \ldots\} = \{1a, 2a, 1aa, 2aa, \ldots\}$
- $a^3 = \text{aaa}$
- $\{0, 1\}^{32}$
Formal Languages

- **String reversal**: \( w^R \)  
  \[ (aabc)^R = cbaa \]

- **Language reversal**: \( L^R = \{ w^R \mid w \in L \} \)  
  \[ \{ ab, cd \}^R = \{ ba, dc \} \]

- **Kleene closure**: \( L^* = L^0 \cup L^1 \cup L^2 \cup L^3 \cup \ldots \)  
  \[ L^+ = L^1 \cup L^2 \cup L^3 \cup L^4 \cup \ldots \]

**Theorem**: \( L^+ = LL^* \)

- **Trivial language**: \( \{ \varepsilon \} \)

- **Empty language**: \( \emptyset \)

- **All finite strings** (over \( \Sigma \)): \( \Sigma^* \)  
  \( L \subseteq \Sigma^* \)  
  \( \forall L \)  
  \[ \{ a, aa, aaa, \ldots \} \]

**Theorem**: \( \Sigma^* \) is countable,  
\( |\Sigma^*| = |\mathbb{Z}| \)

**Theorem**: \( 2^{\Sigma^*} \) is uncountable.

**Theorem**: \( \Sigma^* \) contains no infinite strings.

**Theorem**: \( (L^*)^* = L^* \)

\[ L^* \subseteq (L^*)^* \ \& \ \ (L^*)^* \subseteq L^* \]
Finite Automata

Basic idea: a FA is a “machine” that changes states while processing symbols, one at a time.

- **Finite set of states:** \( Q = \{ q_0, q_1, q_3, ..., q_k \} \)
- **Transition function:** \( \delta: Q \times \Sigma \rightarrow Q \)
- **Initial state:** \( q_0 \in Q \)
- **Final states:** \( F \subseteq Q \)
- **Finite automaton is** \( M = (Q, \Sigma, \delta, q_0, F) \)

Ex: an FA that accepts all odd-length strings of zeros:

\[
M = (\{ q_0, q_1 \}, \{0\}, \{((q_0,0),q_1), ((q_1,0),q_0)\}, q_0, \{q_1\})
\]
Finite Automata

**FA operation**: consume a string $w \in \Sigma^*$ one symbol at a time while changing states

**Acceptance**: end up in a **final state**

**Rejection**: anything else (including hang-up / crash)

Ex: FA that accepts all strings of form abababab...$= (ab)^*$

$$M = (\{q_0, q_1\}, \{a, b\}, \{((q_0, a), q_1), ((q_1, b), q_0)\}, q_0, \{q_0\})$$

But $M$ “crashes” on input string “abba”!

Solution: add dead-end state to fully specify $M$

$$M' = (\{q_0, q_1, q_2\}, \{a, b\}, \{((q_0, a), q_1), ((q_1, b), q_0), ((q_0, b), q_2), ((q_1, a), q_2), ((q_2, a), q_2), ((q_2, b), q_2)\}, q_0, \{q_0\})$$
Finite Automata

Transition function $\delta$ extends from symbols to strings:

$$\delta: Q \times \Sigma^* \rightarrow Q$$

$$\delta(q_0, wx) = \delta(\delta(q_0, w), x)$$

where $\delta(q_i, \epsilon) = q_i$

Language of $M$ is $L(M) = \{ w \in \Sigma^* | \delta(q_0, w) \in F \}$

**Definition**: language is regular iff it is accepted by some FA.

**Theorem**: Complementation preserves regularity.

**Proof**: Invert final and non-final states in fully specified FA.

$L(M) = (ab)^*$

$L(M') = b(a+b)^* + (a+b)^*a + (a+b)^*(aa+bb)(a+b)^*$

$M'$ “simulates” $M$ and does the opposite!
Problem: design a DFA that accepts all strings over \{a,b\} where any a’s precede any b’s.

Idea: skip over any contiguous a’s, then skip over any b’s, and then accept iff the end is reached.

$L = a^*b^*$

Q: What is the complement of L?
Problem: what is the complement of $L = a^*b^*$?

Idea: write a regular expression and then simplify.

$L' = (a+b)^*b^+(a+b)^*a^+(a+b)^*$
$= (a+b)^*b(a+b)^*a(a+b)^*$
$= (a+b)^*b^+a(a+b)^*$
$= (a+b)^*ba(a+b)^*$
$= a^*b^+a(a+b)^*$
Finite Automata

Theorem: Intersection preserves regularity.

Proof: ("parallel" simulation):

- Construct all super-states, one per each state pair.
- New super-transition function jumps among super-states, simulating old transition function.
- Initial super state contains both old initial states.
- Final super states contains pairs of old final states.
- Resulting DFA accepts same language as original NFA (but size can be the product of two old sizes).

Given $M_1=(Q_1, \Sigma, \delta_1, q', F_1)$ and $M_2=(Q_2, \Sigma, \delta_2, q'', F_2)$

construct $M=(Q, \Sigma, \delta, q, F) \quad Q = Q_1 \times Q_2$

$F = F_1 \times F_2 \quad q=(q',q'')$

$\delta : Q \times \Sigma \rightarrow Q \quad \delta((q_i,q_j),x) = (\delta_1(q_i,x),\delta_2(q_j,x))$
Finite Automata

Theorem: Union preserves regularity.
Proof: De Morgan's law: \( L_1 \cup L_2 = \overline{L_1} \cap \overline{L_2} \)
Or cross-product construction, i.e.,
parallel simulation with \( F = (F_1 \times Q_2) \cup (Q_1 \times F_2) \)

Theorem: Set difference preserves regularity.
Proof: Set identity \( L_1 - L_2 = L_1 \cap \overline{L_2} \)
Or cross-product construction, i.e.,
parallel simulation with \( F = (F_1 \times (Q_2 - F_2)) \)

Theorem: XOR preserves regularity.
Proof: Set identity \( L_1 \oplus L_2 = (L_1 \cup L_2) - (L_1 \cap L_2) \)
Or cross-product construction, i.e.,
parallel simulation with \( F = (F_1 \times (Q_2 - F_2)) \cup ((Q_1 - F_1) \times F_2) \)

Meta-Theorem: Identity-based proofs are easier!
Finite Automata

Non-determinism: generalizes determinism, where many “next moves” are allowed at each step:

Old \( \delta: Q \times \Sigma \rightarrow Q \)

New \( \delta: 2^Q \times \Sigma \rightarrow 2^Q \)

Computation becomes a “tree”.

Acceptance: \( \exists \) a path from root (start state) to some leaf (a final state)

Ex: non-deterministically accept all strings where the 7\(^{th}\) symbol before the end is a “b”:

```
Input: a b a b b a a a  \Rightarrow  Accept!
```
Finite Automata

**Theorem**: Non-determinism in FAs doesn’t increase power.

**Proof**: by simulation:

- Construct all super-states, one per each state subset.
- New super-transition function jumps among super-states, simulating old transition function.
- Initial super state are those containing old initial state.
- Final super states are those containing old final states.
- Resulting DFA accepts the same language as original NFA, but can have exponentially more states.

Q: Why doesn’t this work for PDAs?
Finite Automata

Note: Powerset construction generalizes the cross-product construction. More general constructions are possible.

EC: Let $\text{HALF}(L)=\{v \mid \exists v,w \in \Sigma^* \exists |v|=|w| \text{ and } vw \in L\}$

Show that HALF preserves regularity.

A two way FA can move its head backwards on the input: $\delta: Q \times \Sigma \rightarrow Q \times \{\text{left, right}\}$

EC: Show that two-way FA are not more powerful than ordinary one-way FA.

$\varepsilon$-transitions: $q_i \xrightarrow{\varepsilon} q_j$

Theorem: $\varepsilon$-transitions don’t increase FA recognition power.

Proof: Simulate $\varepsilon$-transitions FA without using $\varepsilon$-transitions. i.e., consider $\varepsilon$-transitions to be a form of non-determinism.
The movie “Next” (2007) Based on the science fiction story “The Golden Man” by Philip Dick

Premise: a man with the super power of non-determinism!

At any given moment his reality branches into multiple directions, and he can choose the branch that he prefers!

Extra credit!
Top-10 Reasons to Study Non-determinism

1. Helps us understand the ubiquitous concept of *parallelism* / concurrency;

2. Illuminates the structure of problems;

3. Can help *save time & effort* by solving intractable problems more efficiently;

4. Enables vast, deep, and general studies of “*completeness*” theories;

5. Helps explain why verifying proofs & solutions seems to be easier than *constructing* them;
Why Study Non-determinism?

6. Gave rise to new and novel mathematical approaches, proofs, and analyses;

7. Robustly decouples / abstracts complexity from underlying computational models;

8. Gives disciplined techniques for identifying “hardest” problems / languages;

9. Forged new unifications between computer science, math & logic;

10. Non-determinism is interesting fun, and cool!
Regular Expressions

Regular expressions are defined recursively as follows:

- $\emptyset$: empty set
- $\{\varepsilon\}$: trivial language
- $\{x\} \forall x \in \Sigma$: singleton language language

Inductively, if $R$ and $S$ are regular expressions, then so are:

- $(R+S)$: union
- $RS$: concatenation
- $R^*$: Kleene closure

Examples:
- $aa(a+b)^*bb$
- $(a+b)^*b(a+b)^*a(a+b)^*$

Theorem: Any regular expression is accepted by some FA.
Regular Expressions

A FA for a regular expression can be built by composition:

Ex: all strings over \( S = \{a, b\} \) where \( \exists \) a “b” preceding an “a”

\[ (a+b)^* b (a+b)^* a (a+b)^* \]

\[ = (a+b)^* ba (a+b)^* \]

Why?

Remove previous start/final states
FA Minimization

Idea: “Equivalent” states can be merged:

16 states!

3 states!
FA Minimization

**Theorem** [Hopcroft 1971]: the number $N$ of states in a FA can be minimized within time $O(N \log N)$.

Based on earlier work [Huffman 1954] & [Moore 1956].

**Conjecture**: Minimizing the number of states in a **nondeterministic** FA can not be done in **polynomial time**.

**Theorem**: Minimizing the number of states in a **pushdown automaton** (or TM) is **undecidable**.

**Idea**: implement a **finite automaton minimization tool**

- Try to design it to run reasonably **efficiently**
- Consider also including:
  - A **regular-expression-to-FA** transformer
  - A **non-deterministic-to-deterministic** FA converter
Theorem: Any FA accepts a language denoted by some RE.

Proof: Use “generalized finite automata” where a transition can be a regular expression (not just a symbol), and:

Only 1 super start state and 1 (separate) super final state.

Each state has transitions to all other states (including itself), except the super start state, with no incoming transitions, and the super final state, which has no outgoing transitions.
FAs and Regular Expressions

Now reduce the size of the GFA by one state at each step. A transformation step is as follows:

Such a transformation step is always possible, until the GFA has only two states, the super-start and super-final states:

Corollary: FAs and REs denote the same class of languages.
Regular Expressions Identities

- $R+S = S+R$
- $R(ST) = (RS)T$
- $R(S+T) = RS+RT$
- $(R+S)T = RT+ST$
- $\emptyset^* = \varepsilon^* = \varepsilon$
- $R+\emptyset = \emptyset+R = R$
- $R\varepsilon = \varepsilon R = R$  \hspace{1cm} $R+\varepsilon \neq R$
- $(R^*)^* = R^*$ \hspace{1cm} $R\emptyset \neq R$
- $(\varepsilon + R)^* = R^*$
- $(R^*S^*)^* = (R+S)^*$

WHENEVER I LEARN A NEW SKILL I CONCOCT ELABORATE FANTASY SCENARIOS WHERE IT LETS ME SAVE THE DAY.

OH NO! THE KILLER MUST HAVE FOLLOWED HER ON VACATION!

BUT TO FIND THEM WE’D HAVE TO SEARCH THROUGH 200 MB OF EMAILS LOOKING FOR SOMETHING FORMATTED LIKE AN ADDRESS!

EVERYBODY STAND BACK.

I KNOW REGULAR EXPRESSIONS...
Decidable Finite Automata Problems

Def: A problem is decidable if $\exists$ an algorithm which can determine (in finite time) the correct answer for any instance.

Given a finite automata $M_1$ and $M_2$:

Q₁: Is $L(M_1) = \emptyset$ ?
Hint: graph reachability

Q₂: Is $L(M_2)$ infinite ?
Hint: cycle detection

Q₃: Is $L(M_1) = L(M_2)$ ?
Hint: consider $L_1 - L_2$ and $L_2 - L_1$
Regular Expression Minimization

**Problem**: find smallest equivalent regular expression
- Decidable (why?)
- Hard: PSPACE-complete

Turing Machine Minimization

**Problem**: find smallest equivalent Turing machine
- Not decidable (why?)
- Not even recognizable (why?)
JFLAP Version 7.0
RELEASED August 28, 2009

JFLAP

JFLAP is software for experimenting with formal languages topics, including non-deterministic finite automata, nondeterministic pushdown automata, multi-tape Turing machines, several types of grammars, parsing, and E-systems. In addition to constructing and testing examples for these, JFLAP allows one to experiment with construction proofs from one form to another, such as converting an NFA to a DFA to a minimal state DFA to a regular expression or regular grammar. Click here for more information on what one can do with JFLAP.

JFLAP News

- *** September 28, 2009 ***
  Note to Mac users. Macs default to Java 1.5 and JFLAP now requires Java 1.6.
- *** August 28, 2009 *** - JFLAP 7.0 released. See below for more details on changes. Get JFLAP

- December 2008 - JFLAP CD by Linz and Rodger with JFLAP exercises that goes along with the Linz book is now available. Click here for info and see the Supplement.

Note, we have worked closely with Linz over the past several years so JFLAP fits nicely with the Linz book. Eventually we hope to publish a textbook with JFLAP integrated in...

- July 2008 - JFLAP now has a wiki where users can discuss the use or modifications of JFLAP, see jflap.wikia.com

- July 2008 - JFLAP now has two listservs. To join, go to lists.duke.edu
  You do not need to be a member of Duke to join the listserv.
  - jflap-announce@duke.edu - for announcements on new releases of JFLAP or new info on the JFLAP web page
Non-deterministic states are highlighted.
Context-Free Grammars

Basic idea: set of production rules induces a language

- **Finite set of variables:** $V = \{V_1, V_2, ..., V_k\}$
- **Finite set of terminals:** $T = \{t_1, t_2, ..., t_j\}$
- **Finite set of productions:** $P$
- **Start symbol:** $S$
- **Productions:** $V_i \rightarrow \Delta$ where $V_i \in V$ and $\Delta \in (V \cup T)^*$

Applying $V_i \rightarrow \Delta$ to $\alpha V_i \beta$ yields: $\alpha \Delta \beta$

Note: productions do not depend on "context" - hence the name "context free"!
Context-Free Grammars

Def: A language is context-free if it is accepted by some context-free grammar.

Theorem: All regular languages are context-free.

Theorem: Some context-free languages are not regular.

Ex: \( \{0^n1^n \mid n > 0\} \)

Proof by "pumping" argument: long strings in a regular language contain a pumpable substring.

\[ \exists N \in \mathbb{N} \ \exists \forall z \in L, \ |z| \geq N \ \exists u,v,w \in \Sigma^* \ \forall z = uvw, |uv| \leq N, |v| \geq 1, uv^i w \in L \ \forall i \geq 0. \]

Theorem: Some languages are not context-free.

Ex: \( \{0^n1^n2^n \mid n > 0\} \)

Proof by "pumping" argument for CFL’s.
Ambiguity

**Def:** A grammar is **ambiguous** if some string in its language has two non-isomorphic derivations.

**Theorem:** Some context-free grammars are **ambiguous**.

**Ex:** $G_1$: $S \rightarrow SS | a | \epsilon$

Derivation 1: $S \rightarrow SS \rightarrow aa$

Derivation 2: $S \rightarrow SS \rightarrow SSS \rightarrow aa$

**Def:** A context-free language is **inherently ambiguous** if every context-free grammar for it is **ambiguous**.

**Theorem:** Some context-free languages are **inherently ambiguous** (i.e., no non-ambiguous CFG exists).

**Ex:** $\{a^m b^n c^m d^m \mid m>0, n>0\} \cup \{a^n b^m c^n d^m \mid m>0, n>0\}$
Example: design a context-free grammar for strings representing all well-balanced parenthesis.


\[ G_1: S \rightarrow SS \mid (S) \mid \varepsilon \]

Ex: \[ S \rightarrow SS \rightarrow (S)(S) \rightarrow ()() \]
\[ S \rightarrow (S) \rightarrow ((S)) \rightarrow (()()) \]
\[ S \rightarrow (S) \rightarrow (SS) \rightarrow \ldots \rightarrow ()(()())()()) \]

Q: Is \( G_1 \) ambiguous?

Another grammar:

\[ G_2: S \rightarrow (S)S \mid \varepsilon \]

Q: Is \( L(G_1) = L(G_2) \) ?

Q: Is \( G_2 \) ambiguous?
Example: design a context-free grammar that generates all valid regular expressions.

Idea: embed the RE rules in a grammar.

\[
G: \quad S \rightarrow a \quad \text{for each } a \in \Sigma_L \\
S \rightarrow (S) \mid SS \mid S^* \mid S+S
\]

\[
S \rightarrow S^* \rightarrow (S)^* \rightarrow (S+S)^* \rightarrow (a+b)^*
\]

\[
S \rightarrow SS \rightarrow SSSS \rightarrow abS^*b \rightarrow aba^*a
\]

Q: Is G ambiguous?
Pushdown Automata

Basic idea: a pushdown automaton is a finite automaton that can optionally write to an unbounded stack.

- **Finite set of states:** \( Q = \{ q_0, q_1, q_3, \ldots, q_k \} \)
- **Input alphabet:** \( \Sigma \)
- **Stack alphabet:** \( \Gamma \)
- **Transition function:** \( \delta: Q \times (\Sigma \cup \{ \varepsilon \}) \times \Gamma \rightarrow 2^{Q \times \Gamma^*} \)
- **Initial state:** \( q_0 \in Q \)
- **Final states:** \( F \subseteq Q \)

Pushdown automaton is \( M = (Q, \Sigma, \Gamma, \delta, q_0, F) \)

Note: pushdown automata are non-deterministic!
A **pushdown automaton** can use its stack as an unbounded but access-controlled (last-in/first-out or LIFO) storage.

- A PDA accesses its stack using “push” and “pop”
- Stack & input alphabets may differ.
- Input read head only goes 1-way.
- Acceptance can be by final state or by empty-stack.

Note: a PDA can be made **deterministic** by restricting its transition function to unique next moves:

$$\delta: Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to Q \times \Gamma^*$$
Pushdown Automata

**Theorem:** If a language is accepted by some context-free grammar, then it is also accepted by some PDA.

**Theorem:** If a language is accepted by some PDA, then it is also accepted by some context-free grammar.

**Corollary:** A language is context-free iff it is also accepted by some pushdown automaton.

I.E., context-free grammars and PDAs have equivalent “computation power” or “expressiveness” capability.
Closure Properties of CFLs

**Theorem:** The context-free languages are *closed* under *union*.

Hint: Derive a new grammar for the union.

**Theorem:** The CFLs are *closed* under *Kleene* closure.

Hint: Derive a new grammar for the Kleene closure.

**Theorem:** The CFLs are *closed* under $\cap$ with regular langs.

Hint: Simulate PDA and FA in parallel.

**Theorem:** The CFLs are *not closed* under *intersection*.

Hint: Find a counterexample.

**Theorem:** The CFLs are *not closed* under *complementation*.

Hint: Use De Morgan’s law.
Decidable PDA / CFG Problems

Given an arbitrary pushdown automata $M$ (or CFG $G$) the following problems are decidable (i.e., have algorithms):

- $Q_1$: Is $L(M) = \emptyset$?
- $Q_5$: Is $L(G) = \emptyset$?
- $Q_2$: Is $L(M)$ finite?
- $Q_6$: Is $L(G)$ finite?
- $Q_3$: Is $L(M)$ infinite?
- $Q_7$: Is $L(G)$ infinite?
- $Q_4$: Is $w \in L(M)$?
- $Q_8$: Is $w \in L(G)$?

Extra-credit: Prove each!
Theorem: the following are undecidable (i.e., there exist no algorithms to answer these questions):

Q: Is PDA M minimal?
Q: Are PDAs $M_1$ and $M_2$ equivalent?
Q: Is CFG G minimal?
Q: Is CFG G ambiguous?
Q: Is $L(G_1) = L(G_2)$?
Q: Is $L(G_1) \cap L(G_2) = \emptyset$?
Q: Is CFL L inherently ambiguous?
PDA Enhancements

Theorem: 2-way PDAs are more powerful than 1-way PDAs.

Hint: Find an example non-CFL accepted by a 2-way PDA.

Theorem: 2-stack PDAs are more powerful than 1-stack PDAs.

Hint: Find an example non-CFL accepted by a 2-stack PDA.

Theorem: 1-queue PDAs are more powerful than 1-stack PDAs.

Hint: Find an example non-CFL accepted by a 1-queue PDA.

Theorem: 2-head PDAs are more powerful than 1-head PDAs.

Hint: Find an example non-CFL accepted by a 2-head PDA.

Theorem: Non-determinism increases the power of PDAs.

Hint: Find a CFL not accepted by any deterministic PDA.
Turing Machines

Basic idea: a Turing machine is a finite automaton that can optionally write to an unbounded tape.

- **Finite set of states**: $Q = \{ q_0, q_1, q_3, \ldots, q_k \}$
- **Tape alphabet**: $\Gamma$
- **Blank symbol**: $\beta \in \Gamma$
- **Input alphabet**: $\Sigma \subseteq \Gamma - \{ \beta \}$
- **Transition function**: $\delta: (Q - F) \times \Gamma \rightarrow Q \times \Gamma \times \{ L, R \}$
- **Initial state**: $q_0 \in Q$
- **Final states**: $F \subseteq Q$

Turing machine is $M = (Q, \Gamma, \beta, \Sigma, \delta, q_0, F)$
A Turing machine can use its tape as an unbounded storage but reads / writes only at head position.

- Initially the entire tape is blank, except the input portion
- Read / write head goes left / right with each transition
- A Turing machine is usually deterministic
- Input string acceptance is by final state(s)
**Turing Machine “Enhancements”**

Larger alphabet:

old: \( \Sigma = \{0,1\} \)   \[ \text{new: } \Sigma' = \{a,b,c,d\} \]

**Idea:** Encode larger alphabet using smaller one.

Encoding example: \( a=00, b=01, c=10, d=11 \)

![Diagram of Turing Machine Enhancements](attachment:image.png)
Turing Machine “Enhancements”

Double-sided infinite tape:

Idea: Fold into a normal single-sided infinite tape

old: $\delta$

new: $\delta'$
Turing Machine “Enhancements”

Multiple heads:

Idea: Mark heads locations on tape and simulate

Modified $\delta'$ processes each “virtual” head independently:
- Each move of $\delta$ is simulated by a long scan & update
- $\delta'$ updates & marks all “virtual” head positions
Turing Machine “Enhancements”

Multiple tapes:

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Idea: Interlace multiple tapes into a single tape

Modified $\delta'$ processes each “virtual” tape independently:

- Each move of $\delta$ is simulated by a long scan & update
- $\delta'$ updates R/W head positions on all “virtual tapes”
Turing Machine “Enhancements”

Two-dimensional tape:

```
1 1 0 1 0 1
0 1 1 1 0 1
1 0 1 1 1 0 0
```

Idea: Flatten 2-D tape into a 1-D tape

Modified 1-D $\delta'$ simulates the original 2-D $\delta$:

- Left/right $\delta$ moves: $\delta'$ moves horizontally
- Up/down $\delta$ moves: $\delta'$ jumps between tape sections

This is how compilers implement 2D arrays!
Turing Machine “Enhancements”

Non-determinism:

Idea: Parallel-simulate non-deterministic threads

Modified deterministic $\delta'$ simulates the original ND $\delta$:

- Each ND move by $\delta$ spawns another independent “thread”
- All current threads are simulated “in parallel”
Theorem: Combinations of “enhancements” do not increase the power of Turing machines.

Idea: “Enhancements” are independent (and commutative with respect to preserving the language recognized).
Def: A language is Turing-decidable iff it is exactly the set of strings accepted by some always-halting TM.

\[ w \in \Sigma^* = \{ a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb, aaaa, \ldots \} \]

\[ M(w) \Rightarrow \begin{array}{cccccccccccccc}
\checkmark & \times & \checkmark & \times & \times & \times & \checkmark & \times & \times & \times & \times & \times & \checkmark & \ldots
\end{array} \]

\[ L(M) = \{ a, aa, aaaa, \ldots \} \]

Note: M must always halt on every input.
Turing -Recognizable vs. -Decidable

**Def:** A language is **Turing-recognizable** iff it is exactly the set of strings accepted by some Turing machine.

- **Input** $w \rightarrow$ $\sqrt{\text{Accept}} \& \text{halt}$ $\times \equiv \infty \text{Reject} \& \text{halt}$ $\infty \text{Run forever}$

$L(M) = \{ a, aa, \text{aaa, aaaa...} \}$

**Note:** $M$ can run forever on an input, which is implicitly a reject (since it is not an accept).
Recognition vs. Enumeration

Def: “Decidable” means “Turing-decidable”

“Recognizable” means “Turing-recognizable”

Theorem: Every decidable language is also recognizable.

Theorem: Some recognizable languages are not decidable.

Ex: The halting problem is recognizable but not decidable.

Note: Decidability is a special case of recognizability.

Note: It is easier to recognize than to decide.
“A wrong decision is better than indecision.”

“I'm the decider, and I decide what is best.”
Famous Deciders
Recognition and Enumeration

Def: An “enumerator” Turing machine for a language L prints out precisely all strings of L on its output tape.

Note: The order of enumeration may be arbitrary.

Theorem: If a language is \textbf{decidable}, it can be enumerated in \textbf{lexicographic} order by some Turing machine.

Theorem: If a language can be enumerated in \textbf{lexicographic} order by some TM, it is \textbf{decidable}.
Recognition and Enumeration

Def: An “enumerator” Turing machine for a language \( L \) prints out precisely all strings of \( L \) on its output tape.

\[ a \$, a, b, $, b, b, a, $ \ldots \]

Note: The order of enumeration may be arbitrary.

Theorem: If a language is recognizable, then it can be enumerated by some Turing machine.

Theorem: If a language can be enumerated by some TM, then it is recognizable.
<table>
<thead>
<tr>
<th>The Alphabet</th>
<th>In Alphabetical Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aitch</td>
<td>Ex</td>
</tr>
<tr>
<td>Are</td>
<td>Eye</td>
</tr>
<tr>
<td>Ay</td>
<td>Gee</td>
</tr>
<tr>
<td>Bee</td>
<td>Jay</td>
</tr>
<tr>
<td>Cue</td>
<td>Kay</td>
</tr>
<tr>
<td>Dee</td>
<td>Oh</td>
</tr>
<tr>
<td>Double U</td>
<td>Pea</td>
</tr>
<tr>
<td>Ee</td>
<td>See</td>
</tr>
<tr>
<td>Ef</td>
<td>Tee</td>
</tr>
<tr>
<td>El</td>
<td>Vee</td>
</tr>
<tr>
<td>Em</td>
<td>Wy</td>
</tr>
<tr>
<td>En</td>
<td>Yu</td>
</tr>
<tr>
<td>Ess</td>
<td>Zee</td>
</tr>
</tbody>
</table>
**Def:** A language is **Turing-decidable** iff it is exactly the set of strings accepted by some **always-halting** TM.

**Theorem:** The finite languages are decidable.

**Theorem:** The regular languages are decidable.

**Theorem:** The context-free languages are decidable.
A “Simple” Example

Let $S = \{ x^3 + y^3 + z^3 \mid x, y, z \in \mathbb{Z} \}$

Q: Is $S$ infinite?
A: Yes, since $S$ contains all cubes.

Q: Is $S$ Turing-recognizable?
A: Yes, since dovetailing TM can enumerate $S$.

Q: Is $S$ Turing-decidable?
A: Unknown!

Q: Is $29 \in S$?
A: Yes, since $3^3 + 1^3 + 1^3 = 29$

Q: Is $30 \in S$?
A: Yes, since $(2220422932)^3 + (-2218888517)^3 + (-283059965)^3 = 30$

Q: Is $33 \in S$?
A: Unknown!

Theorem [Matiyasevich, 1970]: Hilbert’s 10th problem (1900), namely of determining whether a given Diophantine (i.e., multi-variable polynomial) equation has any integer solutions, is not decidable.
Closure Properties of Decidable Languages

Theorem: The decidable languages are closed under union.
   Hint: use simulation.

Theorem: The decidable languages are closed under $\cap$.
   Hint: use simulation.

Theorem: The decidable languages are closed under complement.
   Hint: simulate and negate.

Theorem: The decidable languages are closed under concatenation.
   Hint: guess-factor string and simulate.

Theorem: The decidable languages are closed under Kleene star.
   Hint: guess-factor string and simulate.
Closure Properties of Recognizable Languages

Theorem: The recognizable languages are closed under union.
   Hint: use simulation.

Theorem: The recognizable languages are closed under \( \cap \).
   Hint: use simulation.

Theorem: The recognizable langs are not closed under compl.
   Hint: reduction from halting problem.

Theorem: The recognizable langs are closed under concat.
   Hint: guess-factor string and simulate.

Theorem: The recognizable langs are closed under Kleene star.
   Hint: guess-factor string and simulate.
Reducibilities

Def: A language $A$ is reducible to a language $B$ if

$\exists$ computable function/map $f:\Sigma^* \rightarrow \Sigma^*$ where

$\forall w \quad w \in A \iff f(w) \in B$

Note: $f$ is called a “reduction” of $A$ to $B$

Denotation: $A \leq B$

Intuitively, $A$ is “no harder” than $B$
Reducibilities

Def: A language $A$ is reducible to a language $B$ if there exists a computable function/map $f: \Sigma^* \rightarrow \Sigma^*$ where

$$\forall w \quad w \in A \iff f(w) \in B$$

Theorem: If $A \leq B$ and $B$ is decidable then $A$ is decidable.

Theorem: If $A \leq B$ and $A$ is undecidable then $B$ is undecidable.

Note: be very careful about the mapping direction!
Reduction Example 1

Def: Let $H_\varepsilon$ be the halting problem for TMs running on $w=\varepsilon$.

"Does TM $M$ halt on $\varepsilon$?" $H_\varepsilon = \{ <M> \in \Sigma^* | M(\varepsilon) \text{ halts} \}$

Theorem: $H_\varepsilon$ is not decidable.

Proof: Reduction from the Halting Problem $H$:

Given an arbitrary TM $M$ and input $w$, construct new TM $M'$ that **if it ran** on input $x$, it would:

1. Overwrite $x$ with the **fixed** $w$ on tape;
2. Simulate $M$ on the **fixed** input $w$;
3. Accept $\iff M$ accepts $w$.

**Note:** $M'$ is not run!

**Note:** $M'$ halts on $\varepsilon$ (and on any $x \in \Sigma^*$) $\iff M$ halts on $w$.

A decider (oracle) for $H_\varepsilon$ can thus be used to decide $H$!

Since $H$ is undecidable, $H_\varepsilon$ must be undecidable also.
Reduction Example 2

**Def:** Let $L_\emptyset$ be the emptiness problem for TMs.

"Is $L(M)$ empty?" $L_\emptyset = \{ <M> \in \Sigma^* | L(M) = \emptyset \}$

**Theorem:** $L_\emptyset$ is not decidable.

**Proof:** Reduction from the Halting Problem $H$:
Given an arbitrary TM $M$ and input $w$, construct new TM $M'$ that if it ran on input $x$, it would:

1. Overwrite $x$ with the fixed $w$ on tape;
2. Simulate $M$ on the fixed input $w$;
3. Accept $\iff M$ accepts $w$.

Note: $M'$ is not run!

Note: $M'$ halts on every $x \in \Sigma^* \iff M$ halts on $w$.

A decider (oracle) for $L_\emptyset$ can thus be used to decide $H$!
Since $H$ is undecidable, $L_\emptyset$ must be undecidable also.
Reduction Example 3

Def: Let $L_{\text{reg}}$ be the regularity problem for TMs.

“Is $L(M)$ regular?” $L_{\text{reg}} = \{ <M> \in \sum^* | L(M) \text{ is regular} \}$

Theorem: $L_{\text{reg}}$ is not decidable.

Proof: Reduction from the Halting Problem H:

Given an arbitrary TM $M$ and input $w$, construct new TM $M'$ that if it ran on input $x$, it would:

1. Accept if $x \in 0^n1^n$
2. Overwrite $x$ with the fixed $w$ on tape;
3. Simulate $M$ on the fixed input $w$;
4. Accept $\iff$ $M$ accepts $w$.

Note: $M'$ is not run!

Note: $L(M') = \sum^* \iff M$ halts on $w$

$L(M') = 0^n1^n \iff M$ does not halt on $w$

A decider (oracle) for $L_{\text{reg}}$ can thus be used to decide H!
Rice’s Theorem

Def: Let a “property” $P$ be a set of recognizable languages.

Ex: $P_1 = \{ L \mid L$ is a decidable language $\}$
$P_2 = \{ L \mid L$ is a context-free language $\}$
$P_3 = \{ L \mid L = L^* \}$
$P_4 = \{ \{ \epsilon \} \}$
$P_5 = \emptyset$
$P_6 = \{ L \mid L$ is a recognizable language $\}$

$L$ is said to “have property $P$” iff $L \in P$

Ex: $(a+b)^*$ has property $P_1$, $P_2$, $P_3$ & $P_6$ but not $P_4$ or $P_5$
$\{ww^R\}$ has property $P_1$, $P_2$, & $P_6$ but not $P_3$, $P_4$ or $P_5$

Def: A property is “trivial” iff it is empty or it contains all recognizable languages.
Rice’s Theorem

Theorem: The two trivial properties are decidable.

Proof:

\[ P_{\text{none}} = \emptyset \]

\[ M_{\text{none}} \text{ decides } P_{\text{none}} \]

\[ P_{\text{all}} = \{ L \mid L \text{ is a recognizable language} \} \]

\[ M_{\text{all}} \text{ decides } P_{\text{all}} \]

Q: What other properties (other than \( P_{\text{none}} \) and \( P_{\text{all}} \)) are decidable?

A: None!
Rice’s Theorem

Theorem [Rice, 1951]: All non-trivial properties of the Turing-recognizable languages are not decidable.

Proof: Let $P$ be a non-trivial property.

Without loss of generality assume $\emptyset \notin P$, otherwise substitute $P$’s complement for $P$ in the remainder of this proof.

Select $L \in P$ (note that $L \neq \emptyset$ since $\emptyset \notin P$), and let $M_L$ recognize $L$ (i.e., $L(M_L) = L \neq \emptyset$).

Assume (towards contradiction) that $\exists$ some TM $M_P$ which decides property $P$:

Note: $x$ can be e.g., a TM description.

Does the language denoted by $<x>$ have property $P$? $M_P$

$\rightarrow$ yes

$\rightarrow$ no
Rice’s Theorem

Reduction strategy: use $M_p$ to “solve” the halting problem.
Recall that $L \in P$, and let $M_L$ recognize $L$ (i.e., $L(M_L) = L \neq \emptyset$).

Given an arbitrary TM $M$ & string $w$, construct $M'$:

What is the language of $M'$?
$L(M')$ is either $\emptyset$ or $L(M_L) = L$
If $M$ halts on $w$ then $L(M') = L(M_L) = L$
If $M$ does not halt on $w$ then $L(M') = \emptyset$ since $M_L$ never starts

=> $M$ halts on $w$ iff $L(M')$ has property $P$

“Oracle” $M_p$ can determine if $L(M')$ has property $P$, and thereby “solve” the halting problem, a contradiction!
Rice’s Theorem

Corollary: The following questions are not decidable: given a TM, is its language \( L \):

- Empty?
- Finite?
- Infinite?
- Co-finite?
- Regular?
- Context-free?
- Inherently ambiguous?
- Decidable?
- \( L = \Sigma^* \)?
- \( L \) contains an odd string?
- \( L \) contains a palindrome?
- \( L = \{ \text{Hello, World} \} \)?
- \( L \) is NP-complete?
- \( L \) is in PSPACE?

Warning: Rice’s theorem applies to properties (i.e., sets of languages), not (directly to) TM’s or other object types!
Context-Sensitive Grammars

Problem: design a context-sensitive grammar to accept the (non-context-free) language \( \{ 1^n$1^{2n} \mid n \geq 1 \} \)

Idea: generate \( n \) 1’s to the left & to the right of \( $ \); then double \( n \) times the # of 1’s on the right.

S → 1ND1E  /* Base case; E marks end-of-string */
N → 1ND | $  /* Loop: \( n \) 1’s and \( n \) D’s; end with $ */
D1 → 11D  /* Each D doubles the 1’s on right */
DE → E  /* The E “cancels” out the D’s */
E → ε  /* Process ends when the E vanishes */
Example: Generating strings in \( \{1^n$1^{2n} | n \geq 1\} \)

\[
\begin{align*}
S & \rightarrow 1ND1E \\
N & \rightarrow 1ND | $ \\
D1 & \rightarrow 11D \\
DE & \rightarrow E \\
E & \rightarrow \varepsilon
\end{align*}
\]

\[
\begin{align*}
S & \rightarrow 1ND1E \\
& \rightarrow 11ND1E \\
& \rightarrow 11ND11DE \\
& \rightarrow 111ND11DE \\
& \rightarrow 111ND11D1DE \\
& \rightarrow 111N11D1D1DE \\
& \rightarrow 111$11D1D1E \\
& \rightarrow 111$111D1D1E \\
& \rightarrow 111$1111D1DE \\
& \rightarrow 111$11111D1DE \\
& \rightarrow 111$111111DDE \\
& \rightarrow 111$1111111DE \\
& \rightarrow 111$1111111E \\
& \rightarrow 111$1111111\varepsilon
\end{align*}
\]

\[= 1^3$1^8 = 1^3$1^{2^3}\]
“But this is the simplified version for the general public.”