Formal Methods in Software Engineering

Theory of Computation, Spring 2026

Konstantin Chukharev

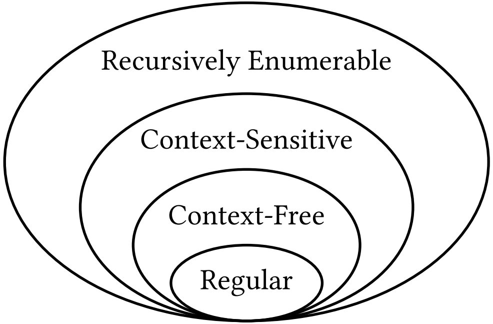
Languages

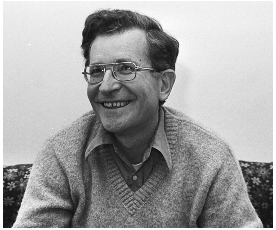
Formal Languages

Definition 1 (Formal language): A set of strings over an alphabet Σ, closed under concatenation.

Formal languages are classified by *Chomsky hierarchy*:

- Type 0: Recursively Enumerable
- Type 1: Context-Sensitive
- Type 2: Context-Free
- Type 3: Regular





Noam Chomsky

Examples:

- $L = \{a^n \mid n \geq 0\}$
- $L = \{a^n b^n \mid n \geq 0\}$
- $L = \{a^n b^n c^n \mid n \geq 0\}$
- $L = \{\langle M, w \rangle \mid M \text{ is a TM that halts on input } w\}$

Decision Problems as Languages

Definition 2 (Decision problem): A *decision problem* is a question with a "yes" or "no" answer.

Formally, the set of inputs for which the problem has an answer "yes" corresponds to a subset $L \subseteq \Sigma^*$, where Σ is an alphabet.

Example: SAT Problem as a language:

$$\mathrm{SAT} = \{\varphi \mid \varphi \text{ is a satisfiable Boolean formula}\}$$

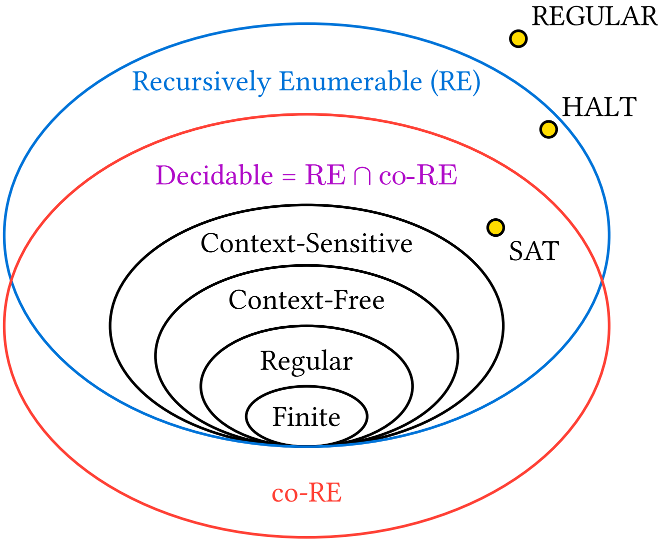
Example: Validity Problem as a language:

$$\mathrm{VALID} = \{\varphi \mid \varphi \text{ is a valid logical formula (tautology)}\}$$

Example: Halting Problem as a language:

$$\mathrm{HALT} = \{\langle M, w\rangle \mid \text{Turing machine } M \text{ halts on input } w\}$$

Language Classes



Machines

Finite Automata

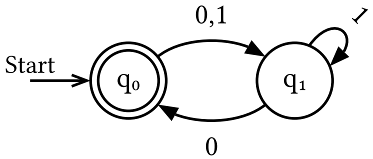
Definition 3: Deterministic Finite Automaton (DFA) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where:

- Q is a *finite* set of states,
- Σ is an *alphabet* (finite set of input symbols),
- $\delta : Q \times \Sigma \to Q$ is a *transition function*,
- $q_0 \in Q$ is the *start* state,
- $F \subseteq Q$ is a set of *accepting* states.

DFA recognizes *regular* languages (Type 3).

Example: Automaton $\mathcal{A}$ recognizing strings with an even number of 0s, $\mathcal{L}(\mathcal{A}) = \{0^n \mid n \text{ is even}\}$.

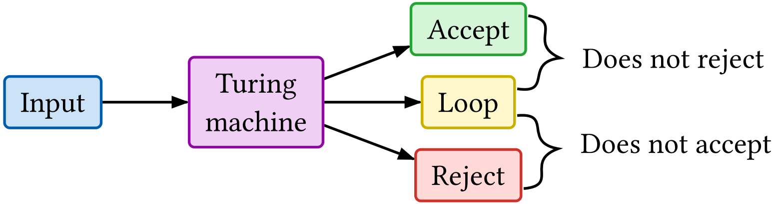
	0	1
q0	q1	q1
q1	q0	q1



Turing Machines

Informally, a Turing machine is a *finite-state* machine with an *infinite tape* and a *head* that can read and write symbols. Initially, the tape contains the *input* string, the rest are blanks, and the machine is in the *start* state. At each step, the machine reads the symbol under the head, changes the state, writes a new symbol, and moves the head left or right. When the machine reaches the *accept* or *reject* state, it immediately halts.

Note: If the machine never reaches the *accept* or *reject* state, it *loops* forever.



TM Formal Definition

Definition 4: Turing Machine (TM) is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$ where:

- Γ is a *tape alphabet* (including blank symbol $\square \in \Gamma$),
- $\Sigma \subseteq \Gamma$ is a *input alphabet*,
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is a transition function,
- q_{acc} and q_{rej} are the *accept* and *reject* states.

TM recognizes *recursively enumerable* languages (Type 0).

TM Language

Definition 5: The language *recognized* by a TM M, denoted $\mathcal{L}(M)$, is the set of strings $w \in \Sigma^*$ that M accepts, that is, for which M halts in the *accept* state.

- For any $w \in \mathcal{L}(M)$, M accepts w.
- For any $w \notin \mathcal{L}(M)$, M does not accept w, that is, M either *rejects* w or *loops forever* on w.

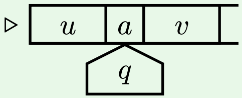
Definition 6: A TM is a *decider* if it halts on all inputs.

TM Configuration

Definition 7: A *configuration* of a TM is a *string* $(u; q; v)$ where $u, v \in \Gamma^*$, $q \in Q$, meaning:

- Tape contents: uv followed by the blanks.
- Current state is q.
- Head position: at the first symbol of v.

For example, configuration $(u; q; av)$, where $a \in \Gamma$, is represented as follows:



TM Computation

Definition 8 (Computation): The process of *computation* by a TM on input $w \in \Sigma^*$ is a *sequence* of configurations $C_1, C_2, ..., C_n$.

- $C_1 = (\triangleright; q_0; w)$ is the *start* configuration with input $w \in \Sigma^*$.
- $C_i \Rightarrow C_{i+1}$ for each i.
- C_n is a *final* configuration.

Configuration C_1 *yields* C_2, denoted $C_1 \Rightarrow C_2$, if TM can move from C_1 to C_2 in *one* step.

- See the formal definition on the next slide.

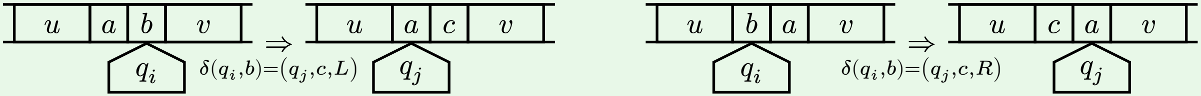
The relation $\Rightarrow^*$ is the *reflexive* and *transitive* closure of $\Rightarrow$.

- $C_1 \Rightarrow^* C_2$ denotes "yields in *some* number of steps".

TM Yields Relation

Definition 9 (Yields): Let $u, v \in \Gamma^*$, $a, b, c \in \Gamma$, $q_i, q_j \in Q$.

- Move left: $(ua; q_i; bv) \Rightarrow \left(u; q_j; acv\right)$ if $\delta(q_i, b) = \left(q_j, c, L\right)$ (overwrite b with c, move left)
- Move right: $(u; q_i; bav) \Rightarrow \left(uc; q_j; av\right)$ if $\delta(q_i, b) = \left(q_j, c, R\right)$ (overwrite b with c, move right)



Special case for the left end:

- $(\triangleright; q_i; bv) \Rightarrow \left(\triangleright; q_j; cv\right)$ if $\delta(q_i, b) = \left(q_j, c, L\right)$ (overwrite b with c, do not move).

Recognizing vs Deciding

There are *two* types of Turing machines:

1. Total TM: always halts. Also called *decider*.
2. General TM: may loop forever. Also called *recognizer*.

Definition 10 (Recognition): A TM *recognizes* a language L, if it halts and accepts all words $w \in L$, but no others. A language recognized by a TM is called *semi-decidable* or *recursively enumerable* or *recursively computable* or *Turing-recognizable*. The set of all recognizable languages is denoted by **RE**.

Definition 11 (Decision): A TM *decides* a language L, if it halts and accepts all words $w \in L$, and halts and rejects any other word $w \notin L$. A language decided by a TM is called *decidable* or *recursive* or *computable*. The set of all decidable languages is denoted by **R**.

MIU. MU?

Definition 12 (MIU system): The *MIU system* is a "formal system" consisting of:

- an alphabet $\Sigma = \{\mathtt{M}, \mathtt{I}, \mathtt{U}\}$,
- a single axiom: `MI`,
- a set of inference rules:

Rule	Description	Example
$x\mathtt{I} \vdash x\mathtt{IU}$	add `U` to the end of any string ending with `I`	`MI` to `MIU`
$\mathtt{M}x \vdash \mathtt{M}xx$	double the string after M	`MIU` to `MIUIU`
$x\mathtt{III}y \vdash x\mathtt{U}y$	replace any `III` with `U`	`MUIIIU` to `MUUU`
$x\mathtt{UU}y \vdash xy$	remove any UU	`MUUU` to `MU`

Question: Is `MU` a theorem of the MIU system?

Complexity

P and NP

Definition 13: Class P consists of problems that can be *solved* in *polynomial time*.

Equivalently, $L \in P$ iff L is *decidable* in polynomial time by a *deterministic* TM.

Examples: Shortest path, primality testing (AKS algorithm), linear programming.

Definition 14: Class NP consists of problems where a *certificate* of a solution ("yes" answer) can be *verified* in polynomial time.

Equivalently, $L \in \mathrm{NP}$ iff L is *decidable* in polynomial time by a *non-deterministic* TM.

Equivalently, $L \in \mathrm{NP}$ iff L is *recognizable* in polynomial time by a *deterministic* TM.

Examples: SAT, graph coloring, graph isomorphism, subset sum, knapsack, vertex cover, clique.

NP-Hard and NP-Complete

Definition 15: A problem H is *NP-hard* if every problem $L \in \mathrm{NP}$ is polynomial-time *reducible* to H.

Examples: Halting problem (undecidable), Traveling Salesman Problem (TSP).

Definition 16: A problem H is *NP-complete* if:

1. $H \in \mathrm{NP}$
2. H is NP-hard

Examples: SAT, 3-SAT, Hamiltonian path… Actually, almost all NP problems are NP-complete!

Theorem 1 (Cook–Levin): SAT is NP-complete.

co-NP

Definition 17: Complexity class co-NP contains problems where *"no"* instances can be *verified* in *polynomial time*.

Equivalently, $L \in$ co-NP iff the complement of L is in NP:

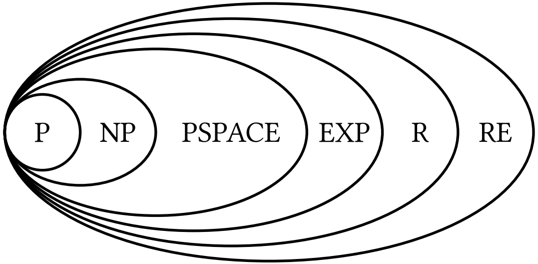
$$\text{co-NP} = \left\{ L \mid \overline{L} \in \text{NP} \right\}$$

Open question: NP $\stackrel{?}{=}$ co-NP? Implies P $\neq$ NP if false.

Examples:
- **VALID**: Check if a Boolean formula is always true (tautology).
- **UNSAT**: Check if a formula has no satisfying assignment.

Computational Hierarchy

$\mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{PSPACE} \subseteq \mathrm{EXP} \subset \mathrm{R} \subset \mathrm{RE}$



- **RE**
  Languages *accepted* (*recognized*) by any TM.
- **R** = RE ∩ co-RE
  Languages *decided* by any TM (always halt).
- **EXP**
  Languages *decided* by a *deterministic* TM in *exponential time*.
- **PSPACE**
  Languages *decided* by a *deterministic* TM in *polynomial space*.
- **NP**
  Languages *accepted* (*recognized*) by any TM, or *decided* by a *non-deterministic* TM, in *polynomial time*.
- **P**
  Languages *decided* by a *deterministic* TM in *polynomial time*.

Complexity Zoo

TODO

See also: https://complexityzoo.net/Petting_Zoo

Computability

Computable Functions

Definition 18 (Church–Turing thesis): *Every effectively computable function* — anything that *can* be computed by a mechanical, step-by-step procedure — *is computable by a Turing machine.*

This is a **thesis**, not a theorem. "Effectively computable" is an informal, intuitive notion; we cannot formally *prove* the thesis, but no counterexample has ever been found.

Note: In 1936, Alonzo Church (λ-calculus) and Alan Turing (Turing machines) independently formalized computability. They proved these models equivalent — and *every other model proposed since* computes exactly the same class of functions.

Definition 19 (Computable function): A partial function $f : \mathbb{N}^k \hookrightarrow \mathbb{N}$ is *computable* ("can be calculated") if there exists a computer program with the following properties:

- If $f(x)$ is defined, then the program terminates on the input x with the value $f(x)$ stored in memory.
- If $f(x)$ is undefined, then the program never terminates on the input x.

Effective Procedures

Definition 20 (Effective procedure): An *effective procedure* is a finite, deterministic, mechanical algorithm that guarantees to terminate and produce the correct answer in a finite number of steps.

An algorithm (set of instructions) is called an *effective procedure* if it:

- Consists of *exact*, finite steps.
- Always *terminates* in finite time.
- Produces the *correct* answer for given inputs.
- Requires no external assistance to execute.
- Can be performed *manually*, with pencil and paper.

Definition 21: A function is *computable* if there exists an effective procedure that computes it.

Examples of Computable Functions

Examples:

- The function $f(x) = x^2$ is computable.
- The function $f(x) = x!$ is computable.
- The function $f(n) =$ "n-th prime number" is computable.
- The function $f(n) =$ "the n-th digit of π" is computable.
- The Ackermann function is computable.
- The function that answers the question "Does God exist?" is computable.
- If the Collatz conjecture is true, the stopping time (number of steps to reach 1) of any n is computable.

Decidability

Decidable Sets

Definition 22 (Decidable set): Given a universal set $\mathcal{U}$, a set $S \subseteq \mathcal{U}$ is *decidable* (or *computable*, or *recursive*) if there exists a computable function $f : \mathcal{U} \to \{0, 1\}$ such that $f(x) = 1$ iff $x \in S$.

Examples:

- The set of all WFFs is decidable.
  - *We can check if a given string is well-formed by recursively verifying the syntax rules.*
- For a given finite set Γ of WFFs, the set $\{\alpha \mid \Gamma \vDash \alpha\}$ of all tautological consequences of Γ is decidable.
  - *We can decide $\Gamma \vDash \alpha$ using a truth table algorithm by enumerating all possible interpretations (at most $2^{|\Gamma|}$) and checking if each satisfies all formulas in Γ.*
- The set of all tautologies is decidable.
  - *It is the set of all tautological consequences of the empty set.*

Undecidable Sets

Definition 23 (Undecidable set): A set S is *undecidable* if it is not decidable.

Example: The existence of *undecidable* sets of expressions can be shown as follows.

An algorithm is completely determined by its *finite* description. Thus, there are only *countably many* effective procedures. But there are uncountably many sets of expressions. (Why? The set of expressions is countably infinite. Therefore, its power set is uncountable.) Hence, there are *more* sets of expressions than there are possible effective procedures.

Undecidability

Halting Problem

Definition 24 (Halting problem ↗): Given a program P and an input x, determine whether P halts on x (stops after finite time) or loops forever.

Theorem 2 (Turing, 1936): The halting problem is undecidable.

Proof *sketch*: Suppose there exists a procedure H that decides the halting problem. We can construct a program P that takes itself as input and runs H on it. If H says that P halts, then P enters an infinite loop. If H says that P does not halt, then P halts. This leads to a contradiction, proving that H cannot exist. $\square$

Halting Problem Pseudocode

```python
def halts(P, x) -> bool:
  """
  Returns True if program P halts on input x.
  Returns False if P loops forever.
  """


def self_halts(P):
  if halts(P, P):
    while True: # loop forever
  else:
    return # halt
```

Observe that `halts(self_halts, self_halts)` cannot return neither `True` nor `False`. **Contradition!**

Thus, the `halts` *does not exist* (cannot be implemented), and thus the halting problem is *undecidable*.

Post Correspondence Problem

Definition 25 (Post correspondence problem): Given two finite lists $a_1, \ldots, a_n$ and $b_1, \ldots, b_n$ of strings (over the alphabet with at least two symbols), determine whether there exists a sequence of indices $i_1, \ldots, i_k$, such that $a_{i_1} \ldots a_{i_k} = b_{i_1} \ldots b_{i_k}$.

Example: Let $A = [a, ab, bba]$, $B = [baa, aa, bb]$. A solution is $(3, 2, 3, 1)$:

$$a_3 a_2 a_3 a_1 = bba \cdot ab \cdot bba \cdot a = bbaabbbaa = bb \cdot aa \cdot bb \cdot baa = b_3 b_2 b_3 b_1$$

An alternative formulation of PCP is a collection of *dominoes*, each with a *top* and a *bottom* half, with an unlimited supply of each block, and the goal is to find a sequence of blocks such that the string formed by the *top* halves is equal to the string formed by the *bottom* halves.

bba	ab	bba	a
bb	aa	bb	baa
$i_1 = 3$	$i_2 = 2$	$i_3 = 3$	$i_4 = 1$

Semi-decidability

Semi-decidability

Suppose we want to determine $\Sigma \vDash \alpha$, where Σ is infinite. In general, it is *undecidable*.

Definition 26 (Semi-decidable set): A set S is *computably enumerable* if there is an *enumeration procedure* which lists, in some order, every member of S: $s_1, s_2, s_3 \ldots$

Equivalently (see Theorem 3), a set S is *semi-decidable* if there is an algorithm such that the set of inputs for which the algorithm halts is exactly S.

Note: There are more synonyms for *computably enumerable*, such as *effectively enumerable*, *recursively enumerable* (do not confuse with just *recursive*!), and *Turing-recognizable*, or simply *recorgizable*.

Note: If S is infinite, the enumeration procedure will *never* finish, but every member of S will be listed *eventually*, after some finite amount of time.

Note: Some properties of *decidable* and *semi-decidable* sets:

- Decidable sets are closed under union, intersection, Cartesian product, and complement.
- Semi-decidable sets are closed under union, intersection, and Cartesian product.

Enumerability and Semi-decidability

Theorem 3: A set S is computably enumerable iff it is semi-decidable.

Proof *(⇒)*: *If S is computably enumerable, then it is semi-decidable.*

Since S is computably enumerable, we can check if $\alpha \in S$ by enumerating all members of S and checking if α is among them. If it is, we answer "yes"; otherwise, we continue enumerating. Thus, if $\alpha \in S$, the procedure produces "yes". If $\alpha \notin S$, the procedure runs forever. □

Enumerability and Semi-decidability [2]

Proof *($\Leftarrow$)*: *If S is semi-decidable, then it is computably enumerable.*

Suppose we have a procedure P which, given α, terminates and produces "yes" iff $\alpha \in S$. To show that S is computably enumerable, we can proceed as follows.

1. Construct a systematic enumeration of *all* expressions (for example, by listing all strings over the alphabet in length-lexicographical order): $\beta_1, \beta_2, \beta_3, \ldots$
2. Break the procedure P into a finite number of "steps" (for example, by program instructions).
3. Run the procedure on each expression in turn, for an increasing number of steps (see dovetailing):
   - Run P on β_1 for 1 step.
   - Run P on β_1 for 2 steps, then on β_2 for 2 steps.
   - …
   - Run P on each of $\beta_1, \ldots, \beta_n$ for n steps each.
   - …
4. If P produces "yes" for some β_i, output (yield) β_i and continue enumerating.

This procedure will eventually list all members of S, thus S is computably enumerable. $\square$

Dual Enumerability and Decidability

Theorem 4: A set is decidable iff both it and its complement are semi-decidable.

Proof *(⇒)*: *If A is decidable, then both A and its complement $\overline{A}$ are effectively enumerable.*

Since A is decidable, there exists an effective procedure P that halts on all inputs and returs “yes” if $\alpha \in A$ and “no” otherwise.

To enumerate A:
- Systematically generate all expressions $\alpha_1, \alpha_2, \alpha_3, \ldots$
- For each α_i, run P. If P outputs “yes”, yield α_i. Otherwise, continue.

Similarly, enumerate $\overline{A}$ by yielding α_i when P outputs “no”.

Both enumerations are effective, since P always halts, so A and its complement are semi-decidable. □

Dual Enumerability and Decidability [2]

Proof *($\Leftarrow$)*: *If both A and its complement $\overline{A}$ are effectively enumerable, then A is decidable.*

Let E be an enumerator for A and $\overline{E}$ an enumerator for $\overline{A}$.

To decide if $\alpha \in A$, *interleave* the execution of E and $\overline{E}$, that is, for $n = 1, 2, 3, \ldots$

- Run E for n steps and if it produces α, *halt* and output "yes".
- Run $\overline{E}$ for n steps and if it produces α, *halt* and output "no".

Since α is either in A or in $\overline{A}$, one of the enumerators will eventually produce α. The interleaving with increasing number of steps ensures fair scheduling without starvation.

Remark: The "dovetailing" technique (alternating between enumerators with increasing step) avoids infinite waiting while maintaining finite memory requirements. The alternative is to run both enumerators *simultaneosly*, in parallel, using, for example, two computers. □

Enumerability of Tautological Consequences

Theorem 5: If Σ is an effectively enumerable set of WFFs, then the set $\{\alpha \mid \Sigma \vDash \alpha\}$ of tautological consequences of Σ is effectively enumerable.

Proof: Consider an enumeration of the elements of Σ: $\sigma_1, \sigma_2, \sigma_3, \ldots$

By the compactness theorem, $\Sigma \vDash \alpha$ iff $\{\sigma_1, \ldots, \sigma_n\} \vDash \alpha$ for some n.

Hence, it is sufficient to successively test (using truth tables)

$$\begin{aligned} \varnothing &\vDash \alpha, \\ \{\sigma_1\} &\vDash \alpha, \\ \{\sigma_1, \sigma_2\} &\vDash \alpha, \end{aligned}$$

and so on. If any of these tests succeeds (each is decidable), then $\Sigma \vDash \alpha$.

This demonstrates that there is an effective procedure that, given any WFF α, will output "yes" iff α is a tautological consequence of Σ. Thus, the set of tautological consequences of Σ is effectively enumerable. $\square$

Universal Machines

Universal Turing Machine

A *universal Turing machine* is a Turing machine that is capable of computing any computable sequence. [1]

Definition 27: A *universal Turing machine* U_{TM} is a Turing machine that can simulate any other TM.

High-level description of a universal Turing machine U_{TM}:

- Given an input $\langle M, w \rangle$, where M is a TM and $w \in \Sigma^*$:
  - Run (simulate a computation of) M on w.
  - If M halts and accepts w, U_{TM} accepts $\langle M, w \rangle$.
  - If M halts and rejects w, U_{TM} rejects $\langle M, w \rangle$.
  - *Implicitly*, if M loops on w, U_{TM} loops on $\langle M, w \rangle$.

Definition 28: The *language of a universal Turing machine* U_{TM} is the set A_{TM} of all pairs (M, w) such that M is a TM and M accepts w.

$$A_{\text{TM}} = \mathcal{L}(U_{\text{TM}}) = \{\langle M, w \rangle \mid M \text{ is a TM and } w \in \mathcal{L}(M)\}$$

Diagonalization Language

Consider all possible Turing machines, listed in some order, and all strings that are valid TM descriptions:

$$\langle M_0 \rangle, \langle M_1 \rangle, \ldots$$

Definition 29: Construct the *diagonalization language* L_Δ of all TMs that do not accept their own description:

$$L_\Delta = \mathcal{L}(M_\Delta) = \{\langle M \rangle \mid M \text{ is a TM and } \langle M \rangle \notin \mathcal{L}(M)\}$$

Note: M_Δ is *not* listed in the table, since its behavior differs from each other M_i at least on input $\langle M_i \rangle$.

	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	…
M_0	Acc	No	No	Acc	No	…
M_1	Acc	Acc	Acc	Acc	Acc	…
M_2	Acc	Acc	No	No	No	…
M_3	No	Acc	Acc	Acc	Acc	…
M_4	No	Acc	No	No	No	…
$\vdots$	…	…	…	…	…	…
M_Δ	No	No	Acc	No	Acc	…

Diagonalization Language is not Recognizable

$L_\Delta = \{\langle M \rangle \mid \langle M \rangle \notin \mathcal{L}(M)\}$

Theorem 6: $L_\Delta \notin \mathrm{RE}$.

Proof: Suppose L_Δ is recognizable. Then there exists a recognizer R such that $\mathcal{L}(R) = L_\Delta$.

It is the case that either $\langle R \rangle \notin \mathcal{L}(R)$ or $\langle R \rangle \in \mathcal{L}(R)$.

1. $\langle R \rangle \notin \mathcal{L}(R)$. Thus, $\langle R \rangle \in L_\Delta$. Since $\mathcal{L}(R) = L_\Delta$, $\langle R \rangle \notin \mathcal{L}(R)$. Contradiction.
2. $\langle R \rangle \in \mathcal{L}(R)$. Thus, $\langle R \rangle \notin L_\Delta$. Since $\mathcal{L}(R) = L_\Delta$, $\langle R \rangle \in \mathcal{L}(R)$. Contradiction.

In either case, we reach a contradiction. Therefore, the initial assumption that L_Δ is recognizable must be false. Thus, L_Δ is not recognizable. $\square$

Universal Language

$A_{\text{TM}} = \mathcal{L}(U_{\text{TM}}) = \{\langle M, w\rangle \mid M \text{ is a TM and } w \in \mathcal{L}(M)\}$

Theorem 7: $A_{\text{TM}} \in \text{RE}$.

Proof: U_{TM} is a TM that recognizes A_{TM}. □

Theorem 8: $\overline{A}_{\text{TM}} \notin \text{RE}$

Proof: $L_\Delta \leq_M \overline{A}_{\text{TM}}$. Build a recognizer (impossible) for L_Δ using a (hypothetical) recognizer for $\overline{A}_{\text{TM}}$. □

Theorem 9: $A_{\text{TM}} \notin \text{R}$.

Proof: R is closed under complement. A language A is decidable iff it is both recognizable ($A \in \text{RE}$) and co-recognizable ($\overline{A} \in \text{RE}$). We know that $\overline{A}_{\text{TM}} \notin \text{RE}$, thus A_{TM} cannot be decidable. □

Reductions

Mapping Reductions

TODO

Extremely Hard Problem

Regular languages are decidable. Some Turing machines accept regular languages and some do not.

Definition 30: Let **REGULAR** be the language of all TMs that accept regular languages.

$$\mathrm{REGULAR}_{\mathrm{TM}} = \{\langle M \rangle \mid \mathcal{L}(M) \text{ is regular}\}$$

This language is *neither* recognizable nor co-recognizable. (See theorems on the next slides.)

- *No computer program can confirm that a given Turing machine has a regular language.*
- *No computer program can confirm that a given Turing machine has a non-regular language.*
- *This problem is beyond the limits of what computers can ever do.*

REGULAR is not Recognizable

Theorem 10: $\mathrm{REGULAR}_{\mathrm{TM}} \notin \mathrm{RE}$.

Proof: $L_{\Delta} \leq_M \mathrm{REGULAR}_{\mathrm{TM}}$. □

REGULAR is not even co-Recognizable

Theorem 11: $\text{REGULAR}_{\text{TM}} \notin \text{co-RE}$

Proof: $\overline{L}_{\Delta} \leq_M \text{REGULAR}_{\text{TM}}$. $\square$

Rice’s Theorem

Rice's Theorem

Rice's theorem shows that *any* non-trivial property of the language recognized by a Turing machine is undecidable.

Definition 31 (Semantic Property): A property P of Turing machines is *semantic* (or a *property of languages*) if whenever $\mathcal{L}(M_1) = \mathcal{L}(M_2)$, then $P(M_1) \iff P(M_2)$.

A semantic property is *non-trivial* if some TMs satisfy it and others do not.

Example:
- "$\mathcal{L}(M)$ is finite" — semantic, non-trivial.
- "$\mathcal{L}(M)$ is regular" — semantic, non-trivial.
- "M has at most 5 states" — *not* semantic (depends on machine, not language).

Rice's Theorem: Statement and Proof

Theorem 12 (Rice's Theorem): Every non-trivial semantic property of TMs is undecidable.
That is, if P is non-trivial and semantic, then $\{\langle M\rangle \mid P(M)\}$ is undecidable.

Proof: Assume WLOG that $P(M_\emptyset) = \text{false}$ (where $\mathcal{L}(M_\emptyset) = \emptyset$). Since P is non-trivial, there exists some M_P with $P(M_P) = \text{true}$.

We reduce HALT_{TM} to P: given $\langle M, w\rangle$, construct M' that on input x:

1. Simulates M on w.
2. If M accepts w, simulates M_P on x.

Then: M halts on $w \to \mathcal{L}(M') = \mathcal{L}(M_P) \to P(M') = \text{true}$.
If M does not halt on $w \to \mathcal{L}(M') = \emptyset \to P(M') = \text{false}$. $\square$

Rice's Theorem: Consequences for FM

What Rice's theorem tells us:

- *"Does this program terminate?"* — undecidable (halting is semantic & non-trivial).
- *"Does this program satisfy its spec?"* — undecidable.
- *"Is this program equivalent to that one?"* — undecidable.

Every interesting program property is undecidable in general.

The FM response: We don't give up — we *approximate*:

- **Sound** over-approximation (abstract interpretation): may report false alarms, but never misses bugs.
- **Decidable fragments** (SMT theories): restrict to decidable sub-problems.
- **Programmer annotations** (Dafny): provide enough hints to make verification tractable.
- **Bounded checking** (SAT/BMC): verify up to bound k, not for all inputs.

Alternative Models of Computation

The Church–Turing Thesis

Beyond Turing machines, other models capture the same notion of “computability”:

Equivalent models:

- λ-calculus (Church, 1936)
- μ-recursive functions (Kleene)
- Post systems
- Register machines (RAM)
- ...and every general-purpose programming language

The Church–Turing Thesis: *Every effectively computable function is computable by a Turing machine.*

This is a *thesis*, not a theorem — it cannot be formally proved because “effectively computable” is an informal notion.

Historical note: Church and Turing independently arrived at equivalent definitions of computability in 1936. Church used λ-calculus; Turing used his machines. Both showed the Entscheidungsproblem is unsolvable.

λ-Calculus in a Nutshell

The λ-calculus is a minimal language with just three constructs:

Definition 32 (λ-Calculus Syntax):

$$M \Coloneqq x \mid (\lambda x.M) \mid (M\ \ N)$$

Variables, abstraction (function definition), and application (function call).

Example:

- *Identity:* $\lambda x.x$ — takes x, returns x.
- *Application:* $(\lambda x.x)\ y \rightsquigarrow y$ — β-reduction.
- *Church numeral* $\overline{2}$: $\lambda f.\lambda x.f(f(x))$ — "apply f twice".

Computation is *β-reduction*: $(\lambda x.M)\ N \rightsquigarrow M[x := N]$ (substitute N for x in M).

λ-Calculus in a Nutshell [2]

Key insight: Despite having no numbers, booleans, or loops, λ-calculus can encode *all* computable functions. This is the theoretical foundation of functional programming (Haskell, ML, Coq, Lean).

From Theory to Practice

Bridging Computability and Software Engineering

Theory says:

- Program correctness is undecidable (Rice).
- Halting is undecidable.
- FOL validity is undecidable.
- Full functional equivalence is undecidable.

Practice responds:

- Sound approximation (abstract interpretation).
- Decidable fragments (SMT theories).
- Programmer annotations (Dafny, ACSL, JML).
- Bounded verification (SAT, BMC, k-induction).

Example: Dafny's approach combines decidable theories (linear arithmetic, arrays, sets) with programmer-supplied loop invariants and pre/postconditions to make verification *tractable* for real programs.

The key message: Undecidability is not a dead end — it is a *design constraint*. Formal methods succeed by carefully choosing *what* to verify and *how much* automation to provide.

Looking Ahead

Week 3: SAT
- NP-completeness
- CDCL solvers
- SAT encodings

Week 6: SMT
- Decidable theories
- DPLL(T) architecture
- Theory combination

Weeks 9–12: Dafny
- Annotations
- Loop invariants
- Verified programs

Each step makes the *gap between theory and practice* narrower: from "undecidable in general" to "verified for this specific program".

Exercises

Exercises: Decidability and Computability

1. Show that the language $\{\langle M \rangle \mid M \text{ accepts at least one string}\}$ is recognizable but not decidable.
2. Using Rice's theorem, explain why each of the following is undecidable:
   - $\{\langle M \rangle \mid \mathcal{L}(M) = \Sigma^*\}$ (universality)
   - $\{\langle M \rangle \mid \mathcal{L}(M) \text{ is context-free}\}$
   - $\{\langle M \rangle \mid |\mathcal{L}(M)| = 42\}$
3. Construct a reduction from $\mathrm{HALT}_{\mathrm{TM}}$ to $\mathrm{TOTAL}_{\mathrm{TM}} = \{\langle M \rangle \mid M \text{ halts on all inputs}\}$ to prove $\mathrm{TOTAL}_{\mathrm{TM}}$ is undecidable.
4. Explain why Rice's theorem does *not* apply to the property "M has fewer than 10 states." What kind of property is this?
5. $\star$ The *Busy Beaver* function $\mathrm{BB}(n)$ = the maximum number of steps any halting TM with n states can make. Argue that BB grows faster than any computable function. *Hint:* if BB were computable, we could decide the halting problem.
6. $\star$ Consider λ-terms $I = \lambda x.x$ and $\Omega = (\lambda x.x\ x)(\lambda x.x\ x)$.
   Show that I has a normal form, but Ω does not (*i.e.*, β-reduction does not terminate on Ω).

Bibliography

[1] A. M. Turing, "On Computable Numbers, with an Application to the Entscheidungsproblem," *Proceedings of the London Mathematical Society*, no. 1, pp. 230–265, 1937, doi: 10.1112/plms/s2-42.1.230.