01204211 Discrete Mathematics
Lecture 12b: Undecidability (2)

Jittat Fakcharoenphol

September 18, 2025

Decision problems

- Given an integer x, is x odd?
- Given a string w, is w palindrome?
- Given a string w, is $w \in \{0^n1^n \mid n \geq 0\}$?
- Given a map, a starting position s, a destination t, and an integer k, does there exist a path from s to t with distance at most k?
- Given a program P and input string w, when running P with w as an input, does P terminate?

Decision problems and languages

For this problem:

Given an integer x, is x odd?

we can define a corresponding language

$$L_E = \{, \ldots, -6, -4, -2, 0, 2, 4, 6, \ldots\}.$$

To solve this problem, given x, we can ask if $x \in L_E$.

Deciders

We say that a python program P **decides** the language L if for any input string x, P when running with x as an input,

- P always terminates,
- P outputs **yes**, if $x \in L$, and
- P outputs **no**, if $x \notin L$.

Deciders

We say that a python program P **decides** the language L if for any input string x, P when running with x as an input,

- P always terminates,
- P outputs **yes**, if $x \in L$, and
- P outputs **no**, if $x \notin L$.

If we believe that anything that a computer can do can be written as a python program, and there is no python program that decides a language A, then we say that

A is undecidable.

Language SelfHalt and Halt

Let $\mathbb{P}$ be the set of all python programs. Let the language $\textsc{SelfHalt}$ be

$$\{P \in \mathbb{P} \mid \text{when running } P \text{ with } P \text{ as an input, } P \text{ terminates}\}$$

Or with a more concise notation:

$$\textsc{SelfHalt} = \{P \in \mathbb{P} \mid P(P) \text{ terminates}\}.$$

We also have another related language

$$\textsc{Halt} = \{(P, w) \mid P \text{ is a python program such that } P(w) \text{ terminates}\}$$

Lemma 1

SelfHalt *and* Halt *are undecidable.*

Lemma 2

There is no python program that decides Sᴇʟꜰ Hᴀʟᴛ.

Proof.

We prove by contradiction. Assume that there is a python program P that decides Sᴇʟꜰ Hᴀʟᴛ.

Lemma 2

There is no python program that decides SELFHALT.

Proof.

We prove by contradiction. Assume that there is a python program P that decides SELFHALT.

We describe a python program B that reads a string Q as an input as follows:

```
Program B
Input Q
1.     Load P as module Pmod
2.     if Pmod.main(Q) == 'yes':     # when Pmod outputs yes
3.        while True: pass           #   loop forever
4.     else:                         # when Pmod outputs no
5.        quit()                     #   halt
```

Given program Q as an input, B loops forever when

Lemma 2

There is no python program that decides SELFHALT.

Proof.

We prove by contradiction. Assume that there is a python program P that decides SELFHALT.

We describe a python program B that reads a string Q as an input as follows:

```
Program B
Input Q
1.     Load P as module Pmod
2.     if Pmod.main(Q) == 'yes':     # when Pmod outputs yes
3.        while True: pass           #   loop forever
4.     else:                         # when Pmod outputs no
5.        quit()                     #   halt
```

Given program Q as an input, B loops forever when

It terminates when

□

Proof.

We know that

- $B(Q)$ loops when $Q(Q)$ terminates, and
- $B(Q)$ terminates when $Q(Q)$ loops.

Does running B using B as an input terminate?

Proof.

We know that

- $B(Q)$ loops when $Q(Q)$ terminates, and
- $B(Q)$ terminates when $Q(Q)$ loops.

Does running B using B as an input terminate?
Let's try to plug in $Q = B$. We have

- $B(B)$ loops when

Proof.

We know that

- $B(Q)$ loops when $Q(Q)$ terminates, and
- $B(Q)$ terminates when $Q(Q)$ loops.

Does running B using B as an input terminate?
Let's try to plug in $Q = B$. We have

- $B(B)$ loops when $B(B)$ terminates,

Proof.

We know that

- $B(Q)$ loops when $Q(Q)$ terminates, and
- $B(Q)$ terminates when $Q(Q)$ loops.

Does running B using B as an input terminate?
Let's try to plug in $Q = B$. We have

- $B(B)$ loops when $B(B)$ terminates, and
- $B(B)$ terminates when

Proof.

We know that

- $B(Q)$ loops when $Q(Q)$ terminates, and
- $B(Q)$ terminates when $Q(Q)$ loops.

Does running B using B as an input terminate?
Let's try to plug in $Q = B$. We have

- $B(B)$ loops when $B(B)$ terminates, and
- $B(B)$ terminates when $B(B)$ loops.

Proof.

We know that

- $B(Q)$ loops when $Q(Q)$ terminates, and
- $B(Q)$ terminates when $Q(Q)$ loops.

Does running B using B as an input terminate?
Let's try to plug in $Q = B$. We have

- $B(B)$ loops when $B(B)$ terminates, and
- $B(B)$ terminates when $B(B)$ loops.

Since either $B(B)$ loops or terminates, and we cannot be in any of the cases, we obtain a contradiction.
Therefore, we conclude that program P does not exist. □

Reduction: proving undecidability of HALT

- We show that if HALT is decidable, then SELFHALT is also decidable.
- However, SELFHALT IS UNDECIDABLE.
- We conclude that HALT is also undecidable.

Reduction in picture

Let $\textsc{Accept} = \{(P, w) \mid P \in \mathbb{P}$ and $P(w)$ terminates with acceptance$\}$.

Lemma 3

$\textsc{Accept}$ *is undecidable.*

Proof.

We prove the lemma by contradiction. Assume that there is a python program Q that decides $\textsc{Accept}$. We construct a program C that decides $\textsc{Halt}$ as follows

```
Program C
Input P,w
1.   Replace every "print('no')" statement in P with "print('yes')"
1.   if Q(P,w) == 'yes':
2.      print('yes')
3.   else
4.      print('no')
```

Proof (cont.)

```
Program C
Input P,w
1.   Replace every "print('no')" statement in P with "print('yes')"
1.   if Q(P,w) == 'yes':
2.      print('yes')
3.   else
4.      print('no')
```

We have to make sure that our reduction is correct by considering two cases.
Case 1: when $P(w)$ halts.

Proof (cont.)

```
Program C
Input P,w
1.   Replace every "print('no')" statement in P with "print('yes')"
1.   if Q(P,w) == 'yes':
2.       print('yes')
3.   else
4.       print('no')
```

We have to make sure that our reduction is correct by considering two cases.
Case 1: when $P(w)$ halts.
Case 2: when $P(w)$ does not halt.

Proof (cont.)

```
Program C
Input P,w
1.   Replace every "print('no')" statement in P with "print('yes')"
1.   if Q(P,w) == 'yes':
2.      print('yes')
3.   else
4.      print('no')
```

We have to make sure that our reduction is correct by considering two cases.
Case 1: when $P(w)$ halts.
Case 2: when $P(w)$ does not halt.
Since in both cases, C answers correctly, we know that given program Q deciding $\textsc{Accept}$, we can construct a program C that decides $\textsc{Halt}$. However, we know that $\textsc{Halt}$ is undecidable; thus, we reach a contradiction. We conclude that $\textsc{Accept}$ is also undecidable. $\square$

Reduction from Halt to Accept in picture

How about $\textsc{Reject}$?

Let

$$\textsc{Reject} = \{(P, w) \mid P \in \mathbb{P} \text{ and } P \text{ rejects } w\}.$$

How about NotHalt?

Let

$$\textsc{NotHalt} = \{(P, w) \mid P \in \mathbb{P} \text{ and } P(w) \text{ does not terminate}\}.$$

Language of program P

For a python program P, let $L(P)$ be the set of all strings that P accepts, i.e.,

$$L(P) = \{w \in \Sigma^* \mid P(w) = yes\}.$$

Let

$$\textsc{All} = \{P \in \mathbb{P} \mid L(P) = \Sigma^*\}.$$

Lemma 4

ALL *is undecidable.*

Proof.

We prove by reduction from HALT. Assume that ALL is decidable, i.e., there is a python program Q that decides ALL. We construct program C that decides HALT as follows

```
Program C (input: P,w)
1. Construct another program R from P and w:
     | Program R (input: x)
     | 1.  Run program P on input w, suppressing any output from P
     | 2.  Accept x
2. if Q(R) == 'yes':
3.     return 'yes'
4. else: return 'no'
```

Proof (cont.)

```
Program C (input: P,w)
1. Construct another program R from P and w:
     | Program R (input: x)
     | 1.  Run program P on input w, suppressing any output from P
     | 2.  Accept x
2. if Q(R) == 'yes':
3.     return 'yes'
4. else: return 'no'
```

To ensure the correctness, we have to consider two cases.
Case 1: when $P(w)$ halts.

Proof (cont.)

```
Program C (input: P,w)
1. Construct another program R from P and w:
     | Program R (input: x)
     | 1.  Run program P on input w, suppressing any output from P
     | 2.  Accept x
2. if Q(R) == 'yes':
3.     return 'yes'
4. else: return 'no'
```

To ensure the correctness, we have to consider two cases.
Case 1: when $P(w)$ halts.
Case 2: when $P(w)$ does not halt.

Proof (cont.)

```
Program C (input: P,w)
1. Construct another program R from P and w:
     | Program R (input: x)
     | 1.  Run program P on input w, suppressing any output from P
     | 2.  Accept x
2. if Q(R) == 'yes':
3.     return 'yes'
4. else: return 'no'
```

To ensure the correctness, we have to consider two cases.
Case 1: when $P(w)$ halts.
Case 2: when $P(w)$ does not halt.
Since we can construct a program for HALT using a program that decides ALL, but HALT is undecidable; therefore, we conclude that ALL is undecidable. □

Empty

Let

$$\textsc{Empty} = \{P \in \mathbb{P} \mid L(P) = \emptyset\}.$$

Lemma 5

EMPTY *is undecidable.*

Proof.

We prove by reduction from HALT. Assume that EMPTY is decidable, i.e., there is a python program Q that decides EMPTY. We construct program C that decides HALT as follows

```
Program C (input: P,w)
1. Construct another program R from P and w:
     | Program R (input: x)
     | 1.  Run program P on input w, suppressing any output from P
     | 2.  Accept x
2. if Q(R) == 'yes':
3.     return 'no'
4. else: return 'yes'
```

Since we can construct a program for HALT using a program that decides EMPTY, but HALT is undecidable; therefore, we conclude that EMPTY is undecidable. □

Lemma 6

Let $\textsc{Eq} = \{(P_1, P_2) \mid P_1, P_2 \in \mathbb{P}$ *and* $L(P_1) = L(P_2)\}$. $\textsc{Eq}$ *is undecidable.*

Proof.

We prove by reduction from $\textsc{All}$. Assume that $\textsc{Eq}$ is decidable, i.e., there is a python program Q that decides $\textsc{Eq}$. We construct program C that decides $\textsc{All}$ as follows

```
Program C (input: P)
1. Construct another program R:
     | Program R (input: x)
     | 1.  Accept x
2. if Q(P,R) == 'yes':
3.     return 'yes'
4. else: return 'no'
```

Since we can construct a program for $\textsc{All}$ using a program that decides $\textsc{Eq}$, but $\textsc{All}$ is undecidable; therefore, we conclude that $\textsc{Eq}$ is undecidable. $\square$

Another proof for EQ

Proof.

We prove by reduction from EMPTY. Assume that EQ is decidable, i.e., there is a python program Q that decides EQ. We construct program C that decides EMPTY as follows

```
Program C (input: P)
1. Construct another program R:
     | Program R (input: x)
     | 1.  Reject x
2. if Q(P,R) == 'yes':
3.     return 'yes'
4. else: return 'no'
```

Since we can construct a program for EMPTY using a program that decides EQ, but EMPTY is undecidable; therefore, we conclude that EQ is undecidable. □

Lemma 7

Let $\textsc{Hello} = \{P \in \mathbb{P} \mid L(P) = \{\mathtt{hello}\}\}$. $\textsc{Hello}$ *is undecidable.*

INCORRECT PROOF.

We prove by reduction from $\textsc{Eq}$. Assume that $\textsc{Eq}$ is decidable, i.e., there is a python program Q that decides $\textsc{Eq}$. We construct program C that decides $\textsc{Hello}$ as follows

```
Program C (input: P)
1. Construct another program R:
     | Program R (input: x)
     | 1.  if x == 'hello':
     | 2.    print('yes')    # accept x
     | 3.  else
     | 4.    print('no')     # reject x
2. if Q(P,R) == 'yes':
3.     return 'yes'
4. else: return 'no'
```

$\square$

Lemma 8

Let $\textsc{Hello} = \{P \in \mathbb{P} \mid L(P) = \{\mathit{hello}\}\}$. $\textsc{Hello}$ *is undecidable.*

Proof

We prove by reduction from $\textsc{Halt}$. Assume that $\textsc{Hello}$ is decidable, i.e., there is a python program Q that decides $\textsc{Hello}$. We construct program C that decides $\textsc{Halt}$ as follows

```
Program C (input: P,w)
1. Construct another program R:
     | Program R (input: x)
     | 1.  if x != 'hello':
     | 2.    print('no')     # reject x
     | 3.  Replace any output statements from P
     | 4.  Run the modified P on w
     | 5.  print('yes')      # accept x
2. if Q(R) == 'yes':
3.     return 'yes'
4. else: return 'no'
```

Proof (cont.)

```
Program C (input: P,w)
1. Construct another program R:
     | Program R (input: x)
     | 1.  if x != 'hello':
     | 2.    print('no')     # reject x
     | 3.  Replace any output statements from P
     | 4.  Run the modified P on w
     | 5.  print('yes')      # accept x
2. if Q(R) == 'yes':
3.     return 'yes'
4. else: return 'no'
```

We consider two cases:
Case 1: $P(w)$ halts.

Proof (cont.)

```
Program C (input: P,w)
1. Construct another program R:
     | Program R (input: x)
     | 1.  if x != 'hello':
     | 2.    print('no')     # reject x
     | 3.  Replace any output statements from P
     | 4.  Run the modified P on w
     | 5.  print('yes')      # accept x
2. if Q(R) == 'yes':
3.     return 'yes'
4. else: return 'no'
```

We consider two cases:
Case 1: $P(w)$ halts.
Case 2: $P(w)$ does not halt.

Proof (cont.)

```
Program C (input: P,w)
1. Construct another program R:
     | Program R (input: x)
     | 1.  if x != 'hello':
     | 2.    print('no')     # reject x
     | 3.  Replace any output statements from P
     | 4.  Run the modified P on w
     | 5.  print('yes')      # accept x
2. if Q(R) == 'yes':
3.     return 'yes'
4. else: return 'no'
```

We consider two cases:
Case 1: $P(w)$ halts.
Case 2: $P(w)$ does not halt.
Since we can construct a program that decides $\textsc{Halt}$ given a program that decides $\textsc{Hello}$, but $\textsc{Halt}$ is undecidable. We conclude that $\textsc{Hello}$ is also undecidable. □

Python as computation

Do you believe in this assumption:
Anything that a computer can do can be written as a python program.

Turing machines

Anything that a computer can do can be carried out using Turing machines.

Turing machines

Anything that a computer can do can be carried out using Turing machines.

Any possible computation can be performed by Turing machines.

Turing Machines

Turing Machines

- Proposed by Alan Turing in 1936.

Turing Machines

- Proposed by Alan Turing in 1936.
- A finite automaton with an unlimited memory with unrestricted access.

Turing Machines

- Proposed by Alan Turing in 1936.
- A finite automaton with an unlimited memory with unrestricted access.
- Can perform any tasks that a computer can. (we'll see)

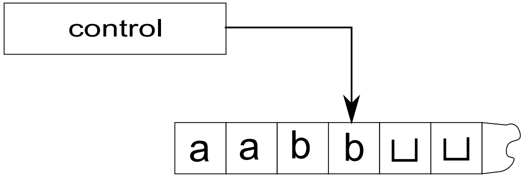
Turing Machines

- Proposed by Alan Turing in 1936.
- A finite automaton with an unlimited memory with unrestricted access.
- Can perform any tasks that a computer can. (we'll see)
- However, there are problems that TM can't solve. These problems are beyond the limit of computation.

Components

- An infinite **tape**.
- A tape head that can
    - **read and write** to the tape, and
    - **move** around the tape.

Schematic



How Turing machines work

- The tape initialy contains an input string.

How Turing machines work

- The tape initialy contains an input string.
- The rest of the tape is blank (denoted by $\sqcup$).

How Turing machines work

- The tape initialy contains an input string.
- The rest of the tape is blank (denoted by $\sqcup$).
- The machine reads a symbol from of the tape where its head is at.

How Turing machines work

- The tape initialy contains an input string.
- The rest of the tape is blank (denoted by $\sqcup$).
- The machine reads a symbol from of the tape where its head is at.
- It can write a symbol back and move left or right.

How Turing machines work

- The tape initialy contains an input string.
- The rest of the tape is blank (denoted by $\sqcup$).
- The machine reads a symbol from of the tape where its head is at.
- It can write a symbol back and move left or right.
- At the end of the computation, the machine outputs accept or reject, by entering accept state of reject state. (After changing, it halts.)

How Turing machines work

- The tape initialy contains an input string.
- The rest of the tape is blank (denoted by $\sqcup$).
- The machine reads a symbol from of the tape where its head is at.
- It can write a symbol back and move left or right.
- At the end of the computation, the machine outputs accept or reject, by entering accept state of reject state. (After changing, it halts.)
- It can go on forever (not entering any accept or reject states).

Examples

Formal definition of a Turing Machine

- Again, the important part is the definition of the transition function.

Formal definition of a Turing Machine

- Again, the important part is the definition of the transition function.
- The machine look at the tape symbol and consider its current state, then makes a move by writing some symbol on the tape and moving its head left or right.

Formal definition of a Turing Machine

- Again, the important part is the definition of the transition function.
- The machine look at the tape symbol and consider its current state, then makes a move by writing some symbol on the tape and moving its head left or right.
- Thus,
    - **Input:** current state and the symbol on the tape

Formal definition of a Turing Machine

- Again, the important part is the definition of the transition function.
- The machine look at the tape symbol and consider its current state, then makes a move by writing some symbol on the tape and moving its head left or right.
- Thus,
  - **Input:** current state and the symbol on the tape
  - **Output:** next state, a symbol to be written to the tape, and the new state.

Formal definition of a Turing Machine

- Again, the important part is the definition of the transition function.
- The machine look at the tape symbol and consider its current state, then makes a move by writing some symbol on the tape and moving its head left or right.
- Thus,
    - **Input:** current state and the symbol on the tape
    - **Output:** next state, a symbol to be written to the tape, and the new state.

Formal definition of a Turing Machine

- Again, the important part is the definition of the transition function.
- The machine look at the tape symbol and consider its current state, then makes a move by writing some symbol on the tape and moving its head left or right.
- Thus,
  - **Input:** current state and the symbol on the tape
  - **Output:** next state, a symbol to be written to the tape, and the new state.
- So, δ is in the form: $Q \times \Gamma \to Q \times \Gamma \times \{\texttt{LEFT}, \texttt{RIGHT}\}$

Formal definition of a Turing Machine

- Again, the important part is the definition of the transition function.
- The machine look at the tape symbol and consider its current state, then makes a move by writing some symbol on the tape and moving its head left or right.
- Thus,
  - **Input:** current state and the symbol on the tape
  - **Output:** next state, a symbol to be written to the tape, and the new state.
- So, δ is in the form: $Q \times \Gamma \rightarrow Q \times \Gamma \times \{\texttt{LEFT}, \texttt{RIGHT}\}$
- E.g., if $\delta(q, a) = (r, b, \texttt{LEFT})$, then if the machine is in state q and reads a, it will change its state to r, write b to the tape and move to the left.

Definition

Definition (Turing Machine)

A **Turing machine** is a 7-tuple, $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, where Q, Σ, Γ are finite sets and

1. Q is the set of states,
2. Σ is the input alphabet not containing the **blank symbol** $\sqcup$,
3. Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subset \Gamma$,
4. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\mathtt{LEFT}, \mathtt{RIGHT}\}$ is the transition function,
5. $q_0 \in Q$ is the start state,
6. $q_{accept} \in Q$ is the accept state, and
7. $q_{reject} \in Q$ is the reject state, where $q_{accept} \neq q_{reject}$.

Variants of Turing Machines

- There are many alternative definitions of TM's.

Variants of Turing Machines

- There are many alternative definitions of TM's.
- They are called variants of TM's.

Variants of Turing Machines

- There are many alternative definitions of TM's.
- They are called variants of TM's.
- They all have the same power. This demonstrates the robustness in the definition of TM's.

Variants of Turing Machines

- There are many alternative definitions of TM's.
- They are called variants of TM's.
- They all have the same power. This demonstrates the robustness in the definition of TM's. Also, this is an evidence that TM's "capture" the idea of computation (because whatever computing machine we can think of they are all equivalent to TM's).

TM with “stay put”

- Let’s start with an easy variant. Suppose we allow additional head movement: “stay put (S)”.

TM with “stay put”

- Let’s start with an easy variant. Suppose we allow additional head movement: “stay put (S)”.
- The transition function will be of the form

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{\texttt{LEFT}, \texttt{RIGHT}, \mathrm{S}\}.$$

- Does this give TM’s more power?

TM with “stay put”

- Let’s start with an easy variant. Suppose we allow additional head movement: “stay put (S)”.
- The transition function will be of the form

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{\texttt{LEFT}, \texttt{RIGHT}, \mathrm{S}\}.$$

- Does this give TM’s more power?
- Not really. We can convert a TM with “stay put” to a standard TM as follows.

TM with "stay put"

- Let's start with an easy variant. Suppose we allow additional head movement: "stay put (S)".
- The transition function will be of the form

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{\mathtt{LEFT}, \mathtt{RIGHT}, \mathrm{S}\}.$$

- Does this give TM's more power?
- Not really. We can convert a TM with "stay put" to a standard TM as follows.
    - For any "stay put" transition, we replace with two transitions: "right" and "left".

Multitape Turing Machines

- A **multitape Turing machines** has many tapes.

Multitape Turing Machines

- A **multitape Turing machines** has many tapes.
- For each tape, the machine has a head for reading and writing it.

Multitape Turing Machines

- A **multitape Turing machines** has many tapes.
- For each tape, the machine has a head for reading and writing it.
- The input appears on tape 1; all other tapes contain blanks.

Multitape Turing Machines

- A **multitape Turing machines** has many tapes.
- For each tape, the machine has a head for reading and writing it.
- The input appears on tape 1; all other tapes contain blanks.
- Let k be the number of tapes. The transition function can be defined as

$$\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{\mathtt{LEFT}, \mathtt{RIGHT}, \mathtt{STAY}\}^k.$$

Multitape Turing Machines

- A **multitape Turing machines** has many tapes.
- For each tape, the machine has a head for reading and writing it.
- The input appears on tape 1; all other tapes contain blanks.
- Let k be the number of tapes. The transition function can be defined as

$$\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{\texttt{LEFT}, \texttt{RIGHT}, \texttt{STAY}\}^k.$$

- E.g., if $\delta(q_i, a_1, \ldots, a_k) = (q_j, b_1, \ldots, b_k, \texttt{LEFT}, \texttt{RIGHT}, \ldots, \texttt{LEFT})$ then if the machine is at state q_i and each head on tape i reads symbol a_i, it'll write b_i on each tape i, change state to q_j and move each head accordingly.

Nondeterministic Turing Machines

- A nondeterministic Turing machine can make “nondeterministic” move.

Nondeterministic Turing Machines

- A nondeterministic Turing machine can make “nondeterministic” move.
- As expected, its transition function has the form

$$\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{\texttt{LEFT}, \texttt{RIGHT}\}).$$

Nondeterministic Turing Machines

- A nondeterministic Turing machine can make "nondeterministic" move.
- As expected, its transition function has the form

$$\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{\texttt{LEFT}, \texttt{RIGHT}\}).$$

- Again, we view the computation of a nondeterministic Turing machine as a tree, where each branching corresponds to the place where the TM can make different moves.

Nondeterministic Turing Machines

- A nondeterministic Turing machine can make “nondeterministic” move.
- As expected, its transition function has the form

$$\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{\mathtt{LEFT}, \mathtt{RIGHT}\}).$$

- Again, we view the computation of a nondeterministic Turing machine as a tree, where each branching corresponds to the place where the TM can make different moves.
- Can nondeterminism help?

Equivalence in Power

- **Should I write programs in C or Pascal?**

Equivalence in Power

- **Should I write programs in C or Pascal?**
- **Should I write programs in Python or Prolog?**

Equivalence in Power

- **Should I write programs in C or Pascal?**
- **Should I write programs in Python or Prolog?**
- **Should I write programs in Ruby or LISP?**

They are all the same, in terms of computability

Since you can write a C interpreter in Pascal and Pascal interpreter in C,

They are all the same, in terms of computability

Since you can write a C interpreter in Pascal and Pascal interpreter in C, what you can do in C, you can do in Pascal.

Turing machine

If you believe that Turing machines are ultimate model of computing, all those programming languages are equivalent because they all can simulate Turing machines (and they runs on Turing machines).

Two sides of a coin

- Computers are powerful

Two sides of a coin

- Computers are powerful (???)

Two sides of a coin

- Computers are powerful (???)
- How powerful?

Two sides of a coin

- Computers are powerful (???)
- How powerful?
- To understand that, we want to see samples of tasks that computers can do, and samples of tasks that they can't do.

Two sides of a coin

- Computers are powerful (???)
- How powerful?
- To understand that, we want to see samples of tasks that computers can do, and samples of tasks that they can't do.
- It's one story to show that computers can do something.

Two sides of a coin

- Computers are powerful (???)
- How powerful?
- To understand that, we want to see samples of tasks that computers can do, and samples of tasks that they can't do.
- It's one story to show that computers can do something. It's another to show that computers can't do something.
  - Maybe there's a limitation with "this" computer, but "other" computers might be able to do that thing.
  - We want to be able to say that **for all** computers.

Two sides of a coin

- Computers are powerful (???)
- How powerful?
- To understand that, we want to see samples of tasks that computers can do, and samples of tasks that they can't do.
- It's one story to show that computers can do something. It's another to show that computers can't do something.
  - Maybe there's a limitation with "this" computer, but "other" computers might be able to do that thing.
  - We want to be able to say that **for all** computers. In fact, **for any "thinkable"** computers.

What is a computer?

What is a computer?
Something that computes?

What is a computer?
Something that computes?
What is computation?

What is a computer?
Something that computes?
What is computation?
An act of following some instructions?

What is a computer?
Something that computes?
What is computation?
An act of following some instructions?
An act of following some algorithm?

What is a computer?
Something that computes?
What is computation?
An act of following some instructions?
An act of following some algorithm?
What is an algorithm?

Hilbert's problems

Mathematician David Hilbert asked:
"Find a process according to which it can be determined by a finite number of operations if a given polynomial has intergral root"

To say NO

We need an argument (a mathematical proof) that covers all possible “processes” or all “computations”.

Possible definitions

- Church's λ-calculus
- Turing's machines

Possible definitions

- Church's λ-calculus
- Turing's machines

They both turned out to be **equivalent**.

Church-Turing thesis

Turing machine algorithms $=$ intuitive notion of algorithms

Final answer to Hilbert

No, there doesn't exist any algorithm for determining if a polynomial has integral root.