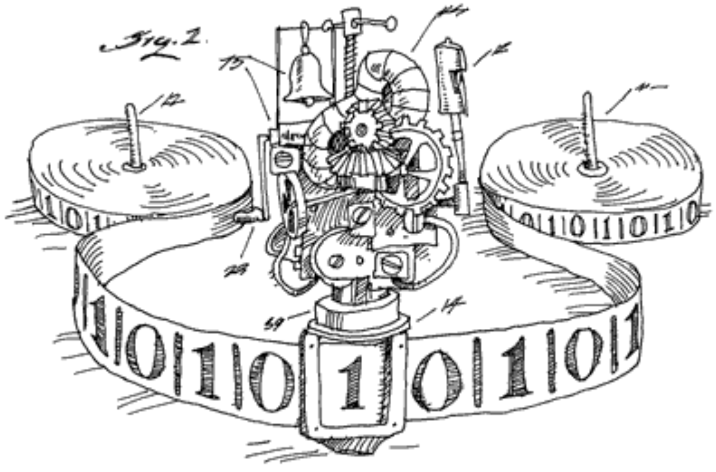
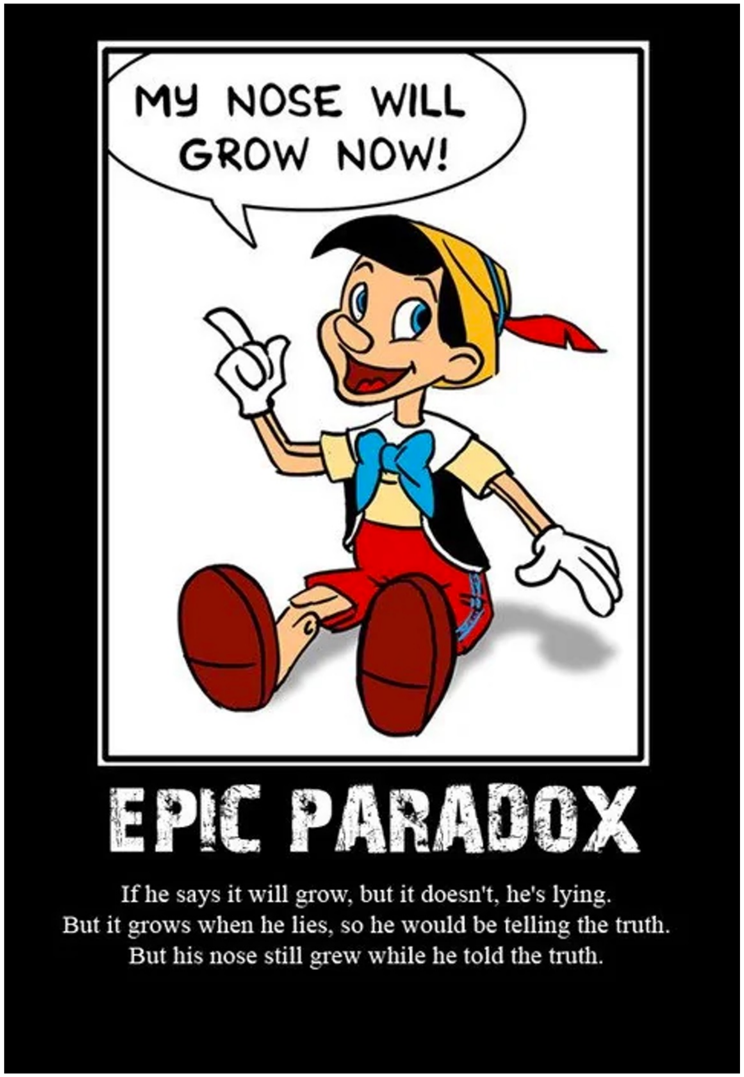
EECS 376: Foundations of Computer Science

Lecture 11 - More Undecidable Problems, Turing Reductions



Explicit Undecidable Problems



Today's Agenda

- Recap:
  - (un)countability
  - "Almost all" problems are undecidable

- An explicit undecidable language from the Barber Paradox
- Two more natural undecidable problems:
  - Accepting problem L_{ACC}
  - Halting problem L_{HALT}
- Turing reductions/reducibility

Recap

Review: Countability

- A set S is ***countable*** if
  - there is an injective from S to $\mathbb{N}$, or equivalently
  - we can list elements in S so each element appear *at some finite position* in the list.

- Which set is countable?
  - rational numbers
  - real numbers
  - all *finite* binary strings
  - all *infinite* binary sequences
  - all Turing machines
  - all decidable languages
  - all languages

Review: Countability

- A set S is ***countable*** if
  - there is an injective from S to $\mathbb{N}$, or equivalently
  - we can list elements in S so each element appear *at some finite position* in the list.

- Which set is countable?
  - rational numbers
  - real numbers
  - all *finite* binary strings
  - all *infinite* binary sequences
  - all Turing machines
  - all decidable languages
  - all languages

There is an Undecidable Language

- Set of all TMs (programs) is countable:
  - Can list them by their "source code" (descriptions as strings)
  - $\langle M_1 \rangle, \langle M_2 \rangle, \langle M_3 \rangle, \ldots \in \{0,1\}^*$
- Set of all decidable languages is countable:
  - Each TM decides at most one language
- Set of all languages (infinite binary strings) is uncountable
- So, there is an undecidable language!
- Actually, "almost all" languages are undecidable.
  (Similarly, "almost all" real numbers are irrational…)

Summary

The ‘good’: We showed that there is an undecidable language

The ‘bad’: This language is “non-constructive” and “unnatural”

Q: Can we say anything about **“natural” problems** that we care about, or are “useful”?

An Explicit
Undecidable Problem

Barber Paradox

- The **barber paradox** is a puzzle derived from Russell's paradox.
- Bertrand Russell used it to illustrate the paradox, though he attributes it to an unnamed person who suggested it to him.
- The puzzle shows that a plausible scenario is logically impossible.
- Specifically, it describes a barber who is defined such that he both shaves himself and does not shave himself, which implies that no such barber exists.

Russell’s Paradox

- In mathematical logic, **Russell’s Paradox** (also known as **Russell’s Antinomy**) is a set-theoretic paradox published by the British philosopher and mathematician Bertrand Russell in 1901.
- **Russell's paradox** shows that every set theory that contains an unrestricted comprehension principle leads to contradictions.
- According to the unrestricted comprehension principle, for any sufficiently well-defined property, there is a set of all and only objects with that property.

The Barber Paradox

- **The Paradox:**
  - Barber B *cuts* the hair of exactly all those people who do not *cut* their own hair.

    Question: Does B cut his own hair?
  - Pick a person P:
    - P cut the hair of $P \Rightarrow$ B does not cut the hair of P.
    - P does not cut the hair of $P \Rightarrow$ B cut the hair of P.
  - Substituting $P = \mathrm{B}$:
    - B cut the hair of B ⇒ B does not cut the hair of B
    - B does not cut the hair of B ⇒ B cut the hair of B
      - A Contradiction! But contradiction to what?

  - Conclusion: this Barber cannot exist

Barber Paradox is actually Diagonalization in Disguise

- **The Paradox**
  - Barber B *cuts* the hair of exactly all those people who do not *cut* their own hair.
- Consider the CUTTER-CUTTEE Matrix:

CUTTEE

CUTTER

	Chico	Harpo	Groucho	Gummo	Zeppo	Barber
Chico	Y					
Harpo		N				
Groucho			N			
Gummo				Y		
Zeppo					N	
Barber	N	Y	Y	N	Y	Y **or** N**?**

Source Code as Input

- Since **a program’s source code (or a TM) is just a string,**
- it can be passed as input to another program — or even to the program itself!
- **Example:**

```
bool M1(string s) {
    return 0;
}
```

$\langle M_1 \rangle =$ *"bool M1(string s) {\n return 0;\n}\n“*

Q: What does $M_1(\langle M_1 \rangle)$ return?

Using the Barber Paradox to Construct an Undecidable Problem

"**Barber B** *cuts* the hair of exactly all **those people** who do not *cut* their own hair."

Let's consider a computational analogy, where:

- **barber, people** = **TM(s)**
- hair = description of TM
- *cut* = *accept*

"***TM M***"

"***TM*** M_{BARBER} *accepts* as input the description of exactly all ***TMs*** that do not *accept* as input their own description."

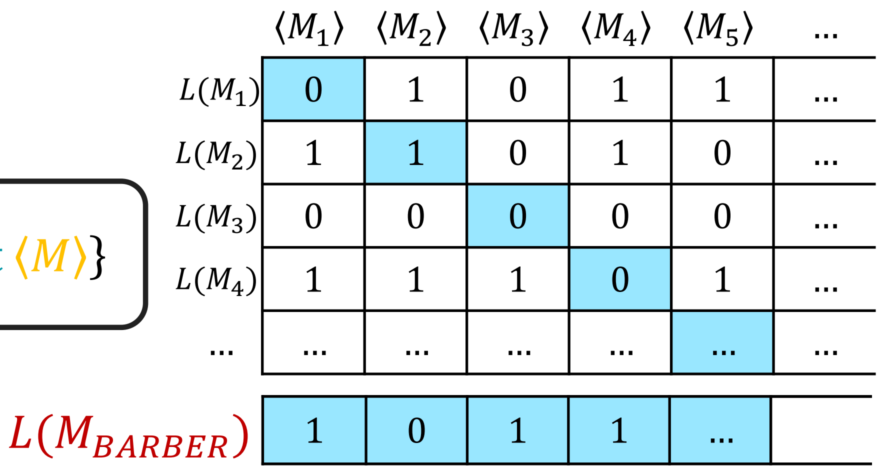
$$L(M_{BARBER}) = \{\langle M \rangle : M \text{ does not accept } \langle M \rangle\}$$

Is $L(M_{BARBER})$ decidable? Does TM M_{BARBER} exist?

Barber performs Diagonalization

- List of all decidable languages $L(M_1), L(M_2), L(M_3), \ldots$
- Define a table $T(i,j) = \begin{cases} 1 & \text{if } M_i \text{ accepts } \langle M_j \rangle \\ 0 & \text{otherwise} \end{cases}$

$$L(M_{BARBER}) = \{\langle M \rangle : M \text{ does not accept } \langle M \rangle\}$$



	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	…
$L(M_1)$	0	1	0	1	1	…
$L(M_2)$	1	1	0	1	0	…
$L(M_3)$	0	0	0	0	0	…
$L(M_4)$	1	1	1	0	1	…
…	…	…	…	…	…	…

$L(M_{BARBER})$	1	0	1	1	…	

- $L(M_{BARBER})$ flips the diagonal!
- $L(M_{BARBER})$ is not on the list.
- So $L(M_{BARBER})$ is undeciable

An ***Explicit*** Undecidable Language

- Since no program has
$$L_{\text{BARBER}} = \{\langle M\rangle : M \text{ does not accept } \langle M\rangle\}$$
as its language, L_{BARBER} is ***undecidable***.

- This is our first example of an explicit undecidable language!

- **Q:** Why do we care about an explicit undecidable language?
- **A:** We can use it to show that ***other languages*** are also undecidable.

A Natural Undecidable Problem: The Acceptance Problem

The Acceptance Problem

Input: Turing Machine **M** and a string **x**

Output: Does **M** accept **x**?

```
bool M(string $x$):
$n \leftarrow 100$
while ($n > 1$):  $n \leftarrow n - 1$
return T
```

Example: Does M accept x=376 ?

Language: $L_{ACC} = \{(\langle M \rangle, x) : M \text{ accepts } x\}$

Q: Why not just run M(x)?

Attempt 1: Interpreter

- An ***interpreter*** is a program that takes another program as input and simulates its behavior.
  - e.g. the Python interpreter

- Specifically, an interpreter **U** takes two inputs: (1) source code **⟨M⟩**, and (2) string **x**.

- **U** simulates the execution of **M** on input **x**:
  - **M** accepts **x** ⇒ **U** accepts (**⟨M⟩**, **x**)
  - **M** rejects **x** ⇒ **U** rejects (**⟨M⟩**, **x**)
  - **M** loops on **x** ⇒ **U** loops on (**⟨M⟩**, **x**)

- This is called the ***Universal Turing Machine*** (and it does exist)

Does Interpreter Decide L_{ACC}?

- $U(\langle M \rangle, x)$ simulates the execution of M on x:
  - M accepts $x \Longrightarrow U$ accepts $(\langle M \rangle, x)$
  - M rejects $x \Longrightarrow U$ rejects $(\langle M \rangle, x)$
  - M loops on $x \Longrightarrow U$ loops on $(\langle M \rangle, x)$

In both cases $(\langle M \rangle, x) \notin L(U)$

- The language of U is: $L(U) \equiv L_{\text{ACC}} = \{(\langle M \rangle, x) : M \text{ accepts } x\}$

- However, U is *not a decider* for L_{ACC} . Why?
- U loops on some inputs.
- L_{ACC} is actually undecidable. We will show this using **reduction**

Reduction

Plan for showing that L_{ACC} is Undecidable

Could try to come up with a diagonalization proof …
… but let's not reinvent the wheel!

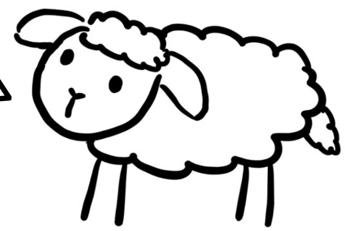
KEY IDEA: Once we have one undecidable language, we can use it to show that other languages are also undecidable!

HOW? Show that if L_{ACC} were decidable,
this would let us **decide some undecidable language!**

This is called a **reduction.**

(Specifically, a Turing-reduction)

The idea of a reduction is one of the most central ideas in computer science!



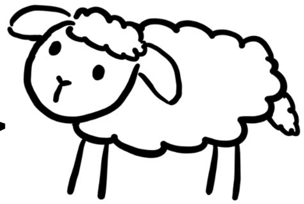
Turing Reduction from **A** to **B** (denoted **A** $\le_T$ **B**):

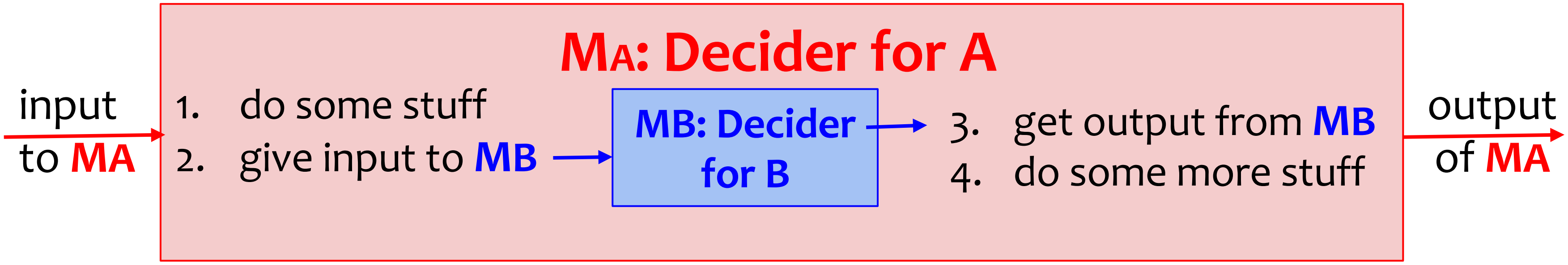
"We can use a **black-box decider** for **B**
as a subroutine to decide **A**."

What it implies:

1. If **B** is decidable then **A** is decidable.
2. Contrapositive: If **A** is undecidable then **B** is undecidable.

You can even invoke M_B
multiple times inside of M_A
if you want





M_A: Decider for A

input
to **MA**

1. do some stuff
2. give input to **MB**

MB: Decider for B

3. get output from **MB**
4. do some more stuff

output
of **MA**

"Problem **B** is at least as hard as Problem **A**"

Reducing L_{BARBER} to L_{ACC} :

$$L_{\text{BARBER}} \leq_T L_{\text{ACC}}$$

Reduction from L_{BARBER} to L_{ACC} (i.e. $L_{BARBER} \leq_T L_{ACC}$)

We need to implement:

M_{BARBER} takes one input: **⟨M⟩**

M does not accept **⟨M⟩** ⇒ **M_{BARBER}** accepts

M accepts **⟨M⟩** ⇒ **M_{BARBER}** rejects

Suppose we have:

M_{ACC} takes two inputs: **⟨M⟩**, **x**

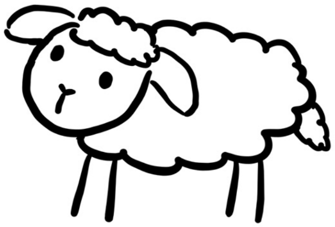
M accepts **x** ⇒ **M_{ACC}** accepts

M does not accepts **x** ⇒ **M_{ACC}** rejects

Task: specify the pseudocode

```
$M_{BARBER}$(⟨M⟩):
```

We are allowed to use M_{ACC}(**⟨M⟩**, **x**) as a subroutine, with the inputs of our choice



Reduction from L_{BARBER} to L_{ACC} (i.e. $L_{\text{BARBER}} \leq_T L_{\text{ACC}}$)

We need to implement:

M_{BARBER} takes one input: **⟨M⟩**

M does not accept **⟨M⟩** ⇒ **M_{BARBER}** accepts

M accepts **⟨M⟩** ⇒ **M_{BARBER}** rejects

Suppose we have:

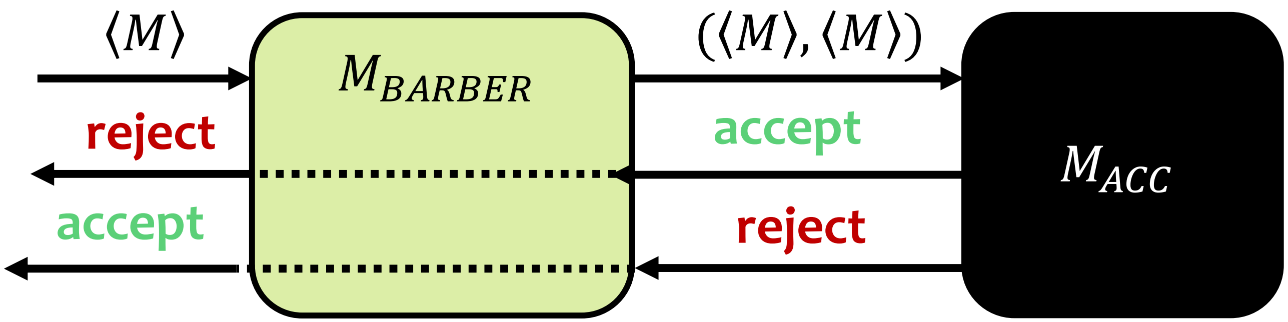
M_{ACC} takes two inputs: **⟨M⟩**, **x**

M accepts **x** ⇒ **M_{ACC}** accepts

M does not accepts **x** ⇒ **M_{ACC}** rejects

Task: specify the pseudocode

```
M_BARBER(⟨M⟩):
    Run M_ACC on (⟨M⟩, ⟨M⟩).
    If M_ACC accepts, then reject.
    If M_ACC rejects, then accept.
```



Reduction from L_{BARBER} to L_{ACC} (i.e. $L_{BARBER} \leq_T L_{ACC}$)

We need to implement:

M_{BARBER} takes one input: **⟨M⟩**

M does not accept **⟨M⟩** ⇒ **M_{BARBER}** accepts

M accepts **⟨M⟩** ⇒ **M_{BARBER}** rejects

Suppose we have:

M_{ACC} takes two inputs: **⟨M⟩**, **x**

M accepts **x** ⇒ **M_{ACC}** accepts

M does not accepts **x** ⇒ **M_{ACC}** rejects

Task: specify the pseudocode

```
M_BARBER(⟨M⟩):
    Run M_ACC on (⟨M⟩, ⟨M⟩).
    If M_ACC accepts, then reject.
    If M_ACC rejects, then accept.
```

Analysis:

- M_{BARBER} halts on any input, because M_{ACC} does.
- $\langle M \rangle \in L_{BARBER} \Leftrightarrow$
  - M does not accept $\langle M \rangle \Leftrightarrow$
  - M_{ACC} rejects $(\langle M \rangle, \langle M \rangle) \Leftrightarrow$
  - M_{BARBER} accepts $\langle M \rangle$.

Conclude: L_{ACC} is undecidable

- We showed $L_{\text{BARBER}} \leq_T L_{\text{ACC}}$

- But L_{BARBER} is undecidable
- So L_{ACC} is undecidable

Another Natural Undecidable Problem:
The Halting Problem

The Halting Problem

Input: Turing Machine **M** and a string **x**

Output: Does **M** halt when given input **x**?

Language: $L_{HALT} = \{(\langle M\rangle, x) : M \text{ halts on input } x\}$

We will show that $L_{ACC} \leq_T L_{HALT}$. So L_{HALT} is undecidable.

Q: Again, what is wrong with just running M(x)?

Reduction from L_{ACC} to L_{HALT} (i.e. $L_{ACC} \le_T L_{HALT}$)

We need to implement:

M_{ACC} takes two inputs: **⟨M⟩**, **x**

M accepts **x** ⇒ **M_{ACC}** accepts

M loops or rejects **x** ⇒ **M_{ACC}** rejects

Suppose we have:

M_{HALT} takes two inputs: **⟨M⟩**, **x**

M accepts or rejects **x** ⇒ **M_{HALT}** accepts

M loops on input **x** ⇒ **M_{HALT}** rejects

We need to specify the pseudocode:

```
    $M_{ACC}$(⟨M⟩, x):
```

Reduction from L_{ACC} to L_{HALT} (i.e. $L_{ACC} \leq_T L_{HALT}$)

We need to implement:

M_{ACC} takes two inputs: **⟨M⟩, x**

M accepts **x** ⇒ **M_{ACC}** accepts

M loops or rejects **x** ⇒ **M_{ACC}** rejects

Suppose we have:

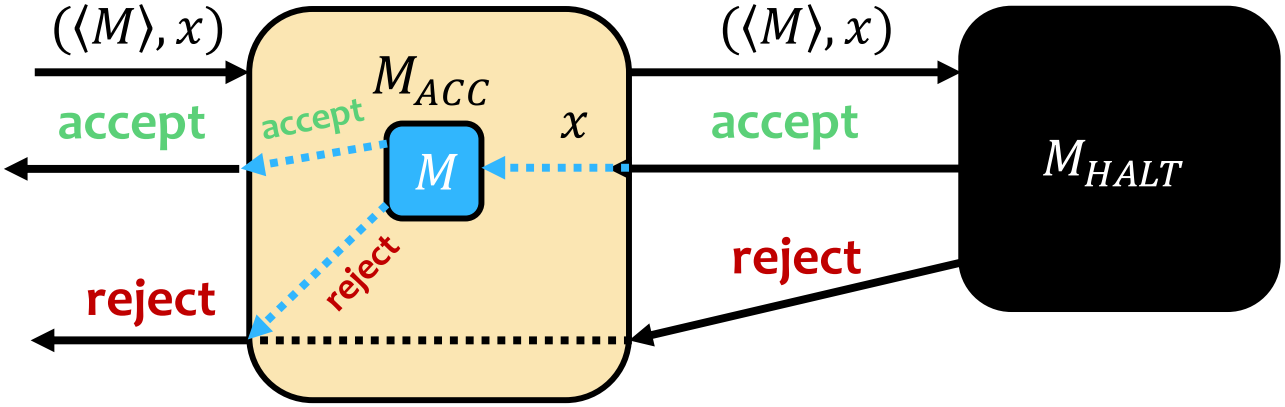
M_{HALT} takes two inputs: **⟨M⟩, x**

M accepts or rejects **x** ⇒ **M_{HALT}** accepts

M loops on input **x** ⇒ **M_{HALT}** rejects

Task: specify the pseudocode

```
M_ACC(⟨M⟩, x):
    Run $M_{HALT}$ on $(\langle M \rangle, x)$
    If $M_{HALT}$ rejects, reject          ($M$ loops on $x$, so $M$ does not accept $x$)
    If $M_{HALT}$ accepts, run $M$ on $x$  (this simulation will terminate!)
        If $M$ accepts, accept
        If $M$ rejects, reject
```



Reduction from L_{ACC} to L_{HALT} (i.e. $L_{ACC} \leq_T L_{HALT}$)

We need to implement:

M_{ACC} takes two inputs: **⟨M⟩, x**

M accepts **x** ⇒ **M_{ACC}** accepts

M loops or rejects **x** ⇒ **M_{ACC}** rejects

Suppose we have:

M_{HALT} takes two inputs: **⟨M⟩, x**

M accepts or rejects **x** ⇒ **M_{HALT}** accepts

M loops on input **x** ⇒ **M_{HALT}** rejects

Task: specify the pseudocode

```
M_ACC(⟨M⟩, x):
    Run $M_{HALT}$ on $(\langle M \rangle, x)$
    If $M_{HALT}$ rejects, reject                 ($M$ loops on $x$, so $M$ does not accept $x$)
    If $M_{HALT}$ accepts, run $M$ on $x$         (this simulation will terminate!)
        If $M$ accepts, accept
        If $M$ rejects, reject
```

Analysis: M_{ACC} halts on any input (why?). Moreover:

M accepts $x \Longrightarrow M_{HALT}$ accepts $(\langle M \rangle, x) \Longrightarrow M_{ACC}$ accepts $(\langle M \rangle, x)$

M rejects $x \Longrightarrow M_{HALT}$ accepts $(\langle M \rangle, x) \Longrightarrow M_{ACC}$ rejects $(\langle M \rangle, x)$

M loops on $x \Longrightarrow M_{HALT}$ rejects $(\langle M \rangle, x) \Longrightarrow M_{ACC}$ rejects $(\langle M \rangle, x)$

Conclude: L_{HALT} is undecidable

- We showed $L_{ACC} \leq_T L_{HALT}$

- But L_{ACC} is undecidable
- So L_{HALT} is undecidable

Wrap Up

Two ways to show undecidability

- **Two Options:**
  1. Direct proof using **diagonalization arguments** (L_{BARBER})
  2. Indirect proof using **reductions** (L_{HALT} and L_{ACC})

- Suppose $A \leq_T B$.
  - If B is decidable, then A is decidable.
  - If A is undecidable, then B is undecidable

- **We have shown today:** $L_{BARBER} \leq_T L_{ACC} \leq L_{HALT}$.
  - So L_{ACC} and L_{HALT} are undecidable.

Exercises

- **Question 1:** Is it true that $L_{\mathrm{HALT}} \leq_T L_{\mathrm{ACC}}$?
- **Question 2:** Suppose $A \leq_T B$. Must $B \leq_T A$?

Optional

- **Q.** What could you do with a program that solved the halting problem?
- **A.** Solve just about any open mathematical problem!
- **Goldbach's Conjecture:** every even number is the sum of two primes. E.g. 20=7+13, 22=3+19, 24=5+19, 26=7+19, 28=11+17, etc.

```
Goldbach()
    For every even $x = 2,4,6,\ldots$
        bool = FALSE;
            For $y$ from 2 to $x - 2$
                If(y and x-y are both prime) bool = TRUE
        If(bool==FALSE) Return().
```

- Does Goldbach *loop* or eventually ***halt***?