UMB CS 622
Undecidability
Monday, April 22, 2024
Announcements

• HW 9 out
  • due Wednesday 4/24 12pm noon
  • Problems 3 and 4 moved to HW 10
Recap: Decidability of Regular and CFLs

- $A_{DFA} = \{ \langle B, w \rangle | B \text{ is a DFA that accepts input string } w \}$  Decidable
- $A_{NFA} = \{ \langle B, w \rangle | B \text{ is an NFA that accepts input string } w \}$  Decidable
- $A_{REX} = \{ \langle R, w \rangle | R \text{ is a regular expression that generates string } w \}$  Decidable
- $E_{DFA} = \{ \langle A \rangle | A \text{ is a DFA and } L(A) = \emptyset \}$  Decidable
- $E_{DFA} = \{ \langle A, B \rangle | A \text{ and } B \text{ are DFAs and } L(A) = L(B) \}$  Decidable
- $A_{CFG} = \{ \langle G, w \rangle | G \text{ is a CFG that generates string } w \}$  Decidable
- $E_{CFG} = \{ \langle G \rangle | G \text{ is a CFG and } L(G) = \emptyset \}$  Decidable
- $E_{CFG} = \{ \langle G, H \rangle | G \text{ and } H \text{ are CFGs and } L(G) = L(H) \}$  Decidable? 
- $A_{TM} = \{ \langle M, w \rangle | M \text{ is a TM and } M \text{ accepts } w \}$  Decidable?
Thm: \( A_{TM} \) is Turing-recognizable

\[ A_{TM} = \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \} \]

\( U \) = “On input \( \langle M, w \rangle \), where \( M \) is a TM and \( w \) is a string:

1. Simulate \( M \) on input \( w \).
2. If \( M \) ever enters its accept state, accept; if \( M \) ever enters its reject state, reject.”

\( U \) = Implements TM computation steps

- i.e., “The Universal Turing Machine”
- “Program” simulating other programs (interpreter)

Problem (Step 1): \( U \) loops when \( M \) loops

So it’s not a decider. Is it recognizer?
**Thm:** $A_{TM}$ is Turing-recognizable

$$A_{TM} = \{ \langle M, w \rangle | M \text{ is a TM and } M \text{ accepts } w \}$$

$U = \text{“On input } \langle M, w \rangle, \text{ where } M \text{ is a TM and } w \text{ is a string:}$$

1. Simulate $M$ on input $w$.
2. If $M$ ever enters its accept state, *accept*; if $M$ ever enters its reject state, *reject.*"

<table>
<thead>
<tr>
<th>Example Str</th>
<th>$M$ on input $w$?</th>
<th>$U$?</th>
<th>In $A_{TM}$ lang?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$&lt;M_1, 01#01&gt;$</td>
<td>Accept</td>
<td>Accept</td>
<td>Yes</td>
</tr>
<tr>
<td>$&lt;M_1, 00#11&gt;$</td>
<td>Reject</td>
<td>Reject</td>
<td>No</td>
</tr>
<tr>
<td>$&lt;M_{loop} *&gt;$</td>
<td>Loop!</td>
<td>Loop!</td>
<td>No</td>
</tr>
</tbody>
</table>

Let:
- $M_1 = \text{“}w\#w\text{” lang decider}$
- $M_{loop} = \text{looping TM}$

Columns must match!

Is this right? Yes!
**Thm:** $A_{TM}$ is undecidable

$$A_{TM} = \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \}$$

• ???
Prove: Spider-Man does not exist

In general, proving something **not true** is different (and harder) than proving it **true**

In some cases, it’s possible, but typically requires **new proof techniques**!

**Example (Regular Languages)**

Prove a language is **regular**:
- Create a DFA

Prove a language is **not regular**:
- Proof by contradiction using **Pumping Lemma**
Thm: $A_{TM}$ is undecidable

$$A_{TM} = \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \}$$

Example (decidable languages)

Prove a language is decidable:
- Create a decider TM (with termination argument)

Prove a language is not decidable:
- ????
Kinds of Functions (a fn maps **Domain** → **Range**)

- **Injective**, a.k.a., “one-to-one”
  - Every element in **Domain** has a unique mapping
  - How to remember:
    - Entire **Domain** is mapped “in” to the **Range**

- **Surjective**, a.k.a., “onto”
  - Every element in **Range** is mapped to
  - How to remember:
    - “Sur” = “over” (eg, survey); **Domain** is mapped “over” the **Range**

- **Bijective**, a.k.a., “correspondence” or “one-to-one correspondence”
  - Is both injective and surjective
  - Unique pairing of every element in **Domain** and **Range**
Countability

• A set is “countable” if it is:
  • Finite
  • Or, there exists a bijection between the set and the natural numbers
    • In this case, the set has the same size as the set of natural numbers
    • This is called “countably infinite”
Exercise: Which set is larger?

• The set of:
  • Natural numbers, or
  • Even numbers?

• They are the same size! Both are countably infinite
  • Proof: Bijection:

<table>
<thead>
<tr>
<th>n</th>
<th>$f(n) = 2n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Every natural number maps to a unique even number, and vice versa
Exercise: Which set is larger?

- The set of:
  - Natural numbers $\mathbb{N}$, or
  - Positive rational numbers? $\mathbb{Q} = \left\{ \frac{m}{n} \mid m, n \in \mathbb{N} \right\}$
- They are the same size! Both are countably infinite

A possible mapping of Natural numbers to Positive rationals?

So these don’t get mapped to: (not a bijection)

But, each row is infinite

Positive rational numbers
Exercise: Which set is larger?

- The set of:
  - Natural numbers $\mathbb{N}$, or
  - Positive rational numbers? $Q = \{ \frac{m}{n} \mid m, n \in \mathbb{N} \}$

- They are the same size! Both are countably infinite.

Another mapping: This is a bijection bc every natural number maps to a unique fraction, and vice versa.
Exercise: Which set is larger?

• The set of:
  • Natural numbers $\mathbb{N}$, or
  • Real numbers $\mathbb{R}$
• There are more real numbers. It is uncountably infinite.

Proof, by contradiction:
• Assume: a bijection between natural and real numbers exists.
  • So: every natural num maps to a unique real, and vice versa
But we show that in any given mapping, ...  
  • Some real number is not mapped to ...
  • E.g., a number that has different digits at each position:
    \[ x = 0.4641 \ldots \]
• This number cannot be in the mapping ...
• ... So we have a contradiction!
Georg Cantor

• Invented set theory

• Came up with countable infinity (1873)

• And uncountability:
  • Also: how to show uncountability with “diagonalization” technique
### Diagonalization with Turing Machines

**Diagonal:** Result of giving a TM its own encoding as input

|   | $M_1$ | $M_2$ | $M_3$ | $M_4$ | ... | $D$ | ...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$M_1$</td>
<td>accept</td>
<td>reject</td>
<td>accept</td>
<td>reject</td>
<td>accept</td>
<td>...</td>
<td>accept</td>
</tr>
</tbody>
</table>
| $M_2$ | accept | accept | accept | accept | accept | ... | accept ...
| $M_3$ | reject | reject | reject | reject | ... | reject |
| $M_4$ | accept | accept | reject | reject | accept | ...
| ... | ... | ... | ... | ... | ... | ...
| $D$ | reject | reject | accept | accept | ... | ...

- **Try to construct “opposite” TM $D$**
- **It must both accept and reject!**
- **What should happen here?**
- **TM $D$ can’t exist!**

- **Opposites**
- **All TMs**
**Thm:** $A_{TM}$ is undecidable

$A_{TM} = \{\langle M, w \rangle | M \text{ is a TM and } M \text{ accepts } w \}$

**Proof** by contradiction:

1. **Assume** $A_{TM}$ is decidable. So there exists a decider $H$ for it:
   
   $H(\langle M, w \rangle) = \begin{cases} 
   \text{accept} & \text{if } M \text{ accepts } w \\
   \text{reject} & \text{if } M \text{ does not accept } w
   \end{cases}$

2. **Use** $H$ in another TM ... the impossible “opposite” machine:
   
   $D = \text{“On input } \langle M \rangle, \text{ where } M \text{ is a TM:}$$$
   1. \text{Run } H \text{ on input } \langle M, \langle M \rangle \rangle.$$
   2. \text{Output the opposite of what } H \text{ outputs. That is, if } H \text{ accepts, reject; and if } H \text{ rejects, accept.”}$

   From previous slide (does opposite of what input TM would do if given itself)

   $H$ computes: $M$'s result with itself as input

   Do the opposite
Thm: $A_{TM}$ is undecidable

$A_{TM} = \{ \langle M, w \rangle | M \text{ is a TM and } M \text{ accepts } w \}$

Proof by contradiction:

1. Assume $A_{TM}$ is decidable. So there exists a decider $H$ for it:

   $$H(\langle M, w \rangle) = \begin{cases} 
   \text{accept} & \text{if } M \text{ accepts } w \\
   \text{reject} & \text{if } M \text{ does not accept } w 
   \end{cases}$$

2. Use $H$ in another TM $D$ ... the impossible “opposite” machine:

   $D = \text{“On input } \langle M \rangle, \text{ where } M \text{ is a TM:} \quad$
   
   1. Run $H$ on input $\langle M, \langle M \rangle \rangle$.
   2. Output the opposite of what $H$ outputs. That is, if $H$ accepts, reject; and if $H$ rejects, accept.”

3. But $D$ does not exist! **Contradiction**! So the assumption is false.
Easier Undecidability Proofs

• We proved \( A_{TM} = \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \} \) is undecidable...

• ... by contradiction:
  • Use hypothetical \( A_{TM} \) decider to create an **impossible** decider “D”!

  
  **reduce “D problem” to** \( A_{TM} \)

• Step # 1: **coming up with “D” --- hard!**
  • Need to invent **diagonalization**

• Step # 2: **reduce “D” problem to** \( A_{TM} \) --- **easier!**

• From now on: **undecidability proofs only need step # 2!**
  • And we now have two “impossible” problems to choose from...
The Halting Problem

**Thm:** $HALT_{TM}$ is undecidable

**Proof,** by contradiction:

- **Assume:** $HALT_{TM}$ has *decider* $R$; use it to create decider for $A_{TM}$:

  $$A_{TM} = \{(M, w) | M \text{ is a TM and } M \text{ accepts } w\}$$

- ...  

- But $A_{TM}$ is undecidable and has no decider!
The Halting Problem

Thm: $HALT_{TM}$ is undecidable

Proof, by contradiction:

- **Assume:** $HALT_{TM}$ has decider $R$; use it to create decider for $A_{TM}$:
  
  $A_{TM} = \{ \langle M, w \rangle | M$ is a TM and $M$ accepts $w \}$

  $S =$ “On input $\langle M, w \rangle$, an encoding of a TM $M$ and a string $w$:
  
  1. Run TM $R$ on input $\langle M, w \rangle$.
  2. If $R$ rejects, reject.
  3. If $R$ accepts, simulate $M$ on $w$ until it halts.
  4. If $M$ has accepted, accept; if $M$ has rejected, reject.”

  This means $M$ loops on input $w$

  This step always halts

Termination argument:

**Step 1:** $R$ is a decider so always halts
**Step 3:** $M$ always halts because $R$ said so
The Halting Problem

Thm: $HALT_{TM}$ is undecidable

Proof, by contradiction:

• **Assume**: $HALT_{TM}$ has *decider* $R$; use it to create decider for $A_{TM}$:

$$A_{TM} = \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \}$$

**$S = \text{"On input } \langle M, w \rangle, \text{ an encoding of a TM } M \text{ and a string } w:\n1. \text{ Run } TM \ R \text{ on input } \langle M, w \rangle.\n2. \text{ If } R \text{ rejects, reject.}\n3. \text{ If } R \text{ accepts, simulate } M \text{ on } w \text{ until it halts.}\n4. \text{ If } M \text{ has accepted, accept; if } M \text{ has rejected, reject."}**

• But $A_{TM}$ is undecidable! i.e., this decider does not exist!
  • So $HALT_{TM}$ is also undecidable!

**Undecidability Proof Technique #1:**
*Reduce (directly) from $A_{TM}$ (by creating $A_{TM}$ decider)*
Interlude: Reducing from $\text{HALT}_\text{TM}$

A practical thought experiment ... 
... about compiler optimizations

Your compiler changes your program!

If TRUE then A else B $\rightarrow$ A

1 + 2 + 3 $\rightarrow$ 6
Compiler Optimizations

Optimization - docs

- **-O0**
  - No optimization, faster compilation time, better for debugging builds.

- **-O2**
- **-O3**
  - Higher level of optimization. Slower compile-time, better for production builds.

- **-Ofast**
  - Enables higher level of optimization than (-O3). It enables lots of flags as can be seen `src (-ffloat-store, -ffast-math, -ffinite-math-only, -O3 ...)`

- **-finline-functions**
- **-m64**
- **-funroll-loops**
- **-fvectorize**
- **-fprofile-generate**

Types of optimization

Techniques used in optimization can be broken up among various scopes which can affect anything from a single statement to the entire program. Generally speaking, locally scoped techniques are easier to implement than global ones but result in smaller gains. Some examples of scopes include:

- **Peephole optimizations**
  - These are usually performed late in the compilation process after machine code has been generated. This form of optimization examines a few adjacent instructions (like "looking through a peephole" at the code) to see whether they can be replaced by a single instruction or a shorter sequence of instructions. For instance, a multiplication of a value by 2 might be more efficiently executed by left-shifting the value or by adding the value to itself (this example is also an instance of strength reduction).

- **Local optimizations**
  - These only consider information local to a basic block. Since basic blocks have no control flow, these optimizations need very little analysis, saving time and reducing storage requirements, but this also means that no information is preserved across jumps.

- **Global optimizations**
  - These are also called "interprocedural methods" and act on whole functions. This gives them more information to work with, but often makes expensive computations necessary. Worst case assumptions have to be made when function calls occur or global variables are accessed because little information about them is available.

- **Loop optimizations**
  - These act on the statements which make up a loop, such as a for loop, for example loop-invariant code motion. Loop optimizations can have a significant impact because many programs spend a large percentage of their time inside loops.

- **Prescient store optimizations**
  - These allow store operations to occur earlier than would otherwise be permitted in the context of threads and locks. The process needs some way of knowing ahead of time what value will be stored by the assignment that it should have followed. The purpose of this relaxation is to allow compiler optimization to perform certain kinds of code rearrangement that preserve the semantics of properly synchronized programs.

- **Interprocedural, whole-program or link-time optimization**
  - These analyze all of a program’s source code. The greater quantity of information extracted means that optimizations can be more effective compared to when they only have access to local information, i.e. within a single function. This kind of optimization can also allow new techniques to be performed. For instance, function inlining, where a call to a function is replaced by a copy of the function body.

- **Machine code optimization and object code optimizer**
  - These analyze the executable task image of the program after all of an executable machine code has been linked. Some of the techniques that can be applied in a more limited scope, such as macro compression which saves space by collapsing common sequences of instructions, are more effective when the entire executable task image is available for analysis.
The Optimal Optimizing Compiler

**Thm:** The Optimal (C++) Optimizing Compiler does not exist

**Proof,** by contradiction:

**Assume:** \( OPT \) is the Perfect Optimizing Compiler

Use it to create \( HALT_{TM} \) decider (accepts \( <M, w> \) if \( M \) halts with \( w \), else reject):

\( S = \) On input \( <M, w> \), where \( M \) is C++ program and \( w \) is string:

- If \( OPT(M) == \) for(;;)
  - a) Then **Reject**
  - b) Else **Accept**

---

**In computer science and mathematics,** a **full employment theorem** is a term used, often humorously, to refer to a theorem which states that no algorithm can optimally perform a particular task done by some class of professionals. The name arises because such a theorem ensures that there is endless scope to keep discovering new techniques to improve the way at least some specific task is done.

For example, the **full employment theorem for compiler writers** states that there is no such thing as a provably perfect size-optimizing compiler, as such a proof for the compiler would have to detect non-terminating computations and reduce them to a one-instruction infinite loop. Thus, the existence of a provably perfect size-optimizing compiler would imply a solution to the halting problem, which cannot exist. This also implies that there may always be a better compiler since the proof that one has the best compiler cannot exist. Therefore, compiler writers will always be able to speculate that they have something to improve.
Summary: The Limits of Algorithms

- \( A_{\text{DFA}} = \{ \langle B, w \rangle | B \text{ is a DFA that accepts input string } w \} \)  
  Decidable

- \( A_{\text{CFG}} = \{ \langle G, w \rangle | G \text{ is a CFG that generates string } w \} \)  
  Decidable

- \( A_{\text{TM}} = \{ \langle M, w \rangle | M \text{ is a TM and } M \text{ accepts } w \} \)  
  Undecidable

- \( \text{HALT}_{\text{TM}} = \{ \langle M, w \rangle | M \text{ is a TM and } M \text{ halts on input } w \} \)  
  Undecidable

It's straightforward to use hypothetical \( \text{HALT}_{\text{TM}} \) decider to create \( A_{\text{TM}} \) decider

Similar languages
Summary: The Limits of Algorithms

- $A_{DFA} = \{ \langle B, w \rangle | B \text{ is a DFA that accepts input string } w \}$ Decidable
- $A_{CFG} = \{ \langle G, w \rangle | G \text{ is a CFG that generates string } w \}$ Decidable
- $A_{TM} = \{ \langle M, w \rangle | M \text{ is a TM and } M \text{ accepts } w \}$ Undecidable
- $HALT_{TM} = \{ \langle M, w \rangle | M \text{ is a TM and } M \text{ halts on input } w \}$ Undecidable
- $E_{DFA} = \{ \langle A \rangle | A \text{ is a DFA and } L(A) = \emptyset \}$ Decidable
- $E_{CFG} = \{ \langle G \rangle | G \text{ is a CFG and } L(G) = \emptyset \}$ Decidable
- $E_{TM} = \{ \langle M \rangle | M \text{ is a TM and } L(M) = \emptyset \}$ Undecidable

How can we use a hypothetical $E_{TM}$ decider to create $A_{TM}$ or $HALT_{TM}$ decider?
Reducibility: Modifying the TM

**Thm:** \( E_{TM} \) is undecidable

**Proof**, by contradiction:

- **Assume** \( E_{TM} \) has **decider** \( R \); use it to create **decider** for \( A_{TM} \):

  1. **S** := “On input \( \langle M, w \rangle \), an encoding of a TM \( M \) and a string \( w \):
      - First, construct \( M_1 \)
      - Run \( R \) on input \( \langle M_1 \rangle \)
      - If \( R \) accepts, **reject** (because it means \( \langle M \rangle \) doesn’t accept \( w \))
      - If \( R \) rejects, then **accept** (\( \langle M \rangle \) accepts \( w \))

- **Idea:** Wrap \( \langle M \rangle \) in a new TM that can **only** accept \( w \):

\[
M_1 = \text{“On input } x: \begin{align*}
1. \text{If } x \neq w, \text{ reject.} \\
2. \text{If } x = w, \text{ run } M \text{ on input } w \text{ and accept if } M \text{ does.} \end{align*}
\]

\( M_1 \) accepts \( w \) if \( M \) does
Reducibility: Modifying the TM

Thm: $E_{TM}$ is undecidable

Proof, by contradiction:

• Assume $E_{TM}$ has decider $R$; use it to create decider for $A_{TM}$:

  $S = \text{"On input } \langle M, w \rangle \text{, an encoding of a TM } M \text{ and a string } w:\n\text{ First, construct } M_1\n\text{ Run } R \text{ on input } \langle M_1 \rangle \n\text{ If } R \text{ accepts, reject (because it means } \langle M \rangle \text{ doesn't accept } \langle w \rangle \n\text{ if } R \text{ rejects, then accept } \langle M \rangle \text{ accepts } \langle w \rangle \n\text{ Idea: Wrap } \langle M \rangle \text{ in a new TM that can only accept } w:\nM_1 = \text{"On input } x:\n\text{ 1. If } x \neq w, \text{ reject.}\n\text{ 2. If } x = w, \text{ run } M \text{ on input } w \text{ and accept if } M \text{ does."}
Summary: The Limits of Algorithms

- $A_{DFA} = \{ \langle B, w \rangle \mid B$ is a DFA that accepts input string $w \}$
  Decidable
- $A_{CFG} = \{ \langle G, w \rangle \mid G$ is a CFG that generates string $w \}$
  Decidable
- $A_{TM} = \{ \langle M, w \rangle \mid M$ is a TM and $M$ accepts $w \}$
  Undecidable
- $E_{DFA} = \{ \langle A \rangle \mid A$ is a DFA and $L(A) = \emptyset \}$
  Decidable
- $E_{CFG} = \{ \langle G \rangle \mid G$ is a CFG and $L(G) = \emptyset \}$
  Decidable
- $E_{TM} = \{ \langle M \rangle \mid M$ is a TM and $L(M) = \emptyset \}$
  Undecidable
- $EQ_{DFA} = \{ \langle A, B \rangle \mid A$ and $B$ are DFAs and $L(A) = L(B) \}$
  Decidable
- $EQ_{CFG} = \{ \langle G, H \rangle \mid G$ and $H$ are CFGs and $L(G) = L(H) \}$
  Undecidable
- $EQ_{TM} = \{ \langle M_1, M_2 \rangle \mid M_1$ and $M_2$ are TMs and $L(M_1) = L(M_2) \}$
  Undecidable
Reduce to something else: $EQ_{TM}$ is undecidable

$EQ_{TM} = \{\langle M_1, M_2 \rangle \mid M_1 \text{ and } M_2 \text{ are TMs and } L(M_1) = L(M_2)\}$

Proof, by contradiction:

• Assume: $EQ_{TM}$ has decider $R$; use it to create decider for $A_{TM}$:

$E_{TM} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) = \emptyset\}$

$S = \text{“On input } \langle M \rangle, \text{ where } M \text{ is a TM:}\\ 1. \text{ Run } R \text{ on input } \langle M, M_1 \rangle, \text{ where } M_1 \text{ is a TM that rejects all inputs.}\\ 2. \text{ If } R \text{ accepts, accept; if } R \text{ rejects, reject.”}
Reduce to something else: $EQ_{\text{TM}}$ is undecidable

$EQ_{\text{TM}} = \{\langle M_1, M_2 \rangle | M_1 \text{ and } M_2 \text{ are TMs and } L(M_1) = L(M_2) \}$

Proof, by contradiction:

• **Assume**: $EQ_{\text{TM}}$ has *decider* $R$; use it to create *decider* for $E_{\text{TM}}$:

\[
S = \text{"On input } \langle M \rangle, \text{ where } M \text{ is a TM:}
\]

1. Run $R$ on input $\langle M, M_1 \rangle$, where $M_1$ is a TM that rejects all inputs.
2. If $R$ accepts, accept; if $R$ rejects, reject.”

• But $E_{\text{TM}}$ is undecidable!
Summary: Undecidability Proof Techniques

• **Proof Technique #1:**
  - Use hypothetical decider to implement impossible \( A_{TM} \) decider
  - Example Proof: \( HALT_{TM} = \{\langle M, w \rangle | M \text{ is a TM and } M \text{ halts on input } w \} \)

• **Proof Technique #2:**
  - Use hypothetical decider to implement impossible \( A_{TM} \) decider
  - But **first modify the input** \( M \)
  - Example Proof: \( E_{TM} = \{\langle M \rangle | M \text{ is a TM and } L(M) = \emptyset \} \)

Can also combine these techniques

• **Proof Technique #3:**
  - Use hypothetical decider to implement non-\( A_{TM} \) impossible decider
  - Example Proof: \( EQ_{TM} = \{\langle M_1, M_2 \rangle | M_1 \text{ and } M_2 \text{ are TMs and } L(M_1) = L(M_2) \} \)
Summary: Decidability and Undecidability

- \( A_{DFA} = \{ \langle B, w \rangle | B \) is a DFA that accepts input string \( w \} \)  
  Decidable

- \( A_{CFG} = \{ \langle G, w \rangle | G \) is a CFG that generates string \( w \} \)  
  Decidable

- \( A_{TM} = \{ \langle M, w \rangle | M \) is a TM and \( M \) accepts \( w \} \)  
  Undecidable

- \( E_{DFA} = \{ \langle A \rangle | A \) is a DFA and \( L(A) = \emptyset \} \)  
  Decidable

- \( E_{CFG} = \{ \langle G \rangle | G \) is a CFG and \( L(G) = \emptyset \} \)  
  Decidable

- \( E_{TM} = \{ \langle M \rangle | M \) is a TM and \( L(M) = \emptyset \} \)  
  Undecidable

- \( E_{Q_{DFA}} = \{ \langle A, B \rangle | A \) and \( B \) are DFAs and \( L(A) = L(B) \} \)  
  Decidable

- \( E_{Q_{CFG}} = \{ \langle G, H \rangle | G \) and \( H \) are CFGs and \( L(G) = L(H) \} \)  
  Undecidable

- \( E_{Q_{TM}} = \{ \langle M_1, M_2 \rangle | M_1 \) and \( M_2 \) are TMs and \( L(M_1) = L(M_2) \} \)  
  Undecidable
Also Undecidable ...

• $\text{REGULAR}_{TM} = \{<M> \mid M \text{ is a TM and } L(M) \text{ is a regular language}\}$
Thm: $\text{REGULAR}_{\text{TM}}$ is undecidable

$\text{REGULAR}_{\text{TM}} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) \text{ is a regular language}\}$

Proof, by contradiction:

• Assume: $\text{REGULAR}_{\text{TM}}$ has decider $R$; use it to create decider for $A_{\text{TM}}$:

  $S = \text{"On input } \langle M, w \rangle, \text{ an encoding of a TM } M \text{ and a string } w$:

  • First, construct $M_2$ (??)
  • Run $R$ on input $\langle M_2 \rangle$
  • If $R$ accepts, accept; if $R$ rejects, reject

Want: $L(M_2) =$

• regular, if $M$ accepts $w$
• nonregular, if $M$ does not accept $w$
**Thm:** \( \text{REGULAR}_{TM} \) is undecidable (continued)

\[
\text{REGULAR}_{TM} = \{ \langle M \rangle | M \text{ is a TM and } L(M) \text{ is a regular language} \}
\]

\( M_2 = \) “On input \( x \):
1. If \( x \) has the form \( 0^n1^n \), accept.
2. If \( x \) does not have this form, run \( M \) on input \( w \) and accept if \( M \) accepts \( w \).”

\( L(M_2) = \) nonregular, so far

If \( M \) accepts \( w \), accept everything else, so \( L(M_2) = \Sigma^* = \text{regular} \)

Want: \( L(M_2) = \)
- **regular**, if \( M \) accepts \( w \)
- **nonregular**, if \( M \) does not accept \( w \)

if \( M \) does not accept \( w \), \( M_2 \) accepts all strings (regular lang)

All strings

if \( M \) accepts \( w \), \( M_2 \) accepts this nonregular lang
Also Undecidable ...

- \( \text{REGULAR}_{TM} = \{ <M> | M \text{ is a TM and } L(M) \text{ is a regular language} \} \)

- \( \text{CONTEXTFREE}_{TM} = \{ <M> | M \text{ is a TM and } L(M) \text{ is a CFL} \} \)

- \( \text{DECIDABLE}_{TM} = \{ <M> | M \text{ is a TM and } L(M) \text{ is a decidable language} \} \)

- \( \text{FINITE}_{TM} = \{ <M> | M \text{ is a TM and } L(M) \text{ is a finite language} \} \)

Seems like no algorithm can compute anything about the language of a Turing Machine, i.e., about the runtime behavior of programs ...
An Algorithm About Program Behavior?

Write a program that, given another program as its argument, returns TRUE if that argument prints “Hello, World!”

```
main()
{
    printf("hello, world\n");
}
```
Write a program that, given another program as its argument, returns `TRUE` if that argument prints "Hello, World!"

```c
main()
{
    if \( x^n + y^n = z^n \), for any integer \( n > 2 \)
    printf("hello, world\n");
}
```

Fermat's Last Theorem
(unknown for ~350 years, solved in 1990s)
Also Undecidable ...

- $\text{REGULAR}_{TM} = \{ <M> \mid M$ is a TM and $L(M)$ is a regular language $\}$

- $\text{CONTEXTFREE}_{TM} = \{ <M> \mid M$ is a TM and $L(M)$ is a CFL $\}$

- $\text{DECIDABLE}_{TM} = \{ <M> \mid M$ is a TM and $L(M)$ is a decidable language $\}$

- $\text{FINITE}_{TM} = \{ <M> \mid M$ is a TM and $L(M)$ is a finite language $\}$

- ...

- $\text{ANYTHING}_{TM} = \{ <M> \mid M$ is a TM and “… anything …” about $L(M)$ $\}$
Rice’s Theorem: $\textit{ANYTHING}_{\text{TM}}$ is Undecidable

$\textit{ANYTHING}_{\text{TM}} = \{<M> \mid M \text{ is a TM and … anything … about } L(M)\}$

• “... Anything …”, more precisely:
  For any $M_1, M_2$,
  • if $L(M_1) = L(M_2)$
  • then $M_1 \in \textit{ANYTHING}_{\text{TM}} \iff M_2 \in \textit{ANYTHING}_{\text{TM}}$

• Also, “... Anything …” must be “non-trivial”:
  • $\textit{ANYTHING}_{\text{TM}} \neq \{\}$
  • $\textit{ANYTHING}_{\text{TM}} \neq \text{set of all TMs}$
Rice’s Theorem: $\text{ANYTHING}_{\text{TM}}$ is Undecidable

$\text{ANYTHING}_{\text{TM}} = \{<M> \mid M \text{ is a TM and } \text{ ... anything ... about } L(M)\}$

Proof by contradiction

• Assume some language satisfying $\text{ANYTHING}_{\text{TM}}$ has a decider $R$.
  • Since $\text{ANYTHING}_{\text{TM}}$ is non-trivial, then there exists $M_{\text{any}} \in \text{ANYTHING}_{\text{TM}}$
  • Where $R$ accepts $M_{\text{any}}$

• Use $R$ to create decider for $A_{\text{TM}}$:

On input $<M, w>$:

• Create $M_w$:
  
  $M_w = \text{on input } x$:  
  - Run $M$ on $w$
  - If $M$ rejects $w$: reject $x$
  - If $M$ accepts $w$:  
    Run $M_{\text{any}}$ on $x$ and accept if it accepts, else reject

• Run $R$ on $M_w$
  • If it accepts, then $M_w = M_{\text{any}}$, so $M$ accepts $w$, so accept
  • Else reject

Wait! What if the TM that accepts nothing is in $\text{ANYTHING}_{\text{TM}}$?

These two cases must be different (so $R$ can distinguish when $M$ accepts $w$)

Proof still works! Just use the complement of $\text{ANYTHING}_{\text{TM}}$ instead!
Rice’s Theorem Implication

\{<M> \mid M \text{ is a TM that installs malware}\}

Undecidable!
(by Rice’s Theorem)