Lecture 27: Undecidability

```c
DEFINE DOESITHALT(PROGRAM):
{
    RETURN TRUE;
}
```

THE BIG PICTURE SOLUTION TO THE HALTING PROBLEM

Be sure to submit your assignment in Gradescope. If you get another chance.
Final exam Monday, Review session Sunday

• Monday at either 2:30-4:20 or 4:30-6:20
  – JHN 102
  – Must select your exam time by Saturday
    No changes permitted after that
  – Bring your UW ID

• Comprehensive: Full probs only on topics that were covered in homework. May have small probs on other topics.
  – May include pre-midterm topics, e.g., formal proofs.
  – Reference sheets will be included. Closed book. No notes.

• Review session: *Sunday starting at 1 pm on Zoom*
  – Bring your questions !!
A set $S$ is **countable** iff we can order the elements of $S$ as

$$S = \{x_1, x_2, x_3, \ldots \}$$

**Countable sets:**

- $\mathbb{N}$ - the natural numbers
- $\mathbb{Z}$ - the integers
- $\mathbb{Q}$ - the rationals
- $\Sigma^*$ - the strings over any finite $\Sigma$

$\mathbb{N}$, $\mathbb{Z}$, and $\mathbb{Q}$ are shown by "dovetailing".

The set of all Java programs is not countable.
Last time: Not every set is countable

**Theorem [Cantor]:**
The set of real numbers between 0 and 1 is not countable.

Proof using “diagonalization”.
Last time: Proof that \([0,1)\) is not countable

Suppose, for the sake of contradiction, that there is a list of them:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>7</th>
<th>1</th>
<th>4</th>
<th>...</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>(r_1)</td>
<td>0.</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(r_2)</td>
<td>0.</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>(r_3)</td>
<td>0.</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>(r_4)</td>
<td>0.</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Flipping rule:

- If digit is 5, make it 1.
- If digit is not 5, make it 5.

For every \(n \geq 1\):

\[ r_n \neq d = 0.\hat{x}_{11}\hat{x}_{22}\hat{x}_{33}\hat{x}_{44}\hat{x}_{55} \ldots \]

because the numbers differ on the \(n\)-th digit!

So the list is incomplete, which is a contradiction.

Thus the real numbers between 0 and 1 are **not countable**: “uncountable”
A note on this proof

• The set of rational numbers in \([0,1)\) also have decimal representations like this
  – The only difference is that rational numbers always have repeating decimals in their expansions 0.33333... or .25000000...

• So why wouldn’t the same proof show that this set of rational numbers is uncountable?
  – Given any listing we could create the flipped diagonal number \(d\) as before
  – However, \(d\) would not have a repeating decimal expansion and so wouldn’t be a rational number
    It would not be a “missing” number, so no contradiction.
Last time:

The set of all functions $f : \mathbb{N} \to \{0, \ldots, 9\}$ is uncountable.

Supposed listing of all the functions:

<table>
<thead>
<tr>
<th>$f_1$</th>
<th>$f_2$</th>
<th>$f_3$</th>
<th>$f_4$</th>
<th>$f_5$</th>
<th>$f_6$</th>
<th>$f_7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 1 0 0 0</td>
<td>3 3 5 3 3</td>
<td>1 4 2 5 8 5 7 1 4</td>
<td>1 4 1 5 1 9 2 6 5</td>
<td>1 2 1 2 2</td>
<td>2 5 0 0 0</td>
<td>7 1 8 2 8 1 8 2</td>
</tr>
</tbody>
</table>

Flipping rule:
- If $f_n(n) = 5$, set $D(n) = 1$
- If $f_n(n) \neq 5$, set $D(n) = 5$

For all $n$, we have $D(n) \neq f_n(n)$. Therefore $D \neq f_n$ for any $n$ and the list is incomplete!  

$\Rightarrow \{f \mid f : \mathbb{N} \to \{0,1, \ldots, 9\}\}$ is **not** countable.
Last time: Uncomputable functions

We have seen that:

- The set of all (Java) programs is countable
- The set of all functions \( f : \mathbb{N} \to \{0, \ldots, 9\} \) is not countable

So: There must be some function \( f : \mathbb{N} \to \{0, \ldots, 9\} \) that is not computable by any program!
Uncomputable functions

Interesting... maybe.

Can we come up with an explicit function that is uncomputable?
A “Simple” Program

```java
public static void collatz(n) {
    if (n == 1) {
        return 1;
    }
    if (n % 2 == 0) {
        return collatz(n/2);
    } else {
        return collatz(3*n + 1);
    }
}
```

What does this program do?

... on n=11?

... on n=100000000000000000001?
A “Simple” Program

public static void collatz(n) {
    if (n == 1) {
        return 1;
    }
    if (n % 2 == 0) {
        return collatz(n/2)
    }
    else {
        return collatz(3*n + 1)
    }
}

Nobody knows whether or not this program halts on all inputs!

What does this program do?
... on n=11?
... on n=10000000000000000001?
Recall our language picture

- All
- Context-Free
  - Binary Palindromes
  - Regular
    - DFA
    - NFA
    - Regex
  - Finite
    - \{001, 10, 12\}
- Java
Some Notation

We’re going to be talking about Java code.

\[ \text{CODE}(P) \] will mean “the code of the program \( P \)”

So, consider the following function:

```java
public String P(String x) {
    return new String(Arrays.sort(x.toCharArray()));
}
```

What is \( P(\text{CODE}(P)) \)?

“((((())));AACPSSaaabceegghiiiiilnnnnnoooprrrrrrrrrrsssttttttttuuwxxyy{}”
The Halting Problem

CODE(P) means “the code of the program P”

Given: - CODE(P) for any program P
        - input x

Output: true if P halts on input x
        false if P does not halt on input x
Undecidability of the Halting Problem

**The Halting Problem**

**Given:**
- CODE(P) for any program P
- input x

**Output:**
- true if P halts on input x
- false if P does not halt on input x

**Theorem [Turing]:** There is no program that solves the Halting Problem

\[
\text{CODE}(P, x)
\]
Proof by contradiction

Suppose that $H$ is a Java program that solves the Halting problem.
Proof by contradiction

Suppose that \( H \) is a Java program that solves the Halting problem.

Then we can write this program:

```java
public static void D(String s) {
    if (H(s, s) == true) {
        while (true);  // don’t halt
    } else {
        return;        // halt
    }
}

public static bool H(String s, String x) { ... }
```

Does \( D(CODE(D)) \) halt?
Does $D(CODE(D))$ halt?

```java
public static void $D(s)$ {
    if ($H(s,s) == true$) {
        ...
    } else {
        ...
    }
}
```
Does \( D(\text{CODE}(D)) \) halt?

\[ s = \text{CODE}(D) \]

\( H \) solves the halting problem implies that

\[ H(\text{CODE}(D),s) \text{ is true iff } D(s) \text{ halts}, \quad H(\text{CODE}(D),s) \text{ is false iff not} \]

```
public static void D(s) {
    if (H(s,s) == true) {
        \[ \text{...} \] // 1
    } else {
        \[ \text{...} \] // 2
    }
}
```
Does \( D(\text{CODE}(D)) \) halt?

\[
\text{public static void } D(s) \{ \\
\quad \text{if } (H(s,s) == \text{true}) \{ \\
\quad\quad \text{while (true);} \quad \text{// don’t halt} \\
\quad\} \text{ else } \{ \\
\quad\quad \ldots \\
\quad\} \\
\}
\]

\( H \) solves the halting problem implies that
\( H(\text{CODE}(D),s) \) is \textbf{true} \iff \( D(s) \) halts, \( H(\text{CODE}(D),s) \) is \textbf{false} \iff \text{not
Suppose that \( D(\text{CODE}(D)) \) halts.

Then, by definition of \( H \) it must be that

\[
H(\text{CODE}(D), \text{CODE}(D)) \text{ is true}
\]

Which by the definition of \( D \) means \( D(\text{CODE}(D)) \) doesn’t halt.
Does \( D(\text{CODE}(D)) \) halt?

\[ H \] solves the halting problem implies that

\[ H(\text{CODE}(D),s) \text{ is true iff } D(s) \text{ halts, } H(\text{CODE}(D),s) \text{ is false iff not} \]

Suppose that \( D(\text{CODE}(D)) \) halts.

Then, by definition of \( H \) it must be that

\[ H(\text{CODE}(D), \text{CODE}(D)) \text{ is true} \]

Which by the definition of \( D \) means \( D(\text{CODE}(D)) \) doesn’t halt.

```java
public static void D(s) {
    if (H(s,s) == true) {
        while (true); // don’t halt
    } else {
        return; // halt
    }
}
```
Does $D(\text{CODE}(D))$ halt?

$H$ solves the halting problem implies that

$H(\text{CODE}(D),s)$ is **true** iff $D(s)$ halts, $H(\text{CODE}(D),s)$ is **false** iff not

Suppose that $D(\text{CODE}(D))$ **halts**.

Then, by definition of $H$ it must be that

$H(\text{CODE}(D), \text{CODE}(D))$ is **true**

Which by the definition of $D$ means $D(\text{CODE}(D))$ **doesn't halt**

Suppose that $D(\text{CODE}(D))$ **doesn't halt**.

Then, by definition of $H$ it must be that

$H(\text{CODE}(D), \text{CODE}(D))$ is **false**

Which by the definition of $D$ means $D(\text{CODE}(D))$ **halts**

```java
public static void D(s) {
    if (H(s,s) == true) {
        while (true);  // don’t halt
    } else {
        return;  // halt
    }
}
```
H solves the halting problem implies that
\( H(\text{CODE}(D), s) \) is \textbf{true} iff \( D(s) \) halts, \( H(\text{CODE}(D), s) \) is \textbf{false} iff not

Suppose that \( D(\text{CODE}(D)) \) \textbf{halts}.
Then, by definition of \( H \) it must be that
\( H(\text{CODE}(D), \text{CODE}(D)) \), \textbf{true}.
Which by the definition of \( D \) means \( D(\text{CODE}(D)) \) \textbf{doesn’t halt}

Suppose that \( D(\text{CODE}(D)) \) \textbf{doesn’t halt}.
Then, by definition of \( H \) it must be that
\( H(\text{CODE}(D), \text{CODE}(D)) \) is \textbf{false}.
Which by the definition of \( D \) means \( D(\text{CODE}(D)) \) \textbf{halts}

\( D(\text{CODE}(D)) \) \textbf{halts}?

The \textbf{ONLY} assumption was that the program \( H \) exists so that assumption must have been false.

Contradiction!
We proved that there is no computer program that can solve the Halting Problem.

There was nothing special about Java*

[Church-Turing thesis]

This tells us that there is no compiler that can check our programs and guarantee to find any infinite loops they might have.
Terminology

• With state machines, we say that a machine “recognizes” the language $L$ iff
  – it accepts $x \in \Sigma^*$ if $x \in L$
  – it rejects $x \in \Sigma^*$ if $x \notin L$

• With Java programs / general computation, we say that the computer “decides” the language $L$ iff
  – it halts with output 1 on input $x \in \Sigma^*$ if $x \in L$
  – it halts with output 0 on input $x \in \Sigma^*$ if $x \notin L$
  (difference is the possibility that machine doesn’t halt)

• If no machine decides $L$, then $L$ is “undecidable”
Where did the idea for creating D come from?

```
public static void D(s) {
    if (H(s,s) == true) {
        while (true); // don’t halt
    } else {
        return; // halt
    }
}
```

halts on input code(P) iff $H(\text{code}(P),\text{code}(P))$ outputs false
iff P doesn’t halt on input code(P)
Connection to diagonalization

<table>
<thead>
<tr>
<th></th>
<th>&lt;P₁&gt;</th>
<th>&lt;P₂&gt;</th>
<th>&lt;P₃&gt;</th>
<th>&lt;P₄&gt;</th>
<th>&lt;P₅&gt;</th>
<th>&lt;P₆&gt;</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P₂</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P₃</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P₄</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P₅</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P₆</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P₇</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P₈</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P₉</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Some possible inputs x

This listing of all programs really does exist since the set of all Java programs is countable.

The goal of this “diagonal” argument is not to show that the listing is incomplete but rather to show that a “flipped” diagonal element is not in the listing.

Write <P> for CODE(P)
### Connection to diagonalization

<table>
<thead>
<tr>
<th></th>
<th>&lt;P₁&gt;</th>
<th>&lt;P₂&gt;</th>
<th>&lt;P₃&gt;</th>
<th>&lt;P₄&gt;</th>
<th>&lt;P₅&gt;</th>
<th>&lt;P₆&gt;</th>
<th>....</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P₂</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P₃</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P₄</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P₅</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P₆</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P₇</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P₈</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>P₉</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>All programs P</th>
<th>&lt;P₁&gt;</th>
<th>&lt;P₂&gt;</th>
<th>&lt;P₃&gt;</th>
<th>&lt;P₄&gt;</th>
<th>&lt;P₅&gt;</th>
<th>&lt;P₆&gt;</th>
<th>....</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₉</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>P₈</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>P₇</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P₆</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>P₅</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P₄</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>P₃</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P₂</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P₁</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Write <P> for CODE(P)

Some possible inputs x

(P,x) entry is **1** if program P halts on input x and **0** if it runs forever
**Connection to diagonalization**

Write \(<P>\) for \(\text{CODE}(P)\)

<table>
<thead>
<tr>
<th>(P)</th>
<th>(x)</th>
<th>(P_1)</th>
<th>(P_2)</th>
<th>(P_3)</th>
<th>(P_4)</th>
<th>(P_5)</th>
<th>(P_6)</th>
<th>(P_7)</th>
<th>(P_8)</th>
<th>(P_9)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(P_1)</td>
<td>01</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(P_2)</td>
<td>1</td>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(P_3)</td>
<td>1</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(P_4)</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>01</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>(P_5)</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(P_6)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(P_7)</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(P_8)</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>10</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>(P_9)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Some possible inputs \(x\)

\((P,x)\) entry is 1 if program \(P\) halts on input \(x\) and 0 if it runs forever.

Want behavior of program \(D\) to be like the flipped diagonal, so it can’t be in the list of all programs.
Where did the idea for creating \( D \) come from?

```java
public static void D(s) {
    if (H(s, s) == true) {
        while (true); /* don’t halt */
    }
    else {
        return; /* halt */
    }
}
```

\( D \) halts on input \( \text{code}(P) \) iff \( H(\text{code}(P), \text{code}(P)) \) outputs false iff \( P \) doesn’t halt on input \( \text{code}(P) \)

Therefore, for any program \( P \), \( D \) differs from \( P \) on input \( \text{code}(P) \)