Unsolvable Problems
Part One
Outline for Today

• **Self-Reference**
  • Party tricks of great importance.

• **TMs as Programs**
  • A little simplification.

• **Self-Defeating Objects**
  • Objects too “powerful” to exist.

• **Undecidable Problems**
  • Something beyond the reach of algorithms.
Recap from Last Time
The *Church-Turing Thesis* claims that every effective method of computation is either equivalent to or weaker than a Turing machine.

“This is not a theorem – it is a falsifiable scientific hypothesis. And it has been thoroughly tested!”

- Ryan Williams
R and RE

A language $L$ is **recognizable** if there is a TM $M$ with the following property:

$$\forall w \in \Sigma^*. (M \text{ accepts } w \leftrightarrow w \in L).$$

That is, for any string $w$:

- If $w \in L$, then $M$ accepts $w$.
- If $w \notin L$, then $M$ does not accept $w$.
  - It might reject $w$, or it might loop on $w$.
- This is a “weak” notion of solving a problem.
- The class $\text{RE}$ consists of all the recognizable languages.
A language $L$ is **decidable** if there is a TM $M$ with the following properties:

\[ \forall w \in \Sigma^*. (M \text{ accepts } w \leftrightarrow w \in L). \]

$M$ halts on all inputs.

That is, for any string $w$:

- If $w \in L$, then $M$ accepts $w$.
- If $w \notin L$, then $M$ rejects $w$.

This is a “strong” notion of solving a problem.

The class $\mathbf{R}$ consists of all the decidable languages.
Object Encodings

• Think about files on your computer:
  • each file represents some data, but
  • each file is encoded purely using 0s and 1s.
• If $Obj$ is an object, then $\langle Obj \rangle$ denotes some string representing $Obj$.
  • Think of it as how you’d store $Obj$ on disk.
• We can encode multiple objects as a single string. For example, if $M$ is a TM and $w$ is a string, then $\langle M, w \rangle$ is a string representing the pair of $M$ and $w$. 
The Universal Turing Machine

There is a TM named \( U_{TM} \) that is a universal Turing machine.

\( U_{TM} \) takes as input a pair \( \langle M, w \rangle \), where \( M \) is a TM and \( w \) is a string.

\( U_{TM} \) does to \( \langle M, w \rangle \) whatever \( M \) does to \( w \).
The Language $A_{TM}$

- The *acceptance language for Turing machines*, denoted $A_{TM}$, is the language of the universal Turing machine:

  $$A_{TM} = \mathcal{L}(U_{TM})$$

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

- Useful fact:

  $$\langle M, w \rangle \in A_{TM} \iff M \text{ accepts } w.$$ 

- Because $A_{TM} = \mathcal{L}(U_{TM})$, we know that $A_{TM} \in \text{RE}$. 
Teaser #1:

This language $A_{\text{TM}}$ has some interesting properties beyond what we’ve seen here.
New Stuff!
Self-Referential Software
Quines

- A *Quine* is a program that, when run, prints its own source code.
- Quines aren't allowed to just read the file containing their source code and print it out; that's cheating (and technically incorrect if someone changes that file!)
- How would you write such a program?
Writing a Quine
Self-Referential Programs

- **Claim:** Going forward, assume that any program can be augmented to include a method called `mySource()` that returns a string representation of its source code.

- General idea:
  - Write the initial program with `mySource()` as a placeholder.
  - Use the Quine technique we just saw to convert the program into something self-referential.
  - Now, `mySource()` magically works as intended.
Self-Referential Programs

- The fact that we can write Quines is not a coincidence.

- **Theorem (Kleene’s Second Recursion Theorem):** It is possible to construct TMs that perform arbitrary computations on their own “source code” (the string encoding of the TM).

- In other words, any computing system that’s equal to a Turing machine possesses some mechanism for self-reference!

- Want to see how deep the rabbit hole goes? Take CS154!
Teaser #2:

Self-reference lets machines compute on themselves. That lets them do Cruel and Unusual Things.
A Note on TM/Program Equivalence
**Equivalence of TMs and Programs**

- Every TM
  - receives some input,
  - does some work, then
  - (optionally) accepts or rejects.
- We can model a TM as a computer program where
  - the input is provided by a special method `getInput()` that returns the input to the program,
  - the program's logic is written in a normal programming language, and
  - the program (optionally) calls the special method `accept()` to immediately accept the input and `reject()` to immediately reject the input.
Equivalence of TMs and Programs

- Here's a sample program we might use to model a Turing machine for \( \{ w \in \{a, b\}^* \mid w \text{ has the same number of } a \text{'s and } b \text{'s } \} \):

```cpp
int main() {
    string input = getInput();
    int difference = 0;

    for (char ch: input) {
        if (ch == 'a') difference++;
        else if (ch == 'b') difference--;
        else reject();
    }

    if (difference == 0) accept();
    else reject();
}
```
Equivalence of TMs and Programs

- As mentioned before, it's always possible to build a method `mySource()` into a program, which returns the source code of the program.
- For example, here's a narcissistic program:

```java
int main() {
    string me = mySource();
    string input = getInput);

    if (input == me) accept();
    else reject();
}
```
Equivalence of TMs and Programs

• Sometimes, TMs use other TMs as subroutines.
• We can think of a decider for a language as a method that takes in some number of arguments and returns a boolean.
• For example, a decider for \( \{ a^n b^n \mid n \in \mathbb{N} \} \) might be represented in software as a method with this signature:

  ```
  bool isAnBn(string w);
  ```

• Similarly, a decider for \( \{ (m, n) \mid m, n \in \mathbb{N} \text{ and } m \text{ is a multiple of } n \} \) might be represented in software as a method with this signature:

  ```
  bool isMultipleOf(int m, int n);
  ```
Self-Defeating Objects
A *self-defeating object* is an object whose essential properties ensure it doesn’t exist.
**Question:** Why is there no largest integer?

**Answer:** Because if $n$ is the largest integer, what happens when we look at $n+1$?
Self-Defeating Objects

**Theorem:** There is no largest integer.

**Proof sketch:** Suppose for the sake of contradiction that there is a largest integer. Call that integer $n$. Consider the integer $n+1$. Notice that $n < n+1$. But then $n$ isn’t the largest integer. Contradiction! ■-ish
Self-Defeating Objects

**Theorem:** There is no largest integer.

**Proof sketch:** Suppose for the sake of contradiction that there is a largest integer. Call that integer $n$. Consider the integer $n+1$. Notice that $n < n+1$. But then $n$ isn’t the largest integer. Contradiction! ■-ish

We’re using $n$ to construct something that undermines $n$, hence the term “self-defeating.”
An Important Detail
**Claim:** There is a largest integer.

**Proof:** Assume $x$ is the largest integer. Notice that $x > x - 1$. So there’s no contradiction. ■-ish

Careful – we’re assuming what we’re trying to prove!

Careful – we’re assuming what we’re trying to prove!

How do we know there’s no contradiction? We just checked one case.
Self-Defeating Objects

• If you can show
  \[ x \text{ exists} \rightarrow \bot \]
  then you know that \( x \) doesn’t exist. (This is a proof by contradiction.)

• If you can show
  \[ x \text{ exists} \rightarrow \top \]
  you cannot conclude that \( x \) exists. (This is not a valid proof technique.)
Teaser #3:

Certain Turing machines can’t exist, as they’d be self-defeating objects.
Time-Out for Announcements!
Problem Sets

- Problem Set Eight is due this Friday at 2:30PM.
  - Have questions? Stop by office hours or post online on Piazza! Feel free to post privately if you’d like.
  - This is the last problem set where you can use late days. One late day extends the deadline to Saturday at 2:30PM, and another extends the deadline to Sunday at 2:30PM.
- We’re working on grading Problem Set Seven. We’ll aim to return it tomorrow.
Your Questions
“Is this course and an intro stats course enough to take CS 161?”

Yep, that’s what CS103 and CS109 are designed to do. If you haven’t taken CS109 but have taken something comparable in another department (Stats 116, CME 106, etc.), you should be good to go!

Have fun in CS161 – that material is beautiful and I hope you enjoy it as much as I think you will. 😊
“What are some of the interesting areas that computer science theoreticians are doing research on today?”

Oh, there’s a bunch. A sampler of what our theory group is working on:

1. Discovering new properties of polynomials and using them to reduce the sizes of data centers.
2. Finding ways to check if machine learning would help you solve a problem, given that you only have limited data available.
3. Designing models of computation powerful enough to answer interesting questions, but too weak to discriminate against disfavored groups.
4. Exploring Nash equilibria (from game theory) from a computational perspective.
5. Getting approximate solutions to important, practical problems that we think can’t be solved efficiently.
Back to CS103!
Learning About a String

• Suppose $M$ is a recognizer for some important language.

• We have a string $w$ and we really, really want to know whether $w \in \mathcal{L}(M)$.

• How could we do this?
Observation:

\[ w \in \mathcal{L}(M) \quad \text{if and only if} \quad M \text{ accepts } w. \]
Learning About a String

- **Option 1:** Run $M$ on $w$.
- What could happen?
  - $M$ could accept $w$. Great! We know $w \in \mathcal{L}(M)$.
  - $M$ could reject $w$. Great! We know $w \notin \mathcal{L}(M)$.
  - $M$ could loop on $w$. Hmmm. We’ve learned nothing.
- This won’t always tell us whether $w \in \mathcal{L}(M)$. We’ll need a different strategy.
If you want to know whether this is true...

**Observation:**

\[ w \in \mathcal{L}(M) \]

if and only if

\[ M \text{ accepts } w \]

if and only if

\[ \langle M, w \rangle \in A_{TM} \]

... you can try to determine whether this is true.
Learning About a String

- **Option 2:** Use the universal Turing machine, which is a recognizer for $A_{TM}$!
- Specifically, run $U_{TM}$ on $\langle M, w \rangle$.
- What could happen?
  - $U_{TM}$ could accept $\langle M, w \rangle$. Great! Then $w \in \mathcal{L}(M)$.
  - $U_{TM}$ could reject $\langle M, w \rangle$. Great! Then $w \notin \mathcal{L}(M)$.
  - $U_{TM}$ could loop on $\langle M, w \rangle$. Hmmm. We’ve learned nothing.
- This won’t always tell us whether $w \in \mathcal{L}(M)$. We’ll need a different strategy.
Option 2: Use the universal Turing machine, which is a recognizer for $A_{TM}$!

Specifically, run $U_{TM}$ on $\langle M, w \rangle$.

What could happen?

- $U_{TM}$ could accept $\langle M, w \rangle$. Great! Then $w \in \mathcal{L}(M)$.
- $U_{TM}$ could reject $\langle M, w \rangle$. Great! Then $w \notin \mathcal{L}(M)$.
- $U_{TM}$ could loop on $\langle M, w \rangle$. Hmmm. We’ve learned nothing.

This won’t always tell us whether $w \in \mathcal{L}(M)$. We’ll need a different strategy.

What if we had a decider for $A_{TM}$, not a recognizer?
Learning About a String

- **Option 3:** Build a *decider* for $A_{TM}$, rather than just a recognizer.

Specifically, build a decider for $A_{TM}$, then run that decider on $\langle M, w \rangle$.

- What could happen?
  - The decider could accept $\langle M, w \rangle$. Then $w \in \mathcal{L}(M)$.
  - The decider could reject $\langle M, w \rangle$. Then $w \notin \mathcal{L}(M)$.

- **Question:** How do we build this decider?
Claim: A decider for \( A_{\text{TM}} \) is a self-defeating object. It therefore doesn’t exist.
A Self-Defeating Object

• Let’s suppose that, somehow, we managed to build a decider for $A_{TM}$.

• Schematically, that decider would look like this:

![Diagram of decider](image)

• We could represent this decider in software as a method:

```cpp
text
bool willAccept(string program, string input);
```

that takes as input a program and a string, then returns whether that program will accept that string.
What does this program do?

```c
bool willAccept(string program, string input) {
    /* ... some implementation ... */
}
```
What does this program do?

```c
bool willAccept(string program, string input) {
    /* … some implementation … */
}

int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) {
        reject();
    } else {
        accept();
    }
}
```
What does this program do?

```c
bool willAccept(string program, string input) {
    /* ... some implementation ... */
}

int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) {
        reject();
    } else {
        accept();
    }
}
```

This program now has its own source code stored in variable `me`.

The program now asks – am I going to accept my input?
What does this program do?

```cpp
bool willAccept(string program, string input) {
    /* ... some implementation ... */
}

int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) {
        reject();
    } else {
        accept();
    }
}
```

Try running this program on any input. What happens if

- this program accepts its input?
- this program doesn’t accept its input?
What does this program do?

```cpp
bool willAccept(string program, string input) {
    /* ... some implementation ... */
}

int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) {
        reject();
    } else {
        accept();
    }
}
```

Try running this program on any input. What happens if...

... this program accepts its input?
What does this program do?

```cpp
bool willAccept(string program, string input) {
    /* … some implementation … */
}

int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) {
        reject();
    } else {
        accept();
    }
}
```

Try running this program on any input. What happens if...

... this program accepts its input?
What does this program do?

```cpp
bool willAccept(string program, string input) {
    /* … some implementation … */
}

int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) {
        reject();
    } else {
        accept();
    }
}
```

Try running this program on any input. What happens if

... this program accepts its input?
What does this program do?

```cpp
bool willAccept(string program, string input) {
    /* … some implementation … */
}

int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) {
        reject();
    } else {
        accept();
    }
}
```

Try running this program on any input. What happens if

... this program accepts its input?
What does this program do?

```cpp
bool willAccept(string program, string input) {
    /* ... some implementation ... */
}

int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) {
        reject();
    } else {
        accept();
    }
}
```

Try running this program on any input. What happens if

... this program accepts its input?

It rejects the input!
What does this program do?

```cpp
bool willAccept(string program, string input) {
    /* ... some implementation ... */
}

int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) {
        reject();
    } else {
        accept();
    }
}
```

Try running this program on any input. What happens if

... this program accepts its input? It rejects the input!
What does this program do?

```cpp
bool willAccept(string program, string input) {
    /* ... some implementation ... */
}

int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) {
        reject();
    } else {
        accept();
    }
}
```

Try running this program on any input.

What happens if

... this program accepts its input?

It rejects the input!

... this program doesn't accept its input?
What does this program do?

```cpp
bool willAccept(string program, string input) {
    /* ... some implementation ... */
}

int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) {
        reject();
    } else {
        accept();
    }
}
```

Try running this program on any input. What happens if

... this program accepts its input? It rejects the input!

... this program doesn't accept its input?
What does this program do?

```c
bool willAccept(string program, string input) {
    /* … some implementation … */
}

int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) {
        reject();
    } else {
        accept();
    }
}
```

Try running this program on any input. What happens if...

... this program accepts its input?

It rejects the input!

... this program doesn't accept its input?
What does this program do?

```cpp
#include <iostream>

bool willAccept(string program, string input) {
    /* ... some implementation ... */
}

int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) {
        reject();
    } else {
        accept();
    }
}
```

Try running this program on any input. What happens if...

... this program accepts its input? It rejects the input!

... this program doesn't accept its input?
What does this program do?

```cpp
bool willAccept(string program, string input) {
    /* ... some implementation ... */
}

int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) {
        reject();
    } else {
        accept();
    }
}
```

Try running this program on any input. What happens if

... this program accepts its input?

It rejects the input!

... this program doesn't accept its input?
What does this program do?

```c
bool willAccept(string program, string input) {
    /* … some implementation … */
}

int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) {
        reject();
    } else {
        accept();
    }
}
```

Try running this program on any input.
What happens if

... this program accepts its input?
It rejects the input!

... this program doesn't accept its input?
What does this program do?

```
bool willAccept(string program, string input) {
    /* … some implementation … */
}

int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) {
        reject();
    } else {
        accept();
    }
}
```

Try running this program on any input. What happens if ...
... this program accepts its input? It rejects the input!
... this program doesn't accept its input? It accepts the input!

What does this program do?
What does this program do?

```cpp
bool willAccept(string program, string input) {
    /* ... some implementation ... */
}

int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) {
        reject();
    } else {
        accept();
    }
}
```

Try running this program on any input. What happens if...

... this program accepts its input?
It rejects the input!

... this program doesn't accept its input?
It accepts the input!
What does this program do?

```cpp
bool willAccept(string program, string input) {
    /* ... some implementation ... */
}

int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) {
        reject();
    } else {
        accept();
    }
}
```

A self-defeating object

Using that object against itself.
What does this program do?

```cpp
bool willAccept(string program, string input) {
    /* ... some implementation ... */
}

int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) {
        reject();
    } else {
        accept();
    }
}
```

"The largest integer, n."

"The number \(n + 1\)."
**Theorem:** There is no largest integer.

**Proof sketch:** Suppose for the sake of contradiction that there is a largest integer. Call that integer $n$. Consider the integer $n+1$. Notice that $n < n+1$. But then $n$ isn’t the largest integer. Contradiction! ■-ish

```cpp
bool willAccept(string program, string input) {
    /* …some implementation… */
}

int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) {
        reject();
    } else {
        accept();
    }
}
```
Theorem: There is no largest integer.

Proof sketch: Suppose for the sake of contradiction that there is a largest integer. Call that integer $n$.

Consider the integer $n+1$.

Notice that $n < n+1$.

But then $n$ isn't the largest integer.

Contradiction! ■

Assume there exists this object $x$ which has these properties that are too powerful to actually work.
What does this program do?

```java
int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) {
        reject();
    } else {
        accept();
    }
}
```

**Theorem:** There is no largest integer.

**Proof sketch:** Suppose for the sake of contradiction that there is a largest integer. Call that integer $n$. Consider the integer $n+1$. Notice that $n < n+1$. But then $n$ isn’t the largest integer. Contradiction! ■-ish

Use the purported properties of $x$ against itself to create a contradiction.
What does this program do?

**Theorem:** There is no largest integer.

**Proof sketch:** Suppose for the sake of contradiction that there is a largest integer. Call that integer $n$. Consider the integer $n+1$. Notice that $n < n+1$. But then $n$ isn't the largest integer. Contradiction! ■-ish

Thus, this object $x$ cannot exist!
What does this program do?

**Theorem:** There is no largest integer.

**Proof sketch:** Suppose for the sake of contradiction that there is a largest integer. Call that integer $n$. Consider the integer $n+1$. Notice that $n < n+1$. But then $n$ isn’t the largest integer. Contradiction! ■-ish

```cpp
bool willAccept(string program, string input) {
    /* ...some implementation... */
}

int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) {
        reject();
    } else {
        accept();
    }
}
```
**Theorem:** $A_{TM} \notin R$. 
Theorem: \( A_{\text{TM}} \notin R \).

Proof: By contradiction; assume that \( A_{\text{TM}} \in R \).

```c
int main() {
    string me = mySource();
    string input = getInput();
    if (willAccept(me, input)) reject();
    else accept();
}
```

Choose any string \( w \) and trace through the execution of program \( P \) on input \( w \), focusing on the answer given back by the \( \text{willAccept} \) method. If \( \text{willAccept}(me, input) \) returns true, then \( P \) must accept its input \( w \). However, in this case \( P \) proceeds to reject its input \( w \). Otherwise, if \( \text{willAccept}(me, input) \) returns false, then \( P \) must not accept its input \( w \). However, in this case \( P \) proceeds to accept its input \( w \). In both cases we reach a contradiction, so our assumption must have been wrong. Therefore, \( A_{\text{TM}} \notin R \). ■
**Theorem:** $A_{TM} \notin R$.

**Proof:** By contradiction; assume that $A_{TM} \in R$. Then there is some decider $D$ for $A_{TM}$, which we can represent in software as a method `willAccept` that takes as input the source code of a program and an input, then returns true if the program accepts the input and false otherwise.

```plaintext
int main() {
    string me = mySource();
    string input = getInput();
    if (willAccept(me, input)) reject();
    else accept();
}
```

Choose any string $w$ and trace through the execution of program $P$ on input $w$, focusing on the answer given back by the `willAccept` method. If `willAccept(me, input)` returns true, then $P$ must accept its input $w$. Otherwise, if `willAccept(me, input)` returns false, then $P$ must not accept its input $w$.

In both cases we reach a contradiction, so our assumption must have been wrong. Therefore, $A_{TM} \notin R$. ■
Theorem: $A_{TM} \notin R$.

Proof: By contradiction; assume that $A_{TM} \in R$. Then there is some decider $D$ for $A_{TM}$, which we can represent in software as a method `willAccept` that takes as input the source code of a program and an input, then returns true if the program accepts the input and false otherwise.

Given this, we could then construct this program $P$:

```java
int main() {
    string me = mySource();
    string input = getInput();
    if (willAccept(me, input)) reject();
    else accept();
}
```
**Theorem:** \( A_{TM} \notin \mathbb{R} \).

**Proof:** By contradiction; assume that \( A_{TM} \in \mathbb{R} \). Then there is some decider \( D \) for \( A_{TM} \), which we can represent in software as a method \( \text{willAccept} \) that takes as input the source code of a program and an input, then returns true if the program accepts the input and false otherwise.

Given this, we could then construct this program \( P \):

```plaintext
int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) reject();
    else accept();
}
```

Choose any string \( w \) and trace through the execution of program \( P \) on input \( w \), focusing on the answer given back by the \( \text{willAccept} \) method.
**Theorem:** $A_{TM} \notin R$.

**Proof:** By contradiction; assume that $A_{TM} \in R$. Then there is some decider $D$ for $A_{TM}$, which we can represent in software as a method `willAccept` that takes as input the source code of a program and an input, then returns true if the program accepts the input and false otherwise.

Given this, we could then construct this program $P$:

```plaintext
int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) reject();
    else accept();
}
```

Choose any string $w$ and trace through the execution of program $P$ on input $w$, focusing on the answer given back by the `willAccept` method. If `willAccept(me, input)` returns true, then $P$ must accept its input $w$. Otherwise, if `willAccept(me, input)` returns false, then $P$ must not accept its input $w$.

In both cases we reach a contradiction, so our assumption must have been wrong. Therefore, $A_{TM} \notin R$. ■
**Theorem:** $A_{TM} \notin R$.

**Proof:** By contradiction; assume that $A_{TM} \in R$. Then there is some decider $D$ for $A_{TM}$, which we can represent in software as a method `willAccept` that takes as input the source code of a program and an input, then returns true if the program accepts the input and false otherwise.

Given this, we could then construct this program $P$:

```cpp
int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) reject();
    else accept();
}
```

Choose any string $w$ and trace through the execution of program $P$ on input $w$, focusing on the answer given back by the `willAccept` method. If `willAccept(me, input)` returns true, then $P$ must accept its input $w$. However, in this case $P$ proceeds to reject its input $w$. However, in this case $P$ proceeds to accept its input $w$. In both cases we reach a contradiction, so our assumption must have been wrong. Therefore, $A_{TM} \notin R$. ■
**Theorem:** $A_{TM} \notin R$.

**Proof:** By contradiction; assume that $A_{TM} \in R$. Then there is some decider $D$ for $A_{TM}$, which we can represent in software as a method `willAccept` that takes as input the source code of a program and an input, then returns true if the program accepts the input and false otherwise.

Given this, we could then construct this program $P$:

```java
int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) reject();
    else accept();
}
```

Choose any string $w$ and trace through the execution of program $P$ on input $w$, focusing on the answer given back by the `willAccept` method. If `willAccept(me, input)` returns true, then $P$ must accept its input $w$. However, in this case $P$ proceeds to reject its input $w$. Otherwise, if `willAccept(me, input)` returns false, then $P$ must not accept its input $w$. In both cases we reach a contradiction, so our assumption must have been wrong. Therefore, $A_{TM} \notin R$. ■
Theorem: $A_{TM} \notin R$.

Proof: By contradiction; assume that $A_{TM} \in R$. Then there is some decider $D$ for $A_{TM}$, which we can represent in software as a method `willAccept` that takes as input the source code of a program and an input, then returns true if the program accepts the input and false otherwise.

Given this, we could then construct this program $P$:

```java
int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) reject();
    else accept();
}
```

Choose any string $w$ and trace through the execution of program $P$ on input $w$, focusing on the answer given back by the `willAccept` method. If `willAccept(me, input)` returns true, then $P$ must accept its input $w$. However, in this case $P$ proceeds to reject its input $w$. Otherwise, if `willAccept(me, input)` returns false, then $P$ must not accept its input $w$. However, in this case $P$ proceeds to accept its input $w$. 

Therefore, $A_{TM} \notin R$. ■
**Theorem:** \( A_{TM} \notin R \).

**Proof:** By contradiction; assume that \( A_{TM} \in R \). Then there is some decider \( D \) for \( A_{TM} \), which we can represent in software as a method `willAccept` that takes as input the source code of a program and an input, then returns true if the program accepts the input and false otherwise.

Given this, we could then construct this program \( P \):

```java
int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) reject();
    else accept();
}
```

Choose any string \( w \) and trace through the execution of program \( P \) on input \( w \), focusing on the answer given back by the `willAccept` method. If `willAccept(me, input)` returns true, then \( P \) must accept its input \( w \). However, in this case \( P \) proceeds to reject its input \( w \). Otherwise, if `willAccept(me, input)` returns false, then \( P \) must not accept its input \( w \). However, in this case \( P \) proceeds to accept its input \( w \).

In both cases we reach a contradiction, so our assumption must have been wrong.
**Theorem:** $A_{TM} \notin \mathbb{R}$.

**Proof:** By contradiction; assume that $A_{TM} \in \mathbb{R}$. Then there is some decider $D$ for $A_{TM}$, which we can represent in software as a method `willAccept` that takes as input the source code of a program and an input, then returns true if the program accepts the input and false otherwise.

Given this, we could then construct this program $P$:

```java
int main() {
    string me = mySource();
    string input = getInput();
    if (willAccept(me, input)) reject();
    else accept();
}
```

Choose any string $w$ and trace through the execution of program $P$ on input $w$, focusing on the answer given back by the `willAccept` method. If `willAccept(me, input)` returns true, then $P$ must accept its input $w$. However, in this case $P$ proceeds to reject its input $w$. Otherwise, if `willAccept(me, input)` returns false, then $P$ must not accept its input $w$. However, in this case $P$ proceeds to accept its input $w$.

In both cases we reach a contradiction, so our assumption must have been wrong. Therefore, $A_{TM} \notin \mathbb{R}$. 

■
**Theorem:** \( A_{TM} \not\in R. \)

**Proof:** By contradiction; assume that \( A_{TM} \in R. \) Then there is some decider \( D \) for \( A_{TM} \), which we can represent in software as a method \( \text{willAccept} \) that takes as input the source code of a program and an input, then returns true if the program accepts the input and false otherwise.

Given this, we could then construct this program \( P \):

```c
int main() {
    string me = mySource();
    string input = getInput();
    if (willAccept(me, input)) reject();
    else accept();
}
```

Choose any string \( w \) and trace through the execution of program \( P \) on input \( w \), focusing on the answer given back by the \( \text{willAccept} \) method. If \( \text{willAccept}(me, input) \) returns true, then \( P \) must accept its input \( w \). However, in this case \( P \) proceeds to reject its input \( w \). Otherwise, if \( \text{willAccept}(me, input) \) returns false, then \( P \) must not accept its input \( w \). However, in this case \( P \) proceeds to accept its input \( w \).

In both cases we reach a contradiction, so our assumption must have been wrong. Therefore, \( A_{TM} \not\in R. \) ■
What Does This Mean?

• In one fell swoop, we've proven that
  • A decider for $A_{TM}$ is a self-defeating object.
  • $A_{TM}$ is *undecidable*; there is no general algorithm that can determine whether a TM will accept a string.
  • $R \neq \text{RE}$, because $A_{TM} \notin R$ but $A_{TM} \in \text{RE}$.
• What do these three statements really mean? As in, why should you care?
Next Time

• **Why All This Matters**
  • Important, practical, undecidable problems.

• **Intuiting RE**
  • What exactly is the class **RE** all about?

• **Verifiers**
  • A totally different perspective on problem solving.

• **Beyond RE**
  • Finding an impossible problem using very familiar techniques.