

Computability

Harry Sha

Fall 2026

,

Section 1

Turing machines

Turing machines

To study computation formally, we need a precise mathematical model.

We will use Turing machines, introduced by Alan Turing in 1936.

Mathematically modelling programs

[1...

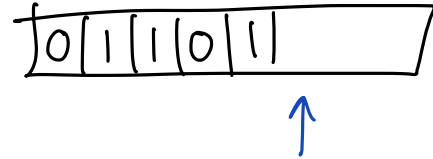
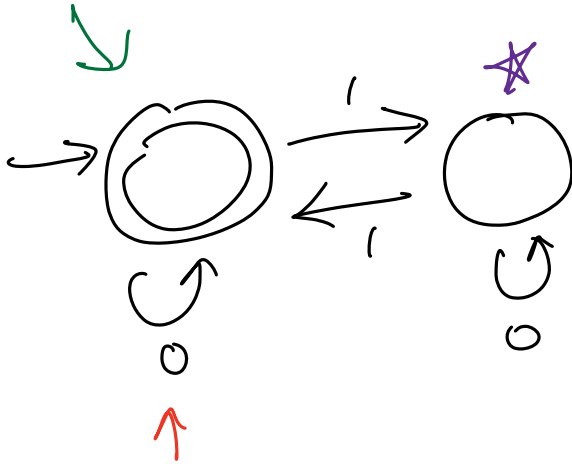
What properties should the model have?

- Source code, syntax rules ✓
- executing, properties of that ✓
- memory ○ → finite - , want unbound . ?
- should be able to express "if, for, while ... etc" → STRONG
- programs should be finite. ✓
want write instead of just read ...

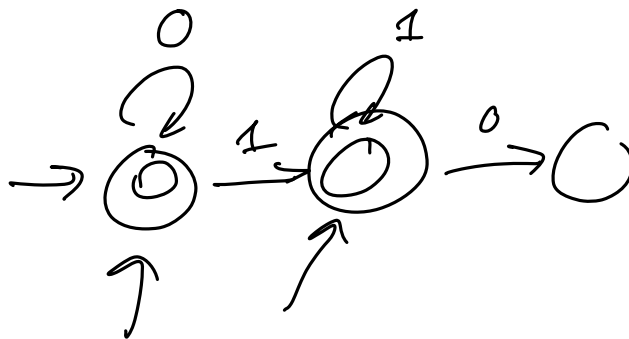
DFAs

Before introducing Turing machines, let's briefly review deterministic finite automata (DFAs).

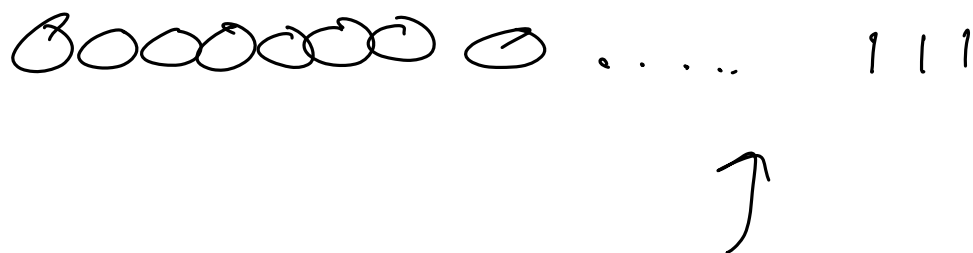
Example DFA: Parity = $\{w \in \{0,1\}^* : \# \text{ of } 1\text{'s in } w \text{ is even}\}$.



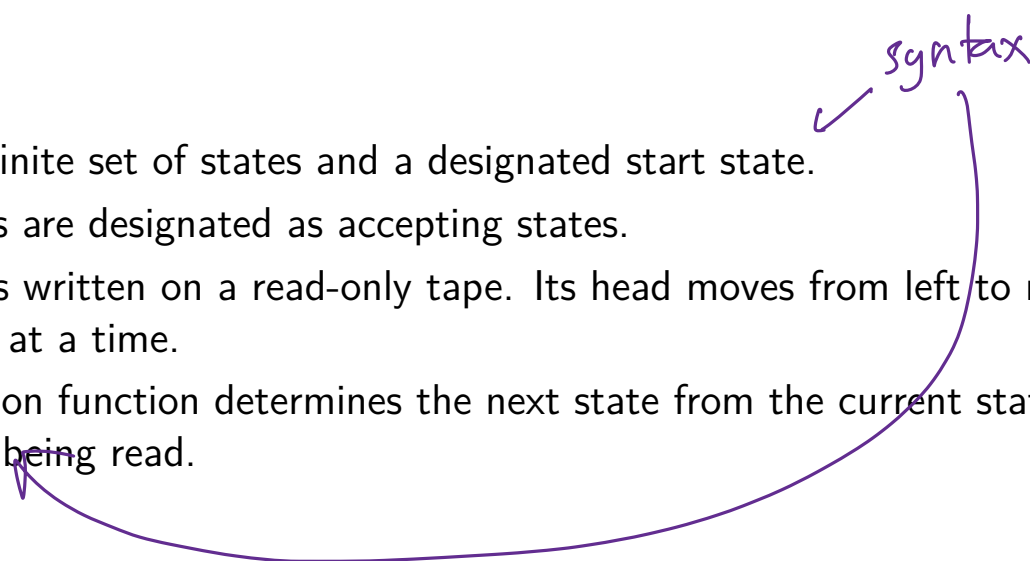
$0^* 1^*$



$0^n 1^n$



DFA summary

- There is a finite set of states and a designated start state.
 - Some states are designated as accepting states.
 - The input is written on a read-only tape. Its head moves from left to right, one symbol at a time.
 - The transition function determines the next state from the current state and the symbol being read.
- 

From DFAs to Turing machines

Turing machines are DFAs, with two differences.

- The tape is unbounded and the tape head can move in either direction.
- The machine can write symbols on the tape.

A Turing machine has designated accept and reject states. It halts as soon as it enters either one.

Examples Notebook

[Link to Notebook](#)

Example: Looping

Looping

For a Turing machine M and input string x , there are three possible outcomes:

- M eventually accepts;
- M eventually rejects;
- M loops forever.

Turing machines formally

Here is the formal definition from Sipser's textbook.

DEFINITION 3.3

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

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

Reading the formal definition

- All inputs come from Σ^* .
- Γ is the tape alphabet containing Σ , and any additional symbols that the TM might want to use.
- The transition function has signature

$$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}.$$

For example, $\delta(q, x) = (q', x', L)$ means: enter state q' , replace x with x' , and move the tape head left.

Encoding

Encoding is a big theme of this course.

Many results in this course can be understood as clever ways to encode mathematical objects as strings.

Let U be some set of objects that we care about. An encoding of the elements of U is a function $\text{Enc} : U \rightarrow \Sigma^*$.

Question: What property should Enc satisfy?

Injective

if not:

u_1
 u_2 $\searrow \swarrow$ x

Example: encoding pairs

Find an encoding $f : \Sigma^* \times \Sigma^* \rightarrow \Sigma^*$. For concreteness, take $\Sigma = \{0, 1\}$.

Proposal: $0 \rightarrow 00$, $1 \rightarrow 11$, 01 means “,”

example $(011, 1) \mapsto 00 \underline{11} \underline{11} \underline{01} \underline{11}$


The ASCII trick

mapping of 128 symbols to 8-bit strings.

essentially: can use as many symbols as you like
for any alphabet, by reading the string
in chunks.

Question

Can we do this trick with the alphabet $\{1\}$?

ascii trick

$$\mathcal{U} \mapsto \{0, 1\}^* \xrightarrow{f} \{1\}^*$$

idea $f(b) = 1^{\text{val}(b)}$

$\text{val}(b)$ is the number represented by b in binary.

Examples

$$\{ \forall n. n \geq 5, \dots \}$$


How can we encode $\mathbb{N}$, $\mathbb{Z}$, formulas, lists, dictionaries, graphs, Turing machines over the alphabet $\{0, 1\}$?

↑
" "
[,

Notation

We use angle brackets to denote the encoding of objects as strings.

- For an object A , $\langle A \rangle$ is its encoding.
- For tuples, instead of writing $\langle (A, B, C) \rangle$, we just write $\langle A, B, C \rangle$

Question

How many TMs are there?

→ countably many!

Summary

- TMs are a model of computation similar to DFAs, except for the fact that they can also write to the tape, and move the tape head in either direction.
- On a particular input, a TM can either accept, reject or loop.
- We can encode essentially any finite object as a string using the ASCII trick.

Section 2

Recognizable, Decidable, Computable

Recognizable

A set $L \subseteq \Sigma^*$ is called *recognizable* if there exists a TM M that satisfies the following. For all $x \in \Sigma^*$,

$$x \in L \iff M \text{ accepts } x$$

Decidable

A set $L \subseteq \Sigma^*$ is called *decidable* if there exists a TM M that satisfies the following. For all $x \in \Sigma^*$,

$$\begin{array}{c} x \in L \implies M \text{ accepts } x \\ \wedge \\ x \notin L \implies M \text{ rejects } x \end{array}$$

Question

What's the difference between recognizability and decidability?

on no-instances ($x \notin L$), recognizability allows looping, decidability doesn't.

Some notation

For a TM M , let $\mathcal{A}(M)$ be the set of strings that M accepts.

I.e.,

$$\mathcal{A}(M) = \{w \in \Sigma^* : M \text{ accepts } w\}.$$

Note. Sipser calls $\mathcal{A}(M)$ the “language” of M , and uses $\mathcal{L}$. We’re going to use language to mean something else later in the course.

With this notation, a set L is recognizable if and only if there exists a TM M such that $\mathcal{A}(M) = L$, and it is decidable if there exists a TM M such that $\mathcal{A}(M) = L$ and M halts on all inputs.

$$\begin{aligned} \text{TR} &\subseteq \mathcal{P}(\Sigma^*) \\ \text{D} &\subseteq \mathcal{P}(\Sigma^*) \end{aligned}$$

Define

$$\underline{\text{TR}} = \{ \underline{L} \subseteq \Sigma^* : L \text{ is recognizable} \},$$

and

$$\underline{\text{D}} = \{ L \subseteq \Sigma^* : L \text{ is decidable} \}.$$

Question

How large are **TR** and **D**?

→ Countable!
b/c countably many TMs.

So, again $\text{TR} \neq \mathcal{P}(\Sigma^*)$
↑ ↑

Exercise (in class)

$$A(M) = L$$

$$A(M) = L'$$

$$\Rightarrow L = L'$$

Show that $\mathbf{D} \subseteq \mathbf{TR}$.



Exercise (in class)

Suppose $\overset{M}{L} \in \mathbf{TR}$ and $\bar{L} \in \mathbf{TR}$. Show that $L \in \mathbf{D}$.

→ run both, one computation is guaranteed to halt.

$\overset{M'}{\Sigma^* \setminus L}$

x might ✓
on input x
simulate M on x .
if accept: accept
simulate M' on x .
if accept: reject

parallel is important!

Computing functions

A function $f : \Sigma^* \rightarrow \Sigma^*$ is computable if some machine M , on every input $w \in \Sigma^*$, halts with $f(w)$ written on its tape.

Example: append one bit

The function $f(w) = w1$ is computable.

The Church–Turing thesis

Computability was meaningful before electronic computers existed. For example, one could ask for an effective procedure to compute a greatest common divisor or the n th Fibonacci number.

The Church–Turing thesis says that Turing machines capture the intuitive notion of effective computation:

A function is effectively computable if and only if a Turing machine computes it.

Is the Church–Turing thesis convincing?

So far it has held up.

We'll give some arguments in support of the thesis in tutorial and exercises.

- In tutorial, we will see how ordinary Turing machines simulate apparently stronger variants, including machines with multiple tapes and machines that use randomness.
- In the exercises, you will implement TMs that simulate conditionals, for loops and while loops.

Fact

{ Turing machines can simulate programming languages. In particular, any Python program can be compiled into a Turing machine.

Consequently, to show that something is computable/decidable/recognizable, it is enough to give pseudocode.

A universal TM

There is a Turing machine U that does the following

$U(M, w)$:

- Simulate M on w

Essentially, U is an interpreter. It's called the *Universal Turing Machine*.

Summary

- Definitions of recognizability, decidability, computability.
- Church–Turing thesis: Anything that can be computed can be computed by a TM.
- There's a TM, U , that takes as input the source code of a TM, and another string, and simulates the TM on the string.

Announcements:

- Tutorial: This week w/ Devansh
- OH : by appointment
- TT1 (next Friday during tutorial)
- Feedback . open on Canvasmark .

Section 3

Undecidability and reductions

Undecidability

Earlier, we showed using Cantor's Theorem that some decision problems are undecidable.

We'll now show some other natural problems are undecidable.

A_{TM}

$\langle \rangle$: means encoding.

The acceptance problem is

$A_{\text{TM}} \in \text{TR}$.

$$\underline{A_{\text{TM}}} = \{ \langle M, w \rangle : M \text{ accepts } w \}.$$

Notice that $A_{\text{TM}} = \underline{\mathcal{A}(U)}$, where U is the universal TM.

A_{TM} is undecidable

$$A_{TM} = \{ \langle M, w \rangle : M \text{ accepts } w \}$$

All strings represent some TM. For a string x , let M_x be the TM encoded by x . By contradiction, suppose A_{TM} is decidable. So, let's say there's a function will-accept(M, w) that accepts if M accepts w , and rejects if M does not accept w .

Then, we can write the following program:

$D(x)$:
if will-accept(M_x, x)
 reject
else : accept.

D always halts.
if its encoding is $\langle D \rangle$. Does D accept $\langle D \rangle$?

$$\mu_i = \mu_{w_i}$$

	w_0	w_1	w_2	...
μ_1	✓ ✗	✗	✓	✗
μ_2	✓	✗ ✓	✗	
μ_3	✗	✗	✗ ✓	
⋮				

● will accept

⊙ D.

Exercise (in class)

StA to present!

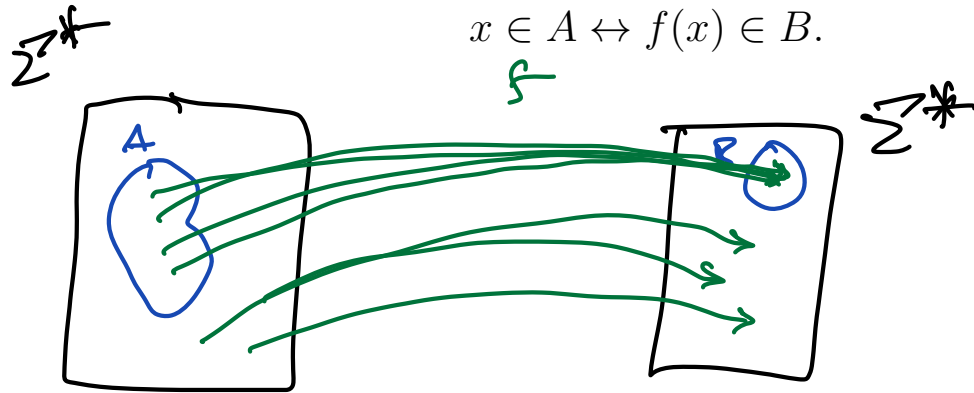
Define

$$\text{HALT}_{\text{TM}} = \{\langle M, w \rangle : M \text{ halts on } w\}.$$

Using a similar argument, show that HALT_{TM} is undecidable.

Mapping reductions

A language A mapping-reduces to B , written $A \leq_m B$, if there is a total computable function f such that



Exercise (in class)

Prove the following claim: If A is undecidable and $A \leq_m B$, then B is undecidable.

$$A \leq_m B : \exists f : \forall x \in \Sigma^* . \\ x \in A \Leftrightarrow f(x) \in B .$$

The contrapositive is also useful

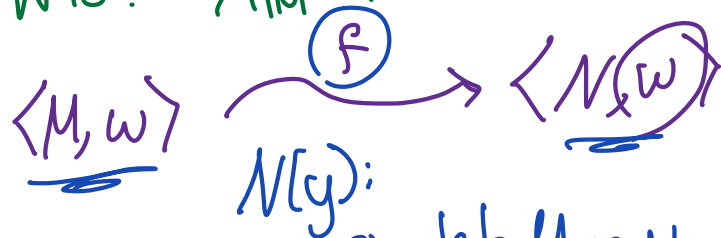
Equivalently, if $A \leq_m B$ and B is decidable, then A is decidable.

Example

$$\rightarrow \text{HALT}_{\text{TM}} = \{ \langle M, w \rangle : M \text{ halts on } w \}.$$

HALT_{TM} is undecidable via reduction from A_{TM} .

WTS. $A_{\text{TM}} \leq_m \text{HALT}_{\text{TM}}$



$N(y)$:

simulate M on w .

if M accepts w : halt

if M rejects w : loop

*** if M loops: N also loops.**

$$\langle M, w \rangle \in A_{\text{TM}} \Leftrightarrow \langle N, x \rangle \in \text{HALT}_{\text{TM}}$$

want: $M \text{ accepts } w \Leftrightarrow N \text{ halts on } w$

$N(w)$. ✓

✓

✓

Exercise (in class)

Let $\text{HALT}_\epsilon = \{\langle M \rangle : M \text{ halts on } \epsilon\}$. Show that HALT_ϵ is undecidable.

Question

We have shown that A_{TM} and $HALT_{TM}$ are undecidable. However, both languages are recognizable.

Can you find a language that is not recognizable?

$\overline{A_{TM}}$ $\overline{HALT_{TM}}$

Exercise (later)

Can you find a language A such that neither A nor $\overline{A}$ is recognizable?

Summary

- A_{TM} and HALT_{TM} are undecidable. The proof is by diagonalization and is similar to the proof of Cantor's Theorem.
- Reductions are a way to transform one problem into another. They can be used to transfer decidability/undecidability results from one set to another.

Section 4

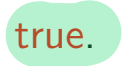
Incompleteness Via Reductions

An Informal Proof of the Incompleteness Theorem

The following is essentially a full proof of the incompleteness theorem with the logic parts handwaved.

Incompleteness (Informal)

No formal system can simultaneously have all four of the following properties:

- 
- 
- 
- 
- 
1. **Completeness.** Every **true** mathematical statement is provable.
 2. **Soundness.** Every provable statement is **true**.
 3. **Verifiable proofs.** An algorithm can decide whether a proposed **proof** is valid.
 4. **Expressivity.** The system can describe enough of mathematics, including the operations of a Turing machine.

Setup

Let

$$\Sigma = \{\vee, \wedge, \rightarrow, \neg, \forall, \exists, x, f, R, 0, 1, 2, 3, 4, \dots, 9\}.$$

Let $T \subseteq \Sigma^*$ be the set of theorems, i.e.

$$T = \{\langle \varphi \rangle : \varphi \text{ has a proof}\}$$

For a contradiction, suppose the system has all four properties.

T is decidable

Idea 1: Search all proofs

On input φ :

- For each string $w \in \Sigma^*$
 - Check (using property 3) if w is a proof of φ .
 - If it is, accept

Question. What's the issue? What's the fix?

→ look for
proof of $\varphi, \neg\varphi$ "in parallel"

T is decidable, take 2

if no soundness, then maybe both φ , and $\neg\varphi$ have proofs.

On input φ :

for each string $w \in \Sigma^*$:

if w is a proof of φ , accept

if w is a proof of $\neg\varphi$, reject.



by completeness, either φ or $\neg\varphi$ has a proof.

T is undecidable

$$\text{HALT}_\varepsilon = \{ \langle M \rangle : M \text{ halts on } \varepsilon \}.$$

We now derive a contradiction by showing T is undecidable. We do this by reducing HALT_ε to T .

By the expressivity property, for each TM $\underline{M}$, we can find a formula φ_M that says $\underline{M}$ halts (on an empty input).

- If M halts... $\Rightarrow \varphi_M$ is true $\Rightarrow \varphi_M$ has a proof $\Rightarrow \varphi_M \in T$.
- If M loops... $\Rightarrow \varphi_M$ is not true $\Rightarrow \neg \varphi_M$ has a proof $\Rightarrow \neg \varphi_M \in T$.

This shows $\text{HALT}_\varepsilon \leq_m T$. ($M \mapsto \varphi_M$).

Completeness: every true statement has a proof.



either φ is true
or $\neg \varphi$ is true

"for all φ , either φ or $\neg \varphi$ has a proof".

Missing ingredients

To make the proof rigorous, we still need a precise notion of truth and a formal account of what it means for φ_M “to say that M halts”. The ability to construct such a formula is the expressivity property we glossed over earlier.

Section 5

The Recursion Theorem and Self-Reference

Stefan & Aaron : HALT_{TM} is not decidable (by diagonalization).

Self-reference

I asked you a while ago whether or not the following sentence is true.

“This sentence is false.”

An important feature of this paradox is self-reference. I.e. the word “this”. Suppose I tried to formalize this sentence in logic...

Self-reference

Can a Turing machine obtain its own encoding? I.e. can I do

```
r = get_source()
```

Warm-up

Let's begin with a program that prints its own source code.

First attempt

```
print("438")
```

This program prints 438, but its source code is `print("438")`. Can we modify it so that it prints its own source code?

Try it yourself

The goal is to make a program print its own source code exactly.

The program may take any form, but it may not read its source file.

You can use [this online Python interpreter](#) or write a script locally.

In Python, `repr(s)` returns a quoted representation of the string `s`. I.e. `print(repr('hi'))` prints `'hi'`.

Hint

Print the following twice, the second time in quotes
"Print the following twice, the second time in quotes"

Let's try together

Generalizing

In general, a program may need to do more than print its own source code.

The recursion theorem

Theorem (Recursion theorem)

Let T be a Turing machine whose input is $\langle x, w \rangle$. There is a Turing machine R such that, for every input w ,

$$R(w) \simeq T(\langle R, w \rangle).$$

Here $\simeq$ means the computations have the same observable behavior: both loop, or both halt with the same outcome and output.

Question. How does this justify the shorthand `get_source()`?

get_source()

$f(x) :$
 $\text{src} = \text{get_source}()$
 $\vdots$

$T(\text{src}, x) :$
 $\vdots$

refactor

From recursion theorem, obtain
 R st. $R(x) \cong T(\langle R \rangle, x)$
 $\cong f(x)$

Encoding of TMs

Pick an encoding of TMs such that every string corresponds to some TM. For strings with 'invalid' syntax, simply map them to a TM that rejects everything.

Thus every string x denotes some Turing machine, which we call M_x .

q

Define a function $q : \Sigma^* \rightarrow \Sigma^*$ by

$$q(x) = \langle Q_x \rangle ,$$

where Q_x is the following machine.

Q_x on input w :

- Simulate M_x on $\langle x, w \rangle$.

This supplies M_x with its own encoding as a parameter.

Question

What does $Q_{\langle T \rangle}$ do? Why doesn't $Q_{\langle T \rangle}$ complete the proof?

Intermediate machine B

Construct a machine B as follows.

$B(x, w) :$

- Simulate T on $\langle q(x), w \rangle$.

Choose $\underline{R = Q_{\langle B \rangle}}$

Note $\langle R \rangle = \langle Q_{\langle B \rangle} \rangle = q(\langle B \rangle)$.

$$\begin{aligned} R(w) &= Q_{\langle B \rangle}(w) \\ &= B(\langle B \rangle, w) \\ &= T(q(\langle B \rangle), w) \\ &= T(\langle R \rangle, w) \end{aligned}$$

write another
quine: that
works like this:

```
f():  
    s = get_source()  
    print(s).
```

↙ Recursion
thm.

get_source

From now on, we may use get_source() as shorthand for the recursion theorem construction.

Section 6

Applications of Self-Reference

A_{TM} is undecidable by self-reference

Suppose H decides A_{TM} . Construct R as follows.

$R(w) :$

- Set $r := \text{get_source}()$.
- If H accepts $\langle r, w \rangle$, reject, otherwise, accept.

Exercise (in class)

$\{ \langle M, w \rangle : M \text{ halts} \}$

Show HALT_{TM} is undecidable by self-reference

by contradiction, assume HALT_{TM} is decidable, so,
 $\exists$ a function $\text{will_halt}(M, w)$. Then, construct D as follows

$D(x)$:

$me = \text{getSource}()$

if $\text{will_halt}(me, x)$:

loop ✓

else · accept

what does $D(\epsilon)$

if D halts on $\epsilon \Rightarrow \text{loop} \Rightarrow \epsilon$

if D loop on $\epsilon \Rightarrow \text{accept} \Rightarrow \epsilon$

Fixed-point theorem

Theorem (Fixed-point theorem)

Let $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ be total and computable. ~~There is a machine encoding e such that, for every w ,~~

$$\del{M_e(w) \sim M_{f(e)}(w)}.$$

Then $\exists e \in \{0, 1\}^*$ st. $M_e \equiv M_{f(e)}$

Proof

$R(w) :$

- Set $e := \text{get_source}()$
- Compute $y := f(e)$
- Simulate M_y on w .

Since e encodes R , we have $M_e(w) = R(w) \simeq M_{f(e)}(w)$.

b/c $e = \langle R \rangle$
 $M_e = R$



b/c trace the
code!



Summary

- `get_source()` is justified by The Recursion Theorem.
- Self-reference is another way to prove undecidability.
- The fixed-point theorem can be proved using self-reference. The fixed-point theorem has some nice applications that we will see in the exercises.

Section 7

Incompleteness via Self-reference

A stronger version of the incompleteness theorem

The earlier argument assumed soundness. Here we replace that assumption with consistency.

- **Soundness**: every provable sentence is true.
- **Consistency**: no sentence φ and its negation $\neg\varphi$ are both provable.

Soundness implies consistency, but consistency does not imply soundness.

Incompleteness (stronger version)

Again, note that this is an informal statement (since we haven't yet developed the logical terminology and theory).

No formal system can simultaneously have all four properties:

- **Completeness:** for all φ either φ or $\neg\varphi$ is **provable**.
- **Consistency:** for all φ at most one of φ and $\neg\varphi$ is **provable**.
- **Verifiable proofs:** there is an algorithm that checks proposed proofs.
- **Expressivity:** the system represents TM computations.

Notice, though, that this time, the word 'truth' doesn't appear...

α_x

For each string x , construct a formula α_x such that

- if M_x accepts the empty string, then α_x is provable,
- if M_x rejects the empty string, then $\neg\alpha_x$ is provable,
- no condition is imposed when M_x loops on the empty string.

The ability to construct α_x is the required expressivity property.

The self-referential proof search

Construct the following machine.

R :

- $\langle R \rangle = \text{get_source}()$
- construct $\alpha_{\langle R \rangle}$.
- For each string p in standard string order:
 - if p is a proof of $\alpha_{\langle R \rangle}$, reject;
 - if p is a proof of $\neg \alpha_{\langle R \rangle}$, accept.

I.e. R searches for proofs that R will accept, and proofs that R will reject, and then does the opposite.

Can either sentence be proved?

Question. Can the system prove $\alpha_{\langle R \rangle}$? Can it prove $\neg \alpha_{\langle R \rangle}$?

Question

What happens when R runs on the empty string?

↪ loop!

Next up...

We'll formally study logic.

How are we going to study logic? With logic. Buckle up! Things are about to get meta :-).