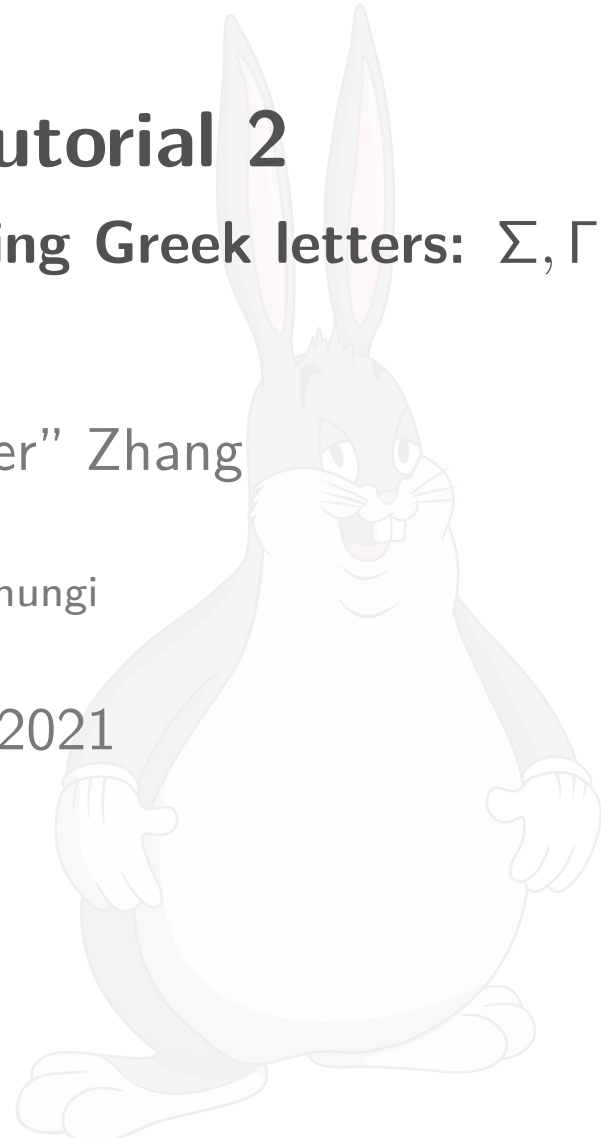
# CSC363H5 Tutorial 2

## I hope you can read the following Greek letters: $\Sigma, \Gamma, \delta$

Paul "sushi_enjoyer" Zhang

University of Chungi

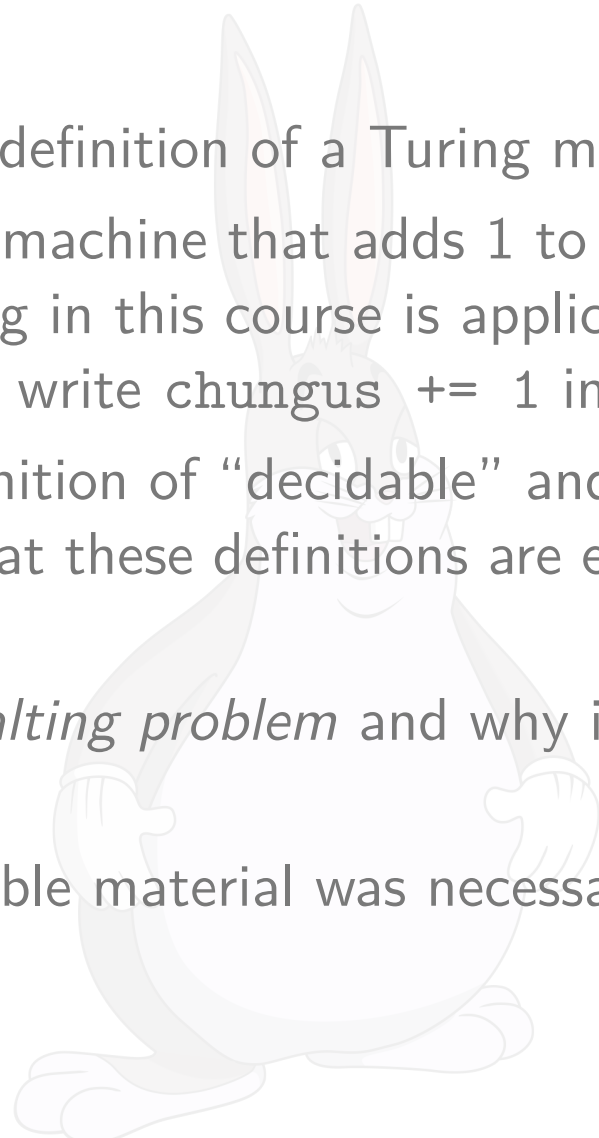
January 18, 2021



# Learning objectives this tutorial

By the end of this tutorial, you should...

- Have an informal idea of the formal definition of a Turing machine.
- Be able to build a very basic Turing machine that adds 1 to a number. Then question why anything in this course is applicable to actual programming, as you can just write `chungus += 1` instead.
- Understand the Turing machine definition of "decidable" and "listable", and take without proof that these definitions are equivalent to the definitions in class.
- Have a brief understanding of the *halting problem* and why it is a listable but undecidable language.
- Question why all the G*del computable material was necessary.



# helo!

`helo_fish.jpg` is feeling sad today :((((


Why is she sad? Because life really do be like that.
Also she is sad because G*del computable is too much math.

Anyway, if you want to make `helo_fish.jpg` feel better, please respond to her greetings by saying "helo", and then post your credit card number(s) in the chat.

**Protip...**

If `helo_fish.jpg` does not appear in your dreams, please find a therapist who can perform Jungian analysis for you.

# sad `helo_fish.jpg` wants you to learn about Turing machines.

But what really is a Turing machine?

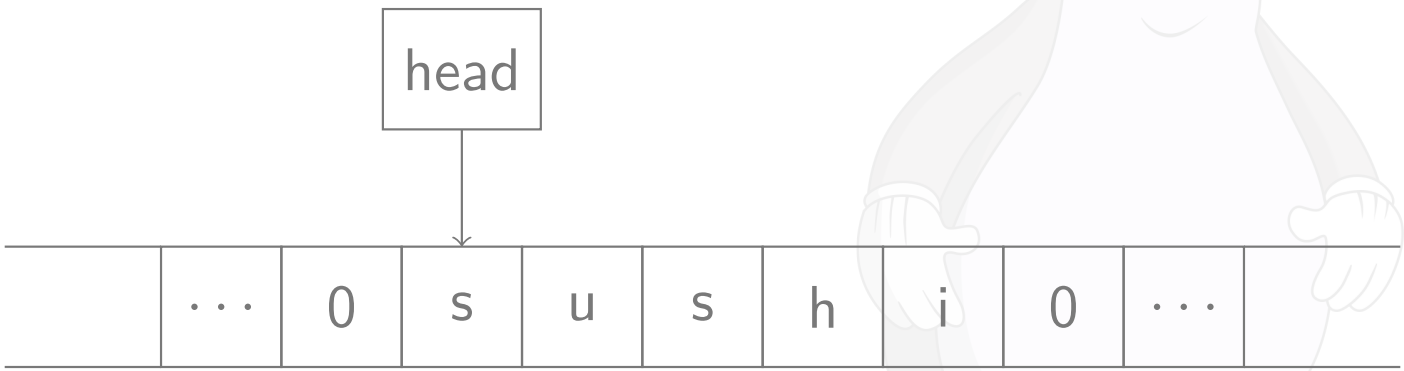

I've tried to introduce it on an informal level last tutorial. Here's a more formal definition, so you can actually build your own Turing machines.

# What *really* is a Turing machine?

Recall: a Turing machine has a set of states (kinda like a DFA).
It reads in a symbol from the tape (at the current read-write head), consults the current state, which gives three things:

- A symbol to write back to the tape.
- A new state to transition to.
- A direction to move the read-write head.

The Turing machine takes in an input string that is prewritten on a tape, and the read-write head pointing to the first symbol in the string.



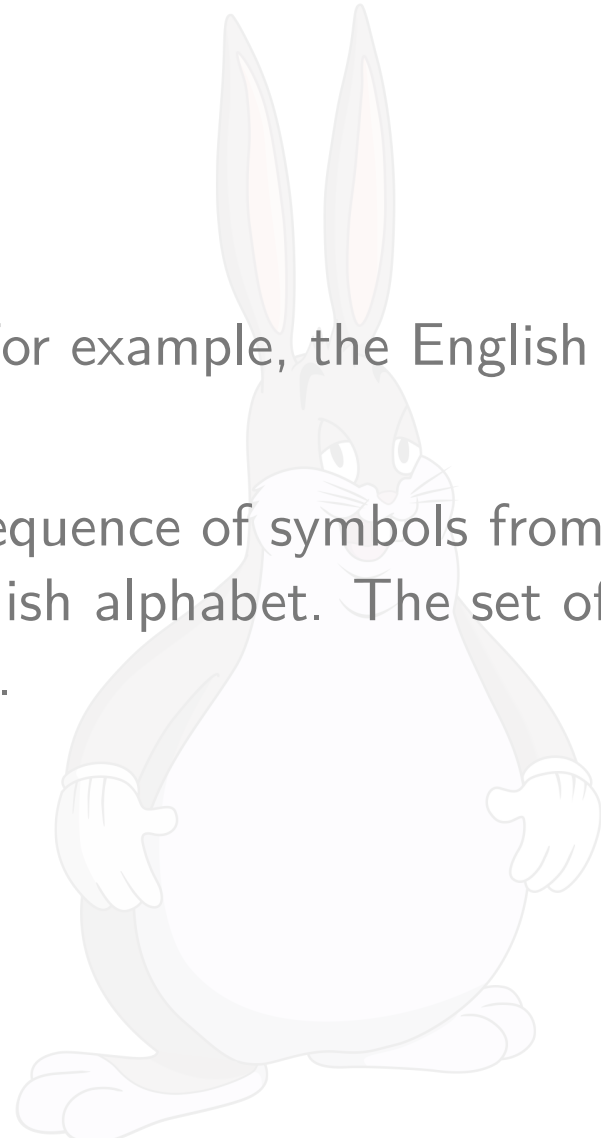

The output would just be whatever string is left on the tape at the end of execution.

# What *really* is a Turing machine?

Definitions (possibly from CSC236):

An **alphabet** is a *finite* set of symbols. For example, the English alphabet is $\{a, b, \ldots, z\}$.

A **string** over an alphabet $\Sigma$ is a *finite* sequence of symbols from $\Sigma$. For example, "sushi" is a string over the English alphabet. The set of all strings over an alphabet $\Sigma$ is denoted $\Sigma^*$.



# Formal definition

A *Turing machine* $M$ is an tuple of the form (don't blink):

$$M = (Q, \Sigma, \Gamma, \delta, q_s, F, 0).$$

Here's what each component means:

- $Q$ is what we referred to as the **set of states**. It's just an arbitrary (but *finite*) set, so we could have $Q = \{$🐟, 🍚, 🍣$\}$ if we wanted.
- $\Sigma$ is the **input alphabet**. The Turing machine receives input strings from $\Sigma^*$.
- $\Gamma$ is the **tape alphabet**. This represents the set of symbols that can be read and written to the tape. This satisfies $\Sigma \subseteq \Gamma$, since we better be able to read input strings.
- $\delta$ is a weird function which will be explained later.

# Formal definition

$$M = (Q, \Sigma, \Gamma, \delta, q_s, F, 0).$$

- $q_s$ is the **starting state**, and satisfies $q_s \in Q$.
- $F$ is the set of **accepting states**. This satisfies $F \subseteq Q$. When we reach an accepting state, we halt and accept.
- $0 \in \Gamma$ is the **blank symbol**. The tape is prewritten with the input string on it, and the blank symbol everywhere else.

# Transition function

$$M = (Q, \Sigma, \Gamma, \delta, q_s, F, 0).$$

$\delta : (Q \times \Gamma) \to (Q \times \Gamma \times \{L, R\})$ is the **transition function**.

Its domain is $(Q \times \Gamma)$. The idea is that $\delta$ takes the current state and the symbol we read as input.
Its codomain is $(Q \times \Gamma \times \{L, R\})$. This means $\delta$ outputs three things: the new state to transition to, the symbol to write back, and the direction to move the read-write head.

Usually you will see $\delta$ defined in what is called a **transition table**.

Again, the Turing machine takes in an input string prewritten on the tape with blank symbols everywhere. *If the Turing machine halts*, it outputs whatever is written on the tape at the time of halting.

# Big Chungus TM

In this example, we attempt to design a Turing machine over the tape alphabet $\{0, 🥕\}$, with input alphabet $\{🥕\}$ and blank symbol 0. (So the possible inputs are '$\epsilon$',[1] ' 🥕 ', "🥕🥕", "🥕🥕🥕", ...). The behaviour of this Turing machine is to output the first character of the input string, if it isn't the empty string.

We let $Q = \{\text{IGNORE}, \text{EAT}, \text{HALT}\}$, $\Sigma = \{🥕\}$, $\Gamma = \{0, 🥕\}$, $q_s = \text{IGNORE}$, $F = \{\text{HALT}\}$, $b = 0$. The transition table is given below:[2]

| State | 0 | 🥕 |
|---|---|---|
| IGNORE | (IGNORE, 0, R) | (EAT, 🥕, R) |
| EAT | (HALT, 0, R) | (EAT, 0, R) |

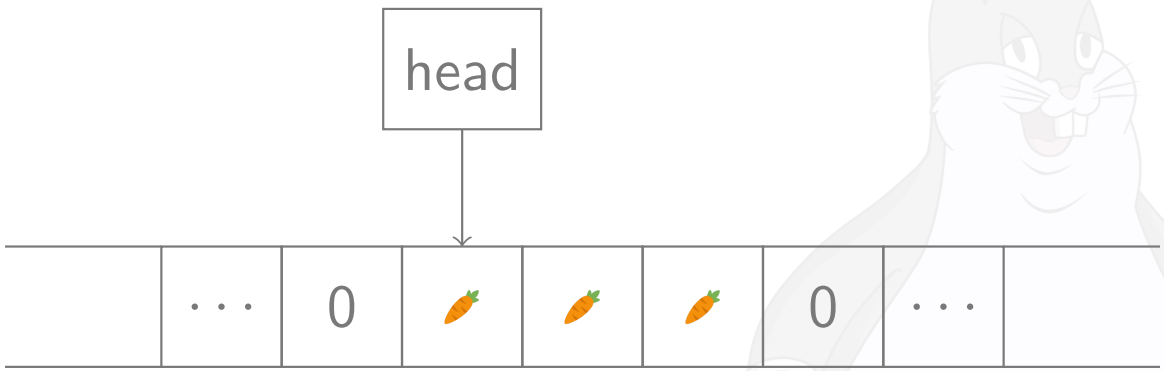
[1] $\epsilon$ is the empty string.

[2] Note that we don't need to define behaviour for halting states, since when we reach a halting state we stop.

# Big Chungus TM

| State | 0 | 🥕 |
|---|---|---|
| IGNORE | (IGNORE, 0, R) | (EAT, 🥕, R) |
| EAT | (HALT, 0, R) | (EAT, 0, R) |

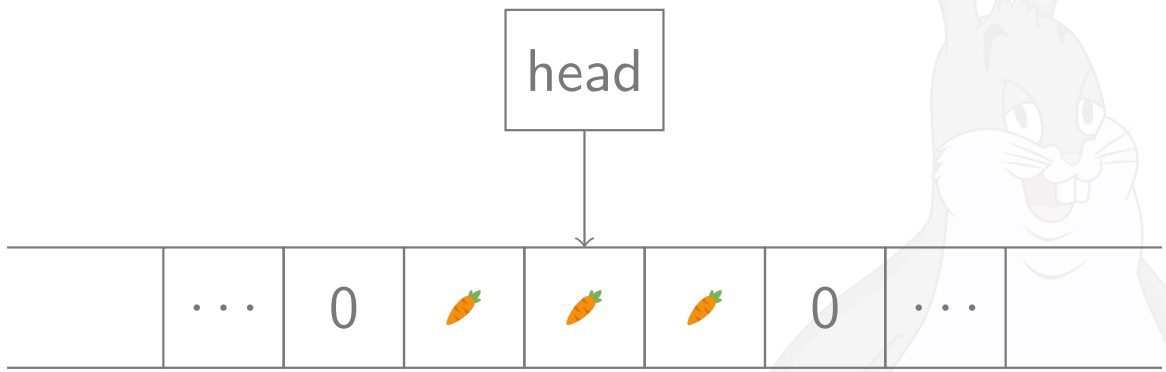
On input string "🥕 🥕 🥕":




Current state: IGNORE (since that's our initial state)
We read in 🥕. The transition table gives us (EAT, 🥕, $R$).

# Big Chungus TM

| State | 0 | 🥕 |
|---|---|---|
| IGNORE | (IGNORE, 0, R) | (EAT, 🥕, R) |
| EAT | (HALT, 0, R) | (EAT, 0, R) |

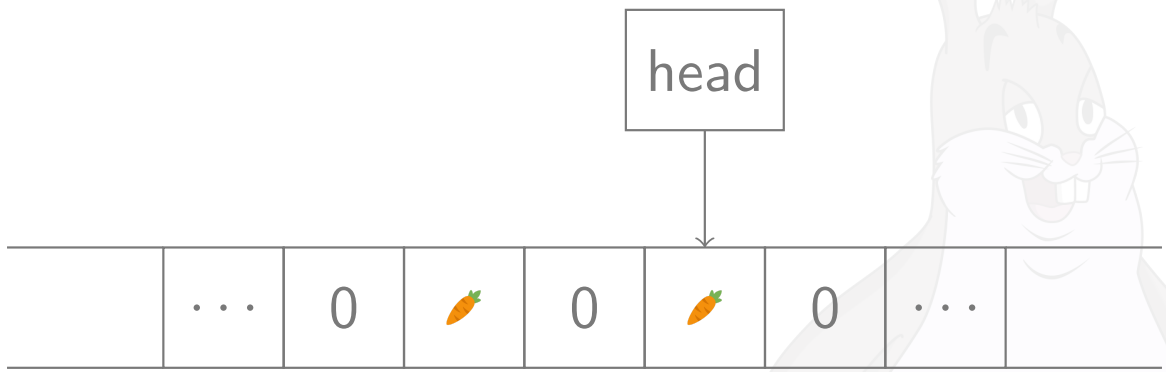



Current state: EAT

We read in 🥕. The transition table gives us $(\text{EAT}, 0, R)$.

# Big Chungus TM

| State | 0 | 🥕 |
|---|---|---|
| IGNORE | (IGNORE, 0, R) | (EAT, 🥕, R) |
| EAT | (HALT, 0, R) | (EAT, 0, R) |

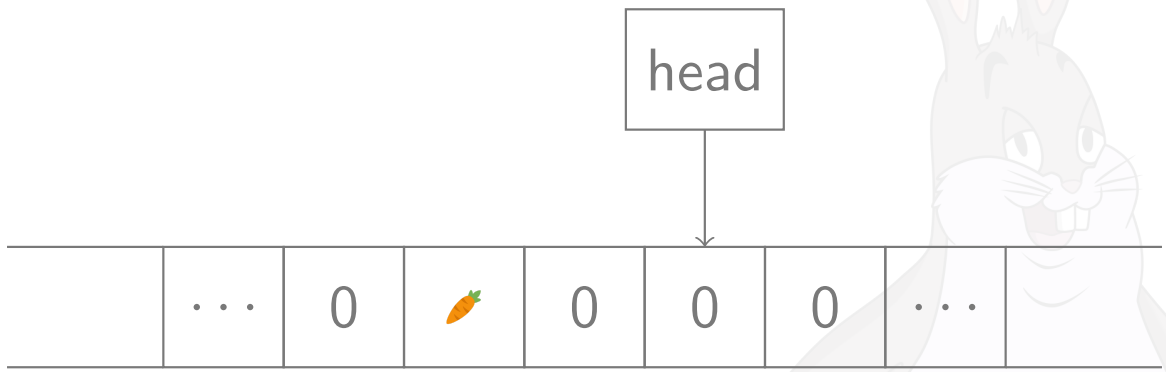



Current state: EAT

We read in 🥕. The transition table gives us $(\text{EAT}, 0, R)$.

# Big Chungus TM

| State | 0 | 🥕 |
|---|---|---|
| IGNORE | (IGNORE, 0, R) | (EAT, 🥕, R) |
| EAT | (HALT, 0, R) | (EAT, 0, R) |



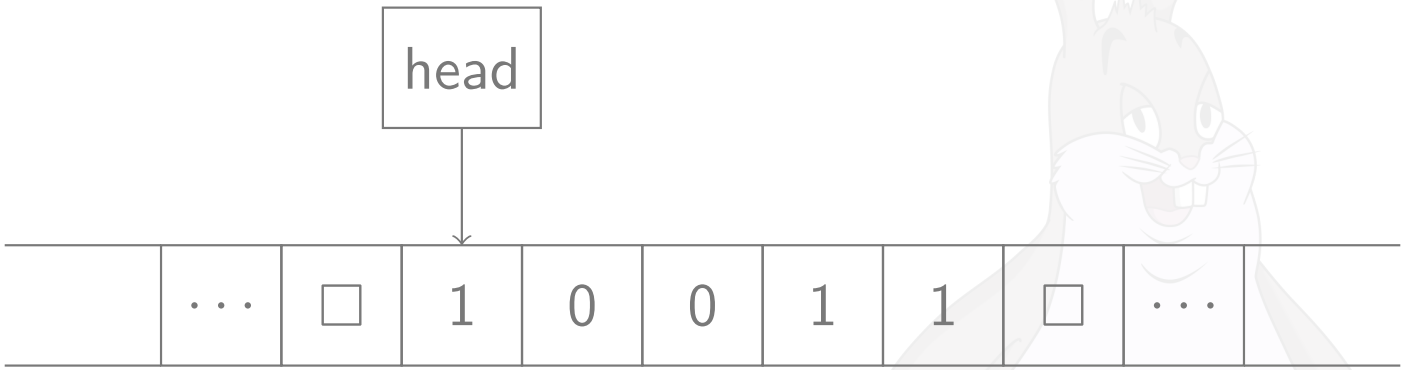

Current state: EAT

We read in 0. The transition table gives us $(\text{HALT}, 0, R)$. We halt and return whatever is left on the tape, which is the string ‘ 🥕 ’.

# Now it's your turn!

Let $\Sigma = \{0, 1\}$ be the input alphabet, $\Gamma = \{\square, 0, 1\}$ the tape alphabet, and $\square$ the blank symbol. Create a Turing machine that takes in a binary number and adds 1 to it.[3]

Example: Given input '10011', the Turing machine should return '10100'.




(GO TO NEXT SLIDE: putting this here as a reminder for myself. if i don't then someone pls remind me)

[3] You may assume the binary number starts with 1. If this is not the case, do whatever you like.

# Now it's your turn!

Oh yea, there's also a Turing machine simulator!
`https://turingmachinesimulator.com/`

```
name: add1
init: move_right
accept: halt

move_right, _
carry_over, _, <

move_right, 0
move_right, 0, >

move_right, 1
move_right, 1, >

carry_over, _
halt, 1, <
```

Ask me any questions on how to use it!

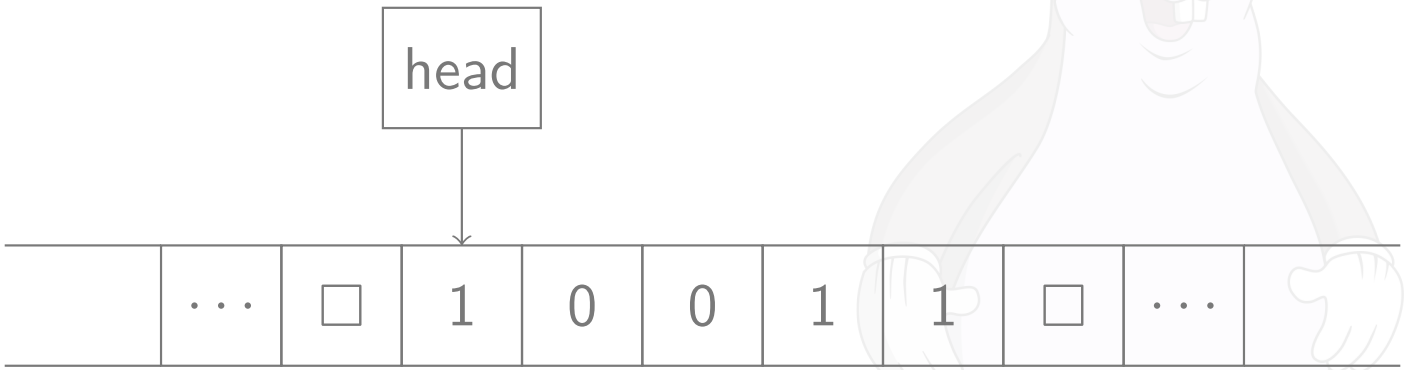
If you're done, try building a TM that takes an input string and concatenates it to itself (example: '10101' goes to '1010110101').

# Your answer may differ.

Our states will be $\{\text{MOVE}, \text{CARRY}, \text{HALT}\}$, with MOVE the initial state and HALT a final state.

| State | $\square$ | 0 | 1 |
|---|---|---|---|
| MOVE | (CARRY, $\square$, L) | (MOVE, 0, R) | (MOVE, 1, R) |
| CARRY | (HALT, 1, L) | (HALT, 1, L) | (CARRY, 0, L) |

Current state: MOVE

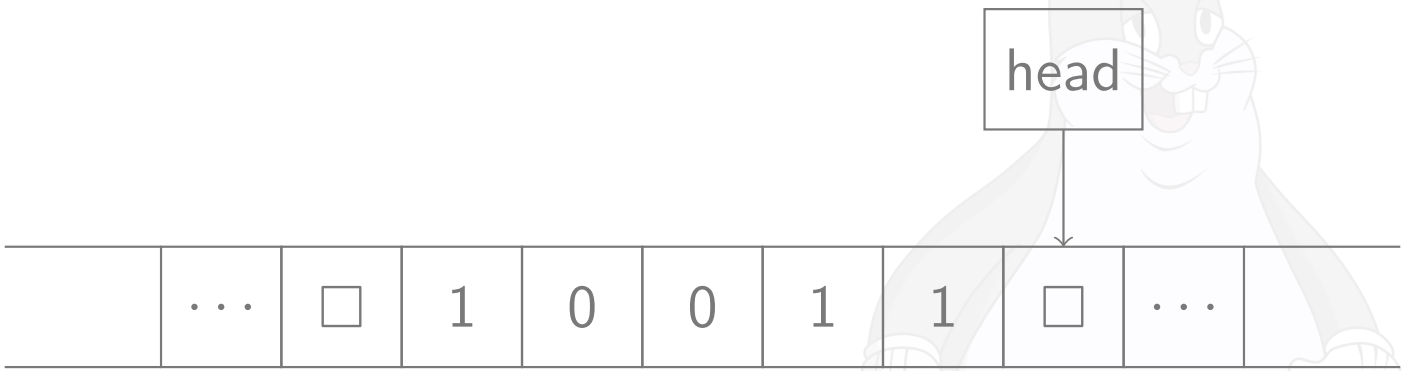



We keep going right until we encounter $\square$.

# Your answer may differ.

| State | □ | 0 | 1 |
|---|---|---|---|
| MOVE | (CARRY, □, L) | (MOVE, 0, R) | (MOVE, 1, R) |
| CARRY | (HALT, 1, L) | (HALT, 1, L) | (CARRY, 0, L) |

Current state: MOVE

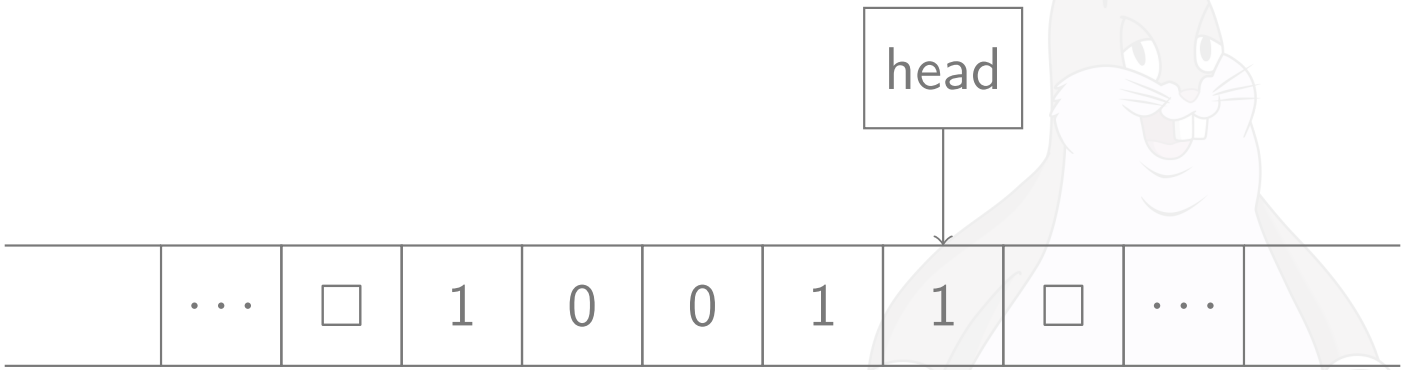



We've encountered □. Transition to state CARRY, write □, and go left.

# Your answer may differ.

| State | □ | 0 | 1 |
|---|---|---|---|
| MOVE | (CARRY, □, L) | (MOVE, 0, R) | (MOVE, 1, R) |
| CARRY | (HALT, 1, L) | (HALT, 1, L) | (CARRY, 0, L) |

Current state: CARRY

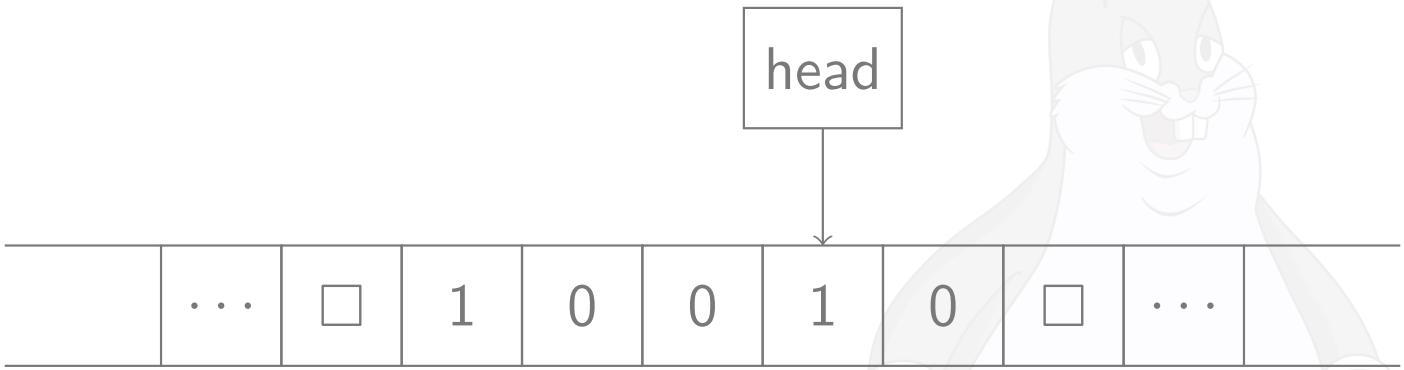



Transition to state CARRY (i mean, we're already in state CARRY), write 0, and move left.

# Your answer may differ.

| State | □ | 0 | 1 |
|---|---|---|---|
| MOVE | (CARRY, □, L) | (MOVE, 0, R) | (MOVE, 1, R) |
| CARRY | (HALT, 1, L) | (HALT, 1, L) | (CARRY, 0, L) |

Current state: CARRY

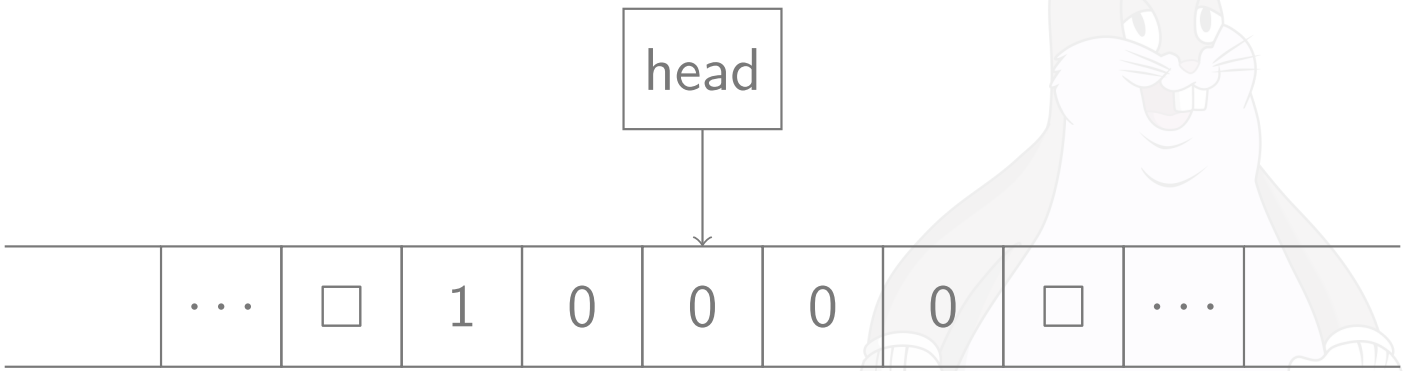



Transition to state CARRY (i mean, we're already in state CARRY), write 0, and move left.

# Your answer may differ.

| State | □ | 0 | 1 |
|---|---|---|---|
| MOVE | (CARRY, □, L) | (MOVE, 0, R) | (MOVE, 1, R) |
| CARRY | (HALT, 1, L) | (HALT, 1, L) | (CARRY, 0, L) |

Current state: CARRY

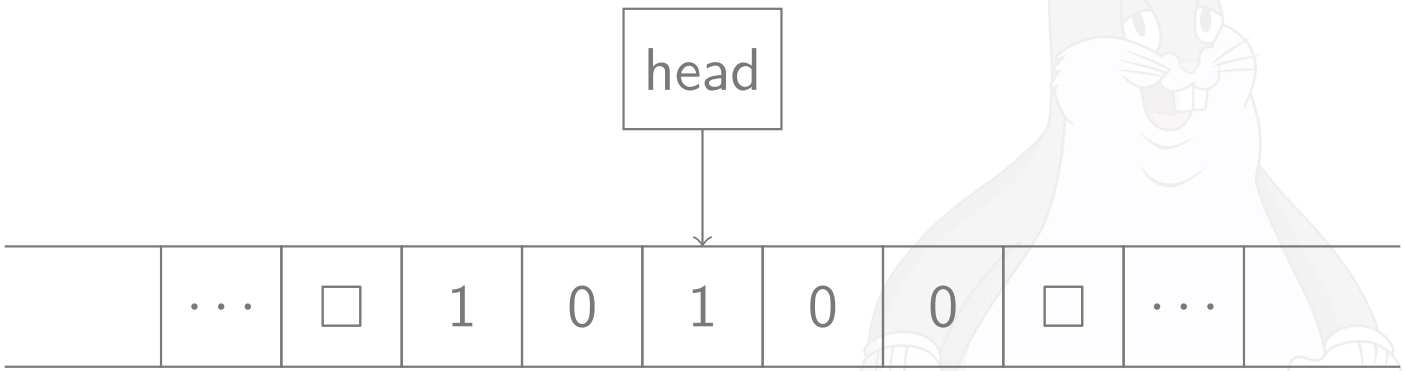



Transition to state HALT, write 1, and move left.

# Your answer may differ.

| State | □ | 0 | 1 |
|---|---|---|---|
| MOVE | (CARRY, □, L) | (MOVE, 0, R) | (MOVE, 1, R) |
| CARRY | (HALT, 1, L) | (HALT, 1, L) | (CARRY, 0, L) |

Current state: HALT




Since HALT is a final state, return '10100'.

# The power of Turing machines

> *Anything a computer program can do can also be done by a Turing machine.*
>
> Church-Turing Thesis, probably (not really)

We won't prove the above statement, just believe in it.[4]

The above allows us to abstract the construction of Turing machines. For example:

$M(s)$ : sort $s$ in place with its characters in alphabetical order
halt (and return $s$ which has been sorted)

But since computers can simulate Turing machines,[5] *there exists a Turing machine that simulates Turing machines*. This Turing machine simulating Turing machine is called the **universal Turing machine**.

[4] Big Chungus demands you to believe in it, and erase any doubts you have.

[5] sponsored by `https://turingmachinesimulator.com/`

# Acceptance and refusal (from cs post ;-;)

Sometimes we want Turing machines to "accept" and "reject" strings, just like DFAs.[6]

Remember $F$, the set of final states in the definition of a Turing machine? Instead of that, we add two additional states to $Q$, called $q_{\text{accept}}$ and $q_{\text{reject}}$.

Instead of transitioning to halting states, we transition to either $q_{\text{accept}}$ or $q_{\text{reject}}$, and halt.

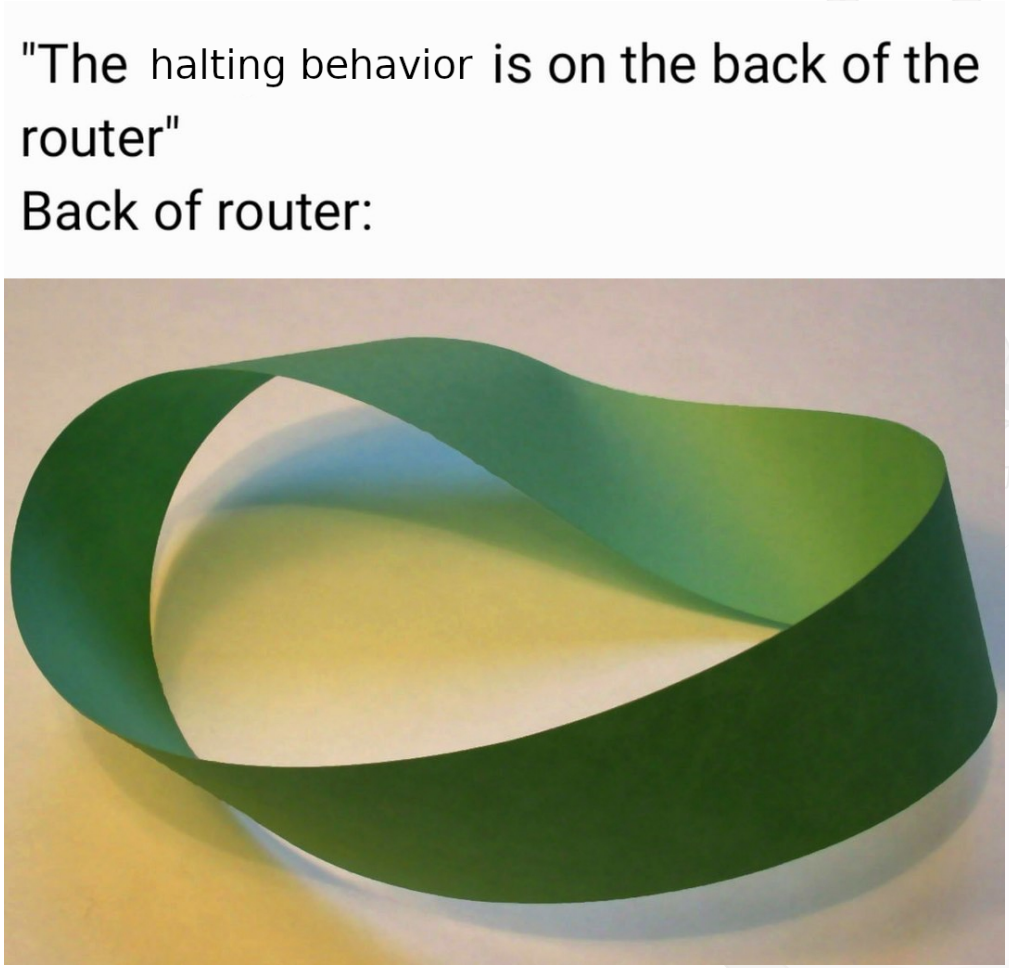
We say a TM **accepts** a string $s$ if on input $s$, it halts and ends in state $q_{\text{accept}}$. We say a TM **rejects** $s$ if on input $s$, it halts and ends in $q_{\text{reject}}$.

Note a TM may still loop on input $s$. In this case it neither accepts or rejects, and we say it **loops** on $s$. A TM **halts** on $s$ if it either accepts or rejects $s$.

[6] In this convention, the Turing machine is a function that takes in a string, and outputs a boolean value (assuming it halts).

# Quick exercise

Give an example of a Turing machine that doesn't halt on any input.



# Acceptance and refusal (from cs post ;-;)

Example: we can modify the add-1 Turing machine earlier to reject inputs not starting with 1, and otherwise add 1 to the input and accept.

Original: set of states was $\{\text{MOVE}, \text{CARRY}, \text{HALT}\}$, initial state MOVE.

| State | $\square$ | 0 | 1 |
|---|---|---|---|
| MOVE | (CARRY, $\square$, L) | (MOVE, 0, R) | (MOVE, 1, R) |
| CARRY | (HALT, 1, L) | (HALT, 1, L) | (CARRY, 0, L) |

Modified: $\{\text{START}, \text{MOVE}, \text{CARRY}, q_{\text{accept}}, q_{\text{reject}}\}$, initial state START.

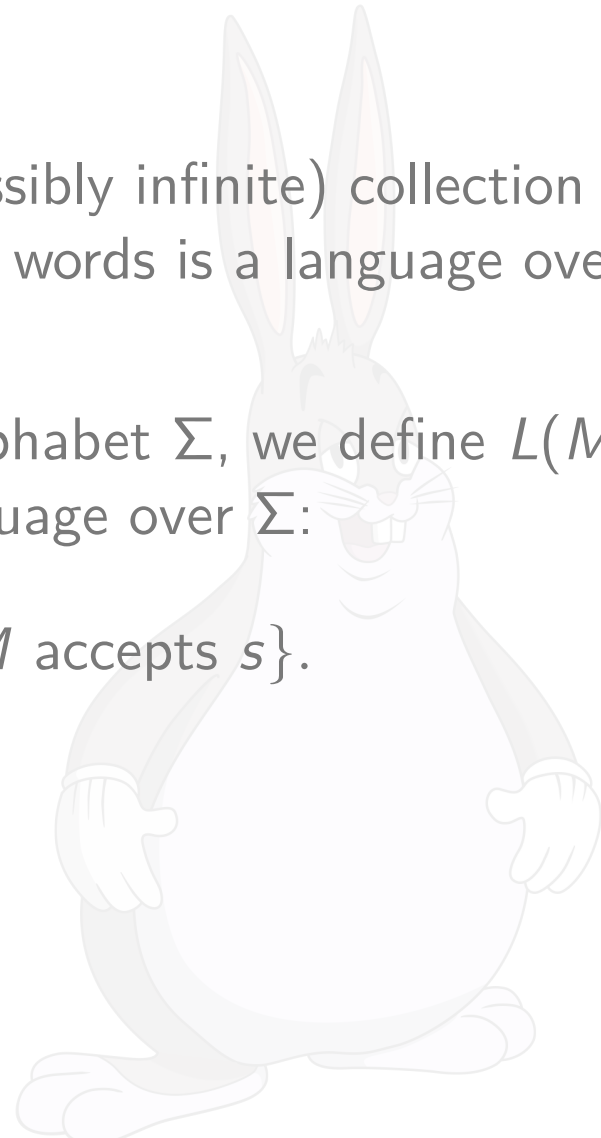
| State | $\square$ | 0 | 1 |
|---|---|---|---|
| START | ($q_{\text{reject}}$, $\square$, R) | ($q_{\text{reject}}$, $\square$, R) | (MOVE, 1, R) |
| MOVE | (CARRY, $\square$, L) | (MOVE, 0, R) | (MOVE, 1, R) |
| CARRY | ($q_{\text{accept}}$, 1, L) | ($q_{\text{accept}}$, 1, L) | (CARRY, 0, L) |

# Computable and listable Sets

A **language** over an alphabet $\Sigma$ is a (possibly infinite) collection of strings from $\Sigma^*$. For example, the set of English words is a language over the English alphabet.

Given a Turing machine $M$ with input alphabet $\Sigma$, we define $L(M)$ (the **language** of $M$) to be the following language over $\Sigma$:

$$L(M) = \{s \in \Sigma^* : M \text{ accepts } s\}.$$



# Computable and listable Sets

A language $\mathcal{L}$ over $\Sigma$ is **listable** if there exists a Turing machine $M$ with input alphabet $\Sigma$ such that $\mathcal{L} = L(M)$.[7]

A language $\mathcal{L}$ over $\Sigma$ is **decidable** if there exists a Turing machine $M$ with input alphabet $\Sigma$ such that $\mathcal{L} = L(M)$, *and $M$ halts on all input strings*.[8]
The "hierarchy" of languages goes like this:

$$\text{Regular Languages} \subset \text{Decidable L.} \subset \text{Listable L.} \subset \text{All L.}$$

If you don't remember regular languages[9] from CSC236, don't worry.

---

[7] The definition of "listable" given here is equivalent to the Turing-machine definition from the first lecture, which is equivalent to "computably enumerable" from the second lecture.

[8] Same here. A Turing machine that halts on all inputs is called a **decider**.

[9] Every regular language is decidable. Why? We can simulate a DFA on a computer, so we can simulate a DFA on a Turing machine. DFAs always halt, so the DFA simulator always halts given any input.

# Computable and listable Sets

One more thing to note: each Turing machine can be encoded in an ASCII string.[10]

```
1  name: add1
2  init: move_right
3  accept: halt
4
5  move_right, _
6  carry_over, _, <
7
8  move_right, 0
9  move_right, 0, >
10
11 move_right, 1
12 move_right, 1, >
13
14 carry_over, _
15 halt, 1, <
```

So if $\Sigma$ is the ASCII alphabet, then "the set of valid Turing machine encodings" is a language over $\Sigma$.

[10] again, sponsored by `https://turingmachinesimulator.com/`

# Halting problem

Now we can prove the existence of an undecidable but listable language. This language is called the **halting problem**, denoted by HP.[11]

$$\mathrm{HP} = \{Mw : M \text{ is a Turing machine that halts on } w\}.$$

Why is this language listable? The language of the following "Turing machine" $M'$ is precisely HP:[12]

$M'(s)$ : if $s$ is of the form $Mw$:
Simulate $M$ on $w$
accept
reject

Note that this isn't guaranteed to halt: in the case $M$ doesn't halt on $w$, the third line will take forever. So this doesn't show HP is decidable.

[11] Formally I have to declare the alphabet first, but this is usually omitted if the alphabet is not relevant. I'll just choose the alphabet to be the ASCII characters.

[12] Remember, whatever a computer can do, so can a Turing machine.

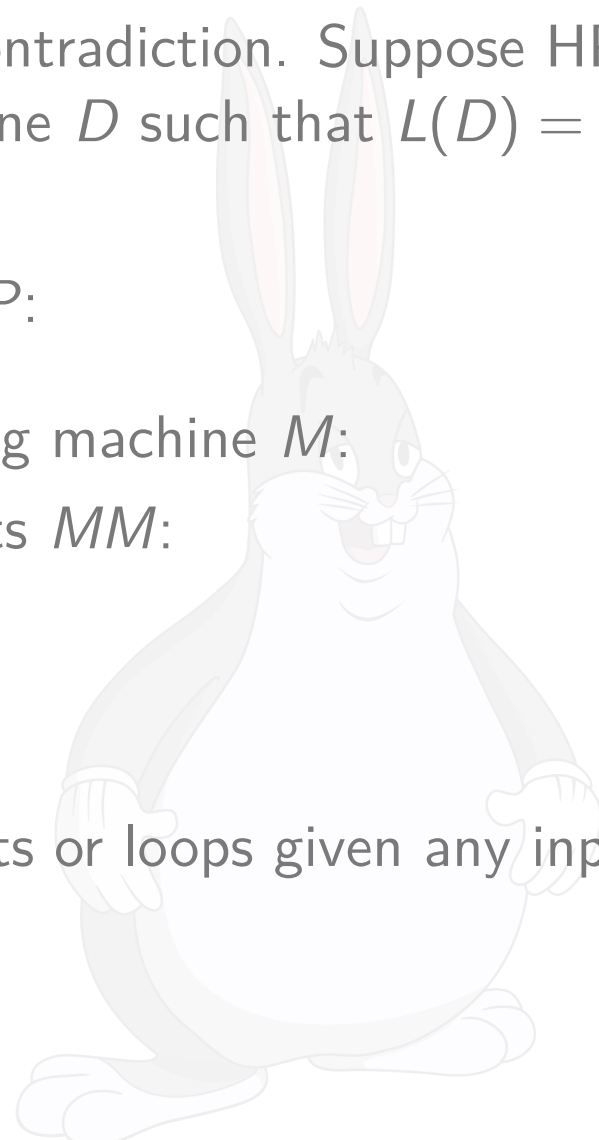
# Halting problem

Why isn't HP decidable? We prove by contradiction. Suppose HP is decidable (so there exists a Turing machine $D$ such that $L(D) = \text{HP}$ and $D$ halts on all inputs).

Construct the following Turing machine $P$:

$P(s)$ : if $s$ is a Turing machine $M$:
If $D$ accepts $MM$:
loop
accept

Notice $P$ never rejects, so it either accepts or loops given any input. Consider $P(P)$.



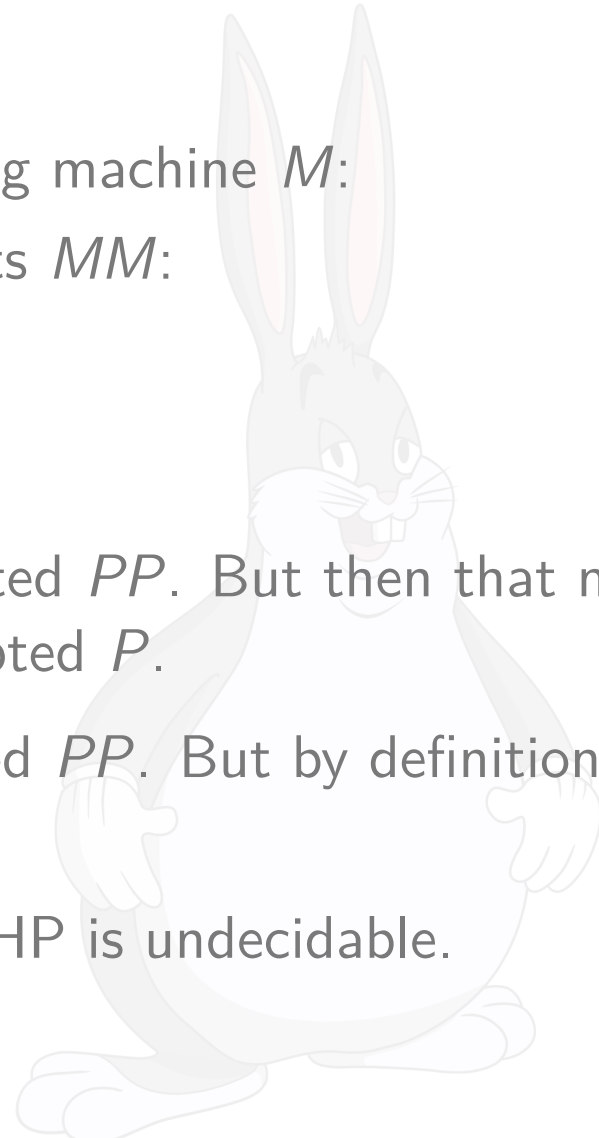
# Halting problem

$P(s)$ : if $s$ is a Turing machine $M$:
If $D$ accepts $MM$:
loop
accept

If $P(P)$ accepts, then $D$ must have rejected $PP$. But then that means $P(P)$ doesn't halt, so $P$ can't have accepted $P$.

If $P(P)$ loops, then $D$ must have accepted $PP$. But by definition that means $P(P)$ halts, so $P(P)$ can't loop.

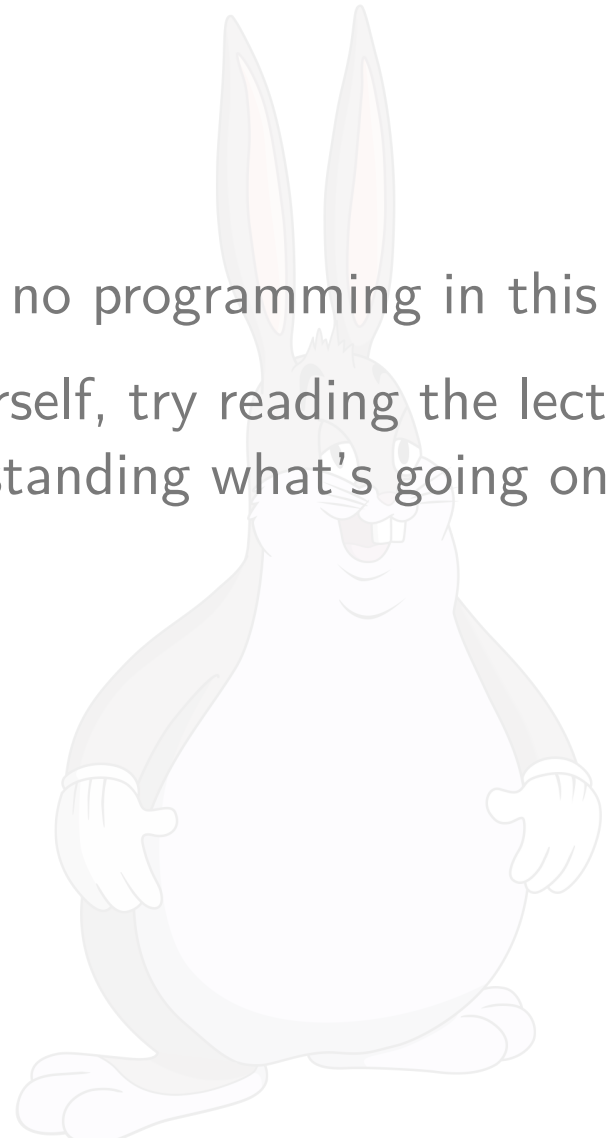
We conclude no such $D$ exists, meaning HP is undecidable.



# Okay but what's the practical use?

None that I know of. Hey, we mentioned no programming in this course!

Anyway, bye! If you wanna challenge yourself, try reading the lecture slides from this week again, but actually understanding what's going on by relating it to what I presented here.



# License

**Get the source of this theme and the demo presentation from**

`http://github.com/famuvie/beamerthemesimple`

The theme itself is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.