Turing Machines

Part Two

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.

Very Important Terminology

Let M be a Turing machine and let w be a string.

M ***accepts w*** if it enters an accept state when run on w.

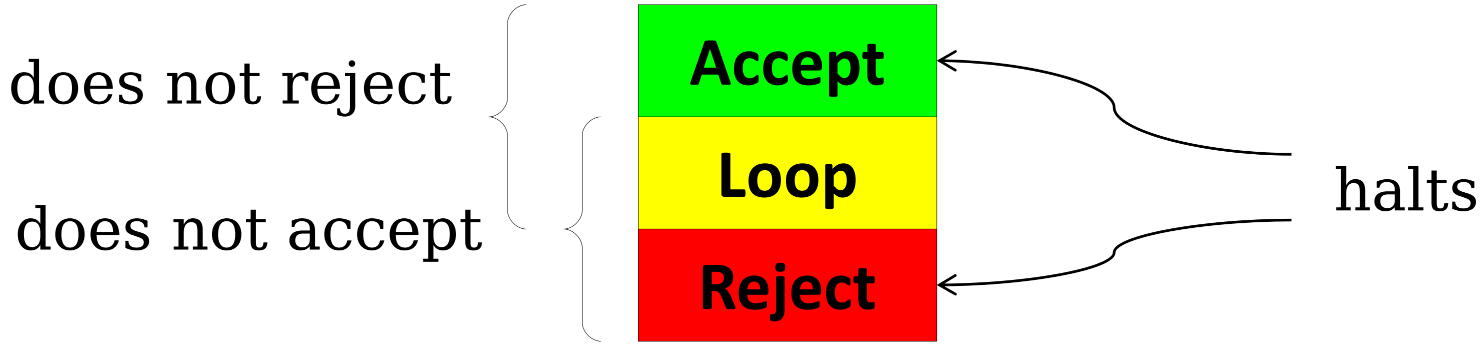
M ***rejects w*** if it enters a reject state when run on w.

M ***loops infinitely on w*** (or just ***loops on w***) if when run on w it enters neither an accept nor a reject state.

M ***does not accept w*** if it either rejects w or loops infinitely on w.

M ***does not reject w*** w if it either accepts w or loops on w.

M ***halts on w*** if it accepts w or rejects w.



The Language of a TM

The language of a Turing machine M, denoted $\mathscr{L}(M)$, is the set of all strings that M accepts:

$$\mathscr{L}(\boldsymbol{M}) = \{\ \boldsymbol{w} \in \boldsymbol{\Sigma^*} \mid \boldsymbol{M} \textbf{ accepts } \boldsymbol{w}\ \}$$

For any $w \in \mathscr{L}(M)$, M accepts w.

For any $w \notin \mathscr{L}(M)$, M does not accept w.

M might reject w, or it might loop on w.

A language is called ***recognizable*** if it is the language of some TM.

A TM M where $\mathscr{L}(M) = L$ is called a ***recognizer*** for L.

Notation: the class **RE** is the set of all recognizable languages.

$$L \in \textbf{RE} \quad \leftrightarrow \quad L \text{ is recognizable}$$

What do you think? Does that correspond to what you think it means to solve a problem?

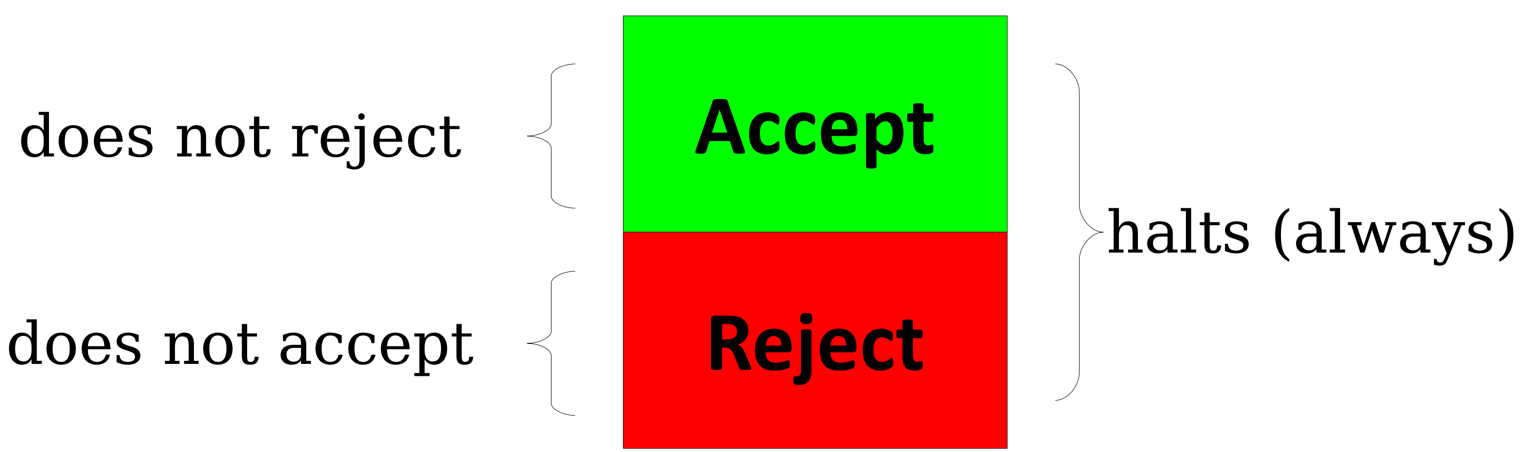
New Stuff!

Deciders

Some Turing machines always halt; they never go into an infinite loop.

If M is a TM and M halts on every possible input, then we say that M is a ***decider***.

For deciders, accepting is the same as not rejecting and rejecting is the same as not accepting.



Decidable Languages

A language L is called ***decidable*** if there is a decider M such that $\mathscr{L}(M) = L$.

Equivalently, a language L is decidable if there is a TM M such that

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

The class **R** is the set of all decidable languages.

$$L \in \mathbf{R} \quad \leftrightarrow \quad L \text{ is decidable}$$

Decidable problems, in some sense, problems that can definitely be "solved" by a computer.

A Feel for **R** and **RE**

Say you're working on a CS assignment and you ask yourself the question "does my program have a bug?"

- An **RE** perspective: if you find a bug, you know for sure the answer is "yes", but not finding one doesn't necessarily mean the answer is "no".
- An **R** perspective: it would be *great* if there were a magic program that could look at your code and tell you whether it's correct. *(Does something like this exist?)*

R and **RE** Languages

Every decider is a Turing machine, but not every Turing machine is a decider.

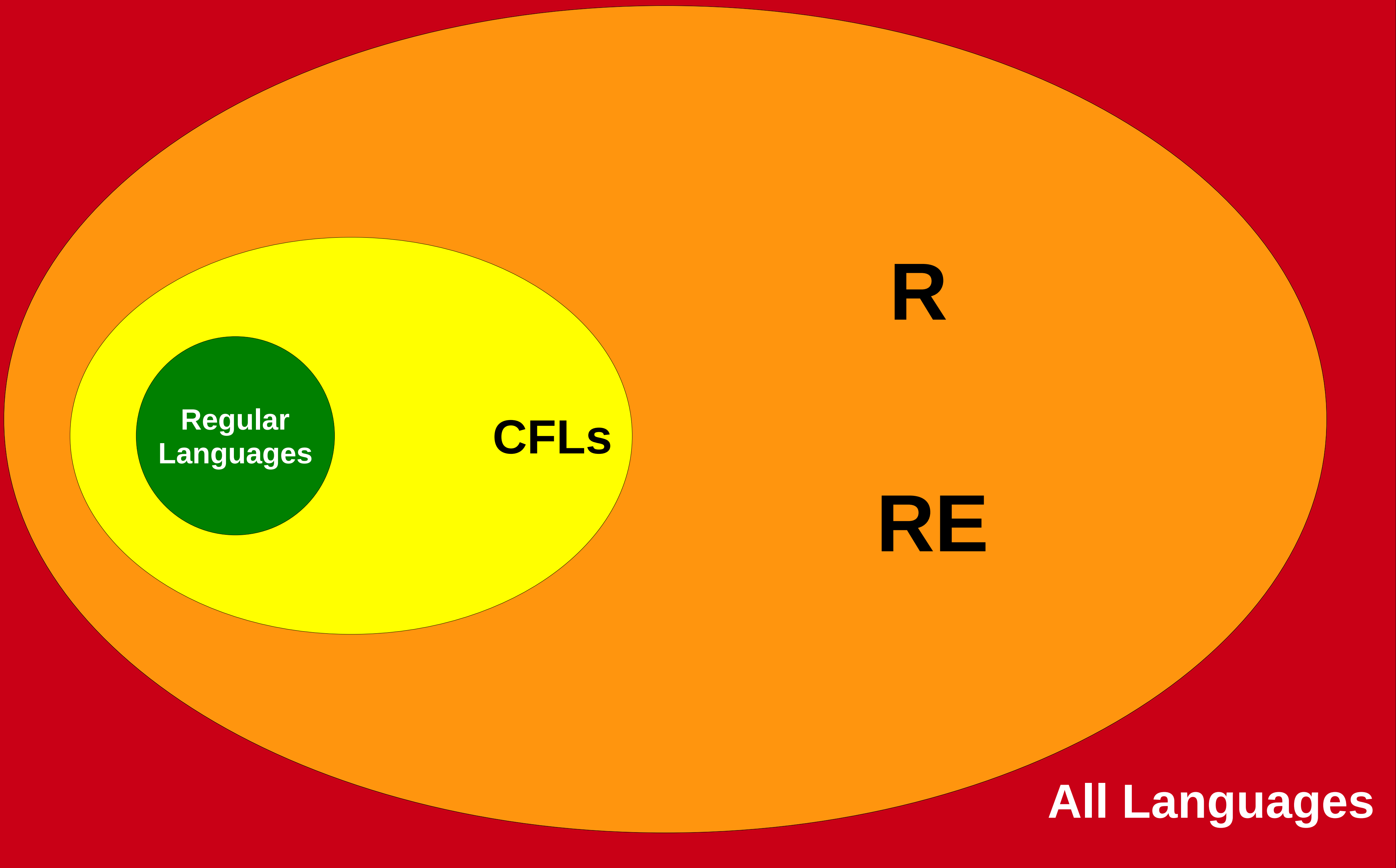
This means that $\mathbf{R} \subseteq \mathbf{RE}$.

Hugely important theoretical question:

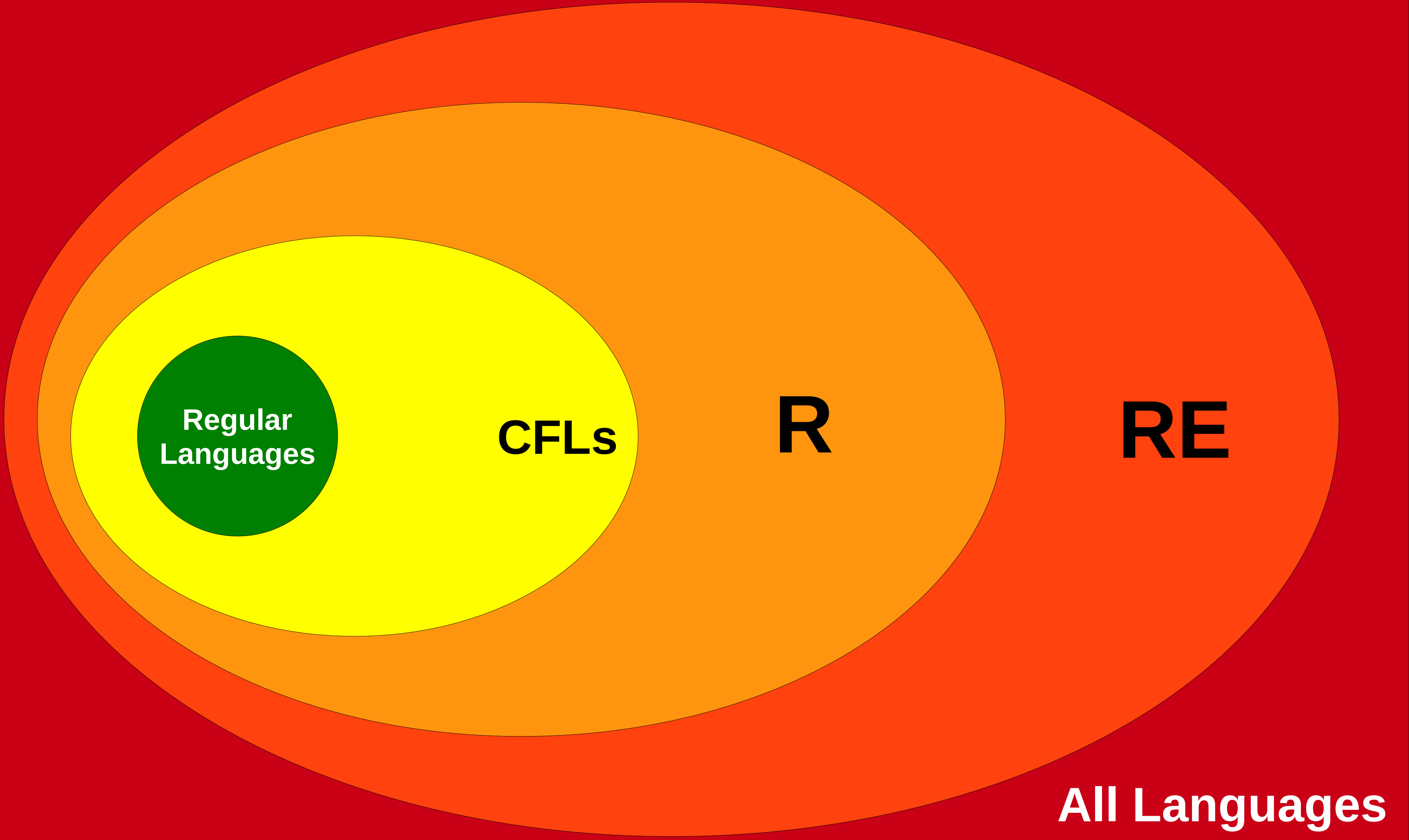
$$\mathbf{R} \stackrel{?}{=} \mathbf{RE}$$

That is, if you can just confirm "yes" answers to a problem, can you necessarily *solve* that problem?

Which Picture is Correct?



Which Picture is Correct?



What problems can we solve with a computer?

What is a “problem?”

Decision Problems

A ***decision problem*** is a type of problem where the goal is to provide a yes or no answer.

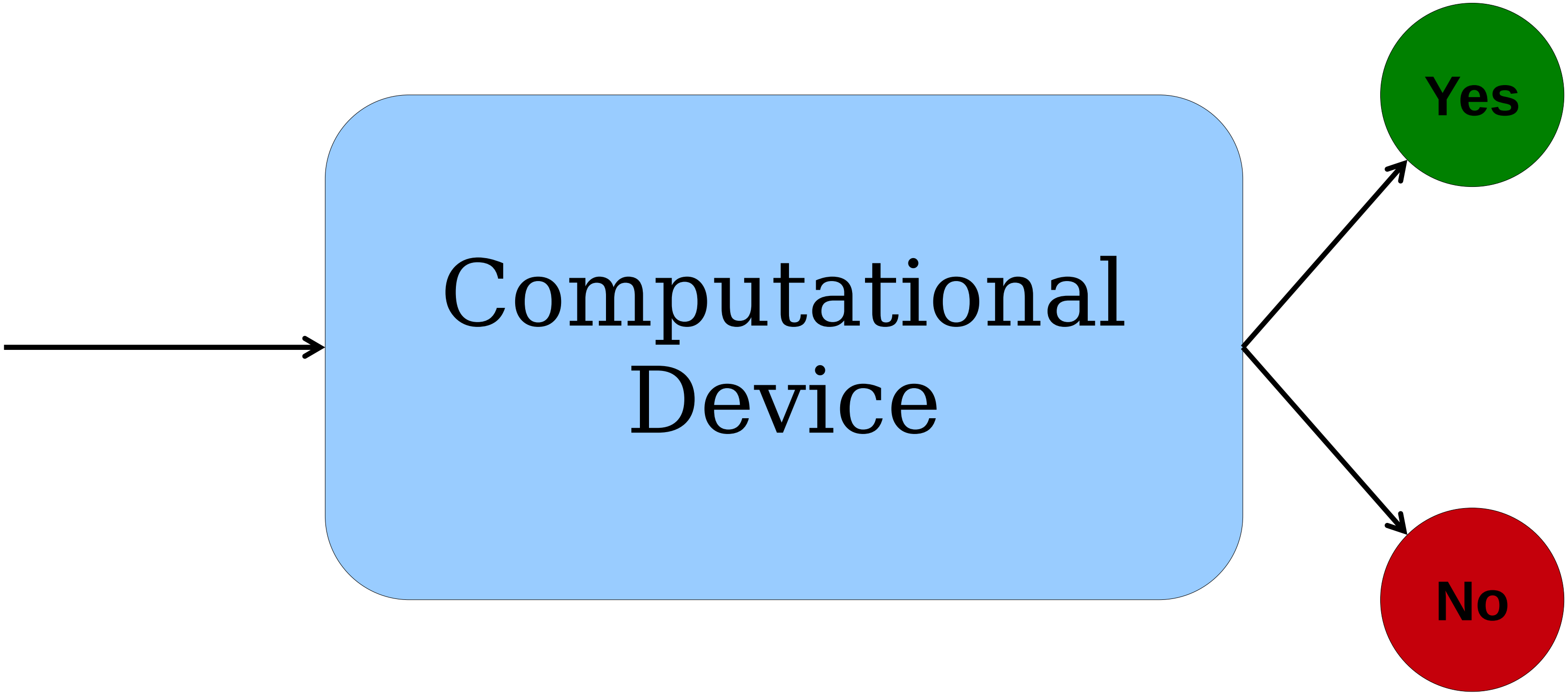
Example: Bin Packing

You're given a list of patients who need to be seen and how much time each one needs to be seen for. You're given a list of doctors and how much free time they have. Is there a way to schedule the patients so that they can all be seen?

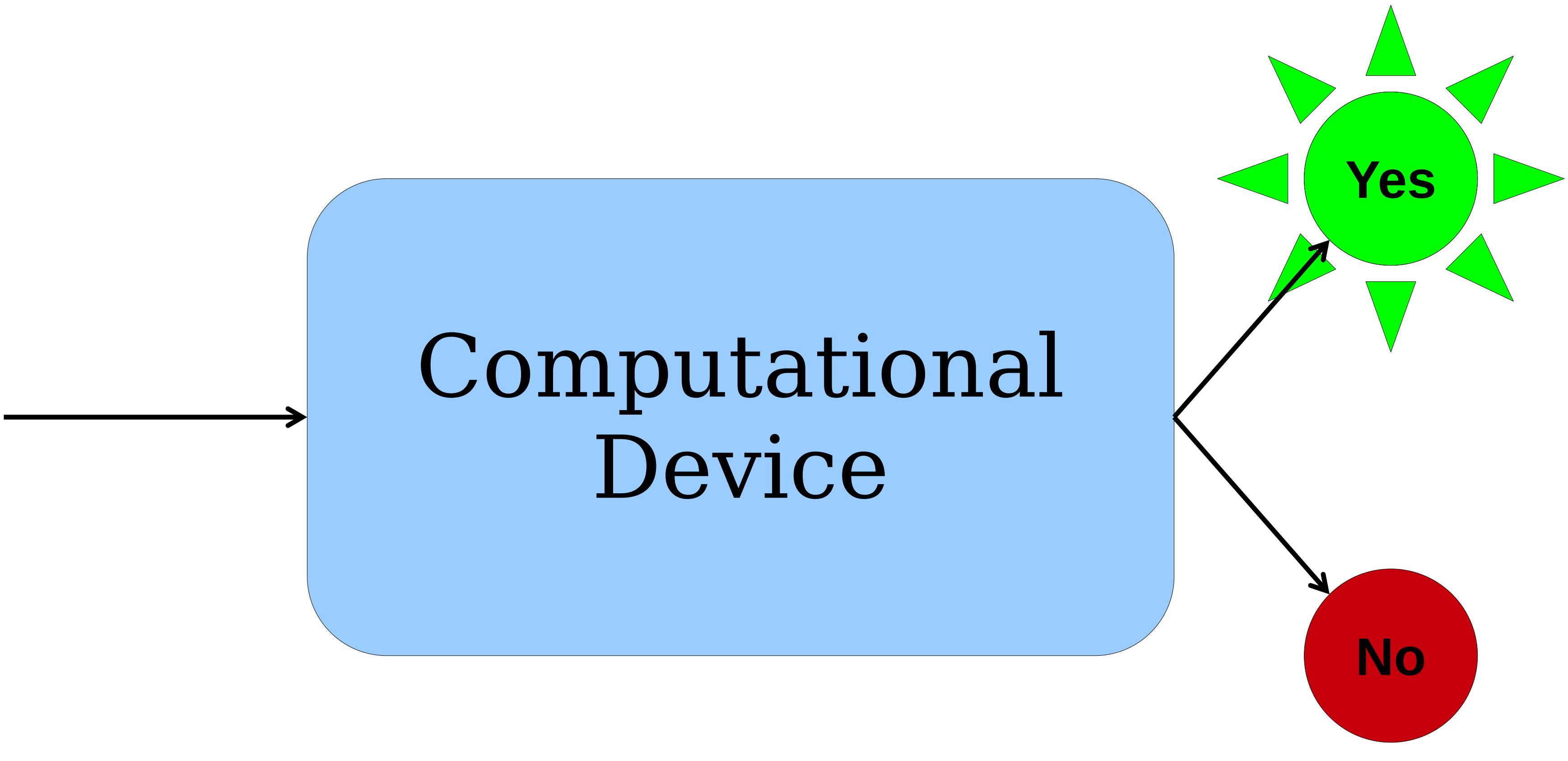
Example: Dominating Set Problem

You're given a transportation grid and a number k. Is there a way to place emergency supplies in at most k cities so that every city either has emergency supplies or is adjacent to a city that has emergency supplies?

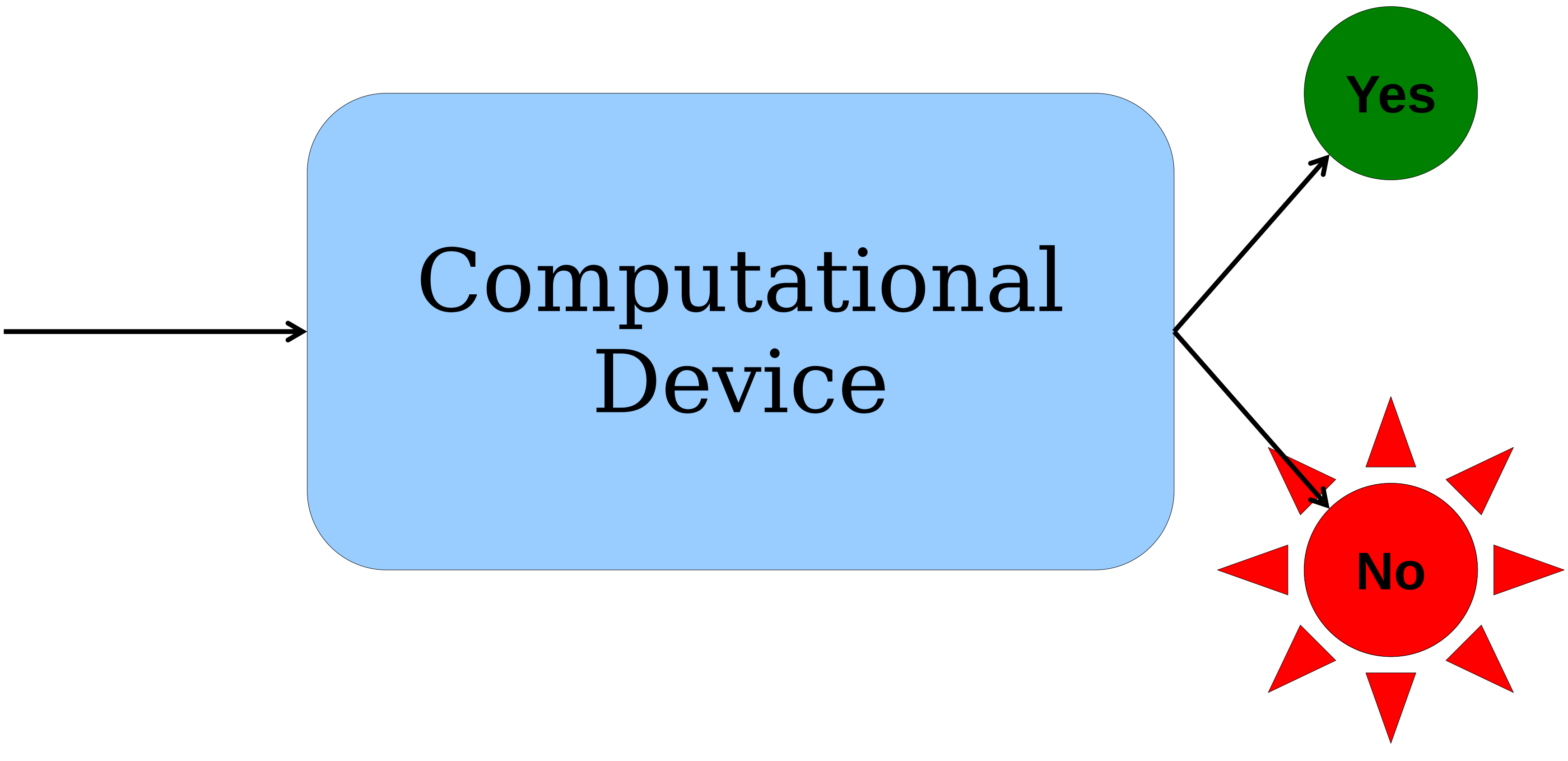
A Model for Solving Problems



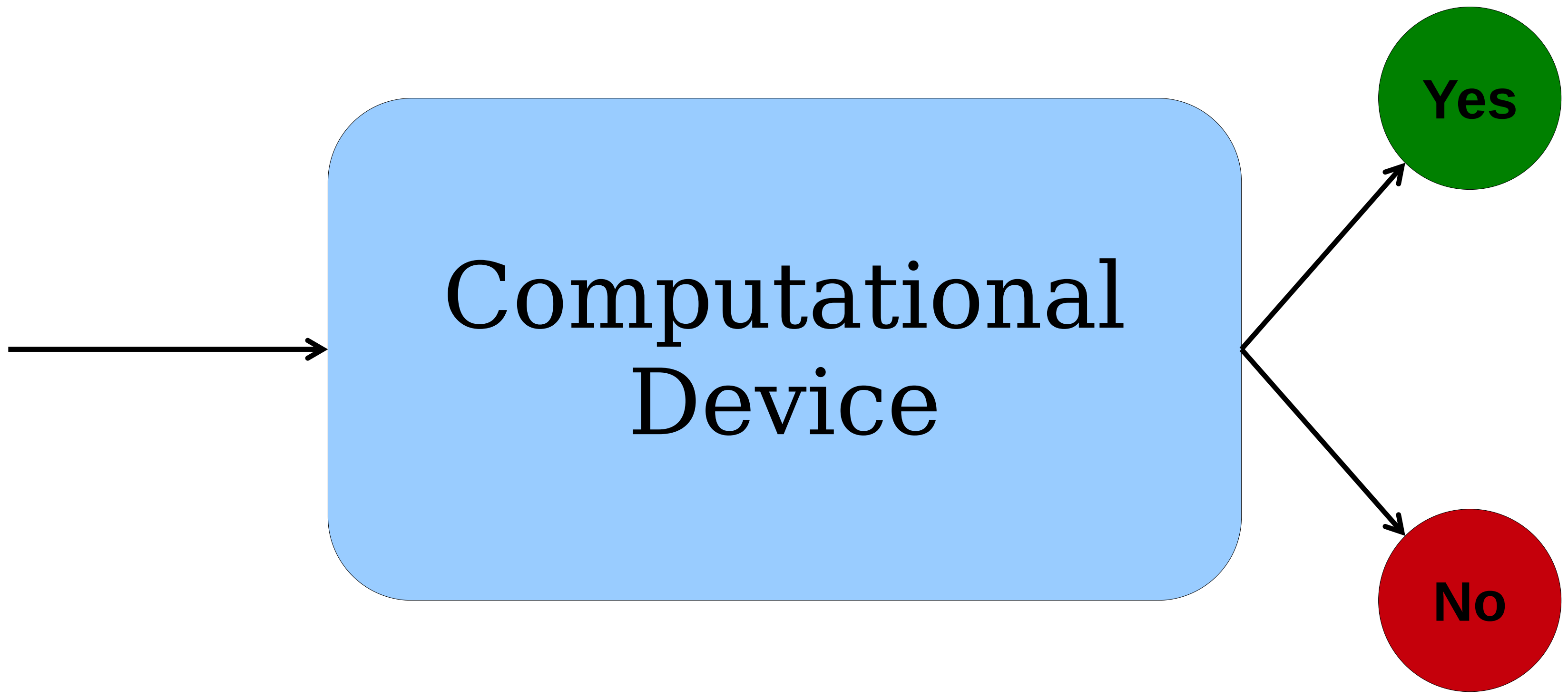
A Model for Solving Problems



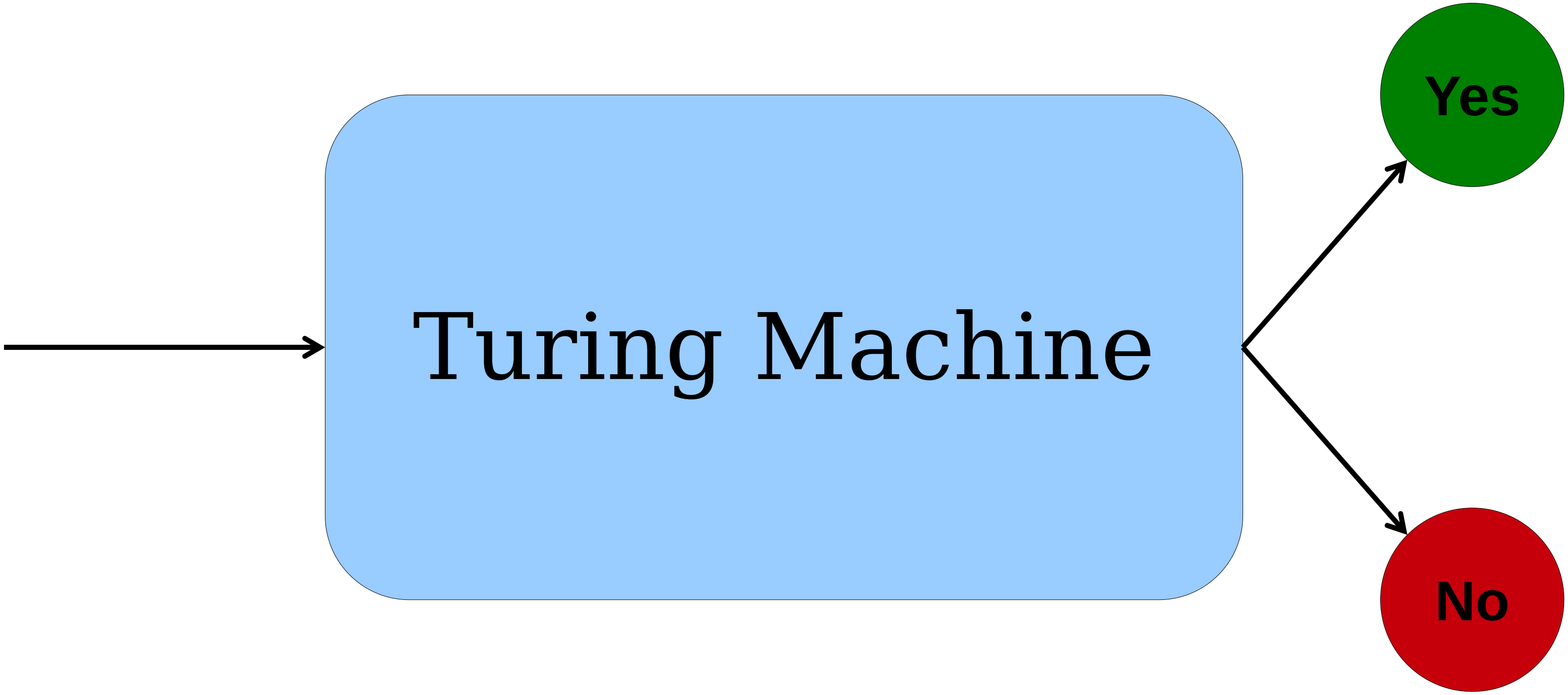
A Model for Solving Problems



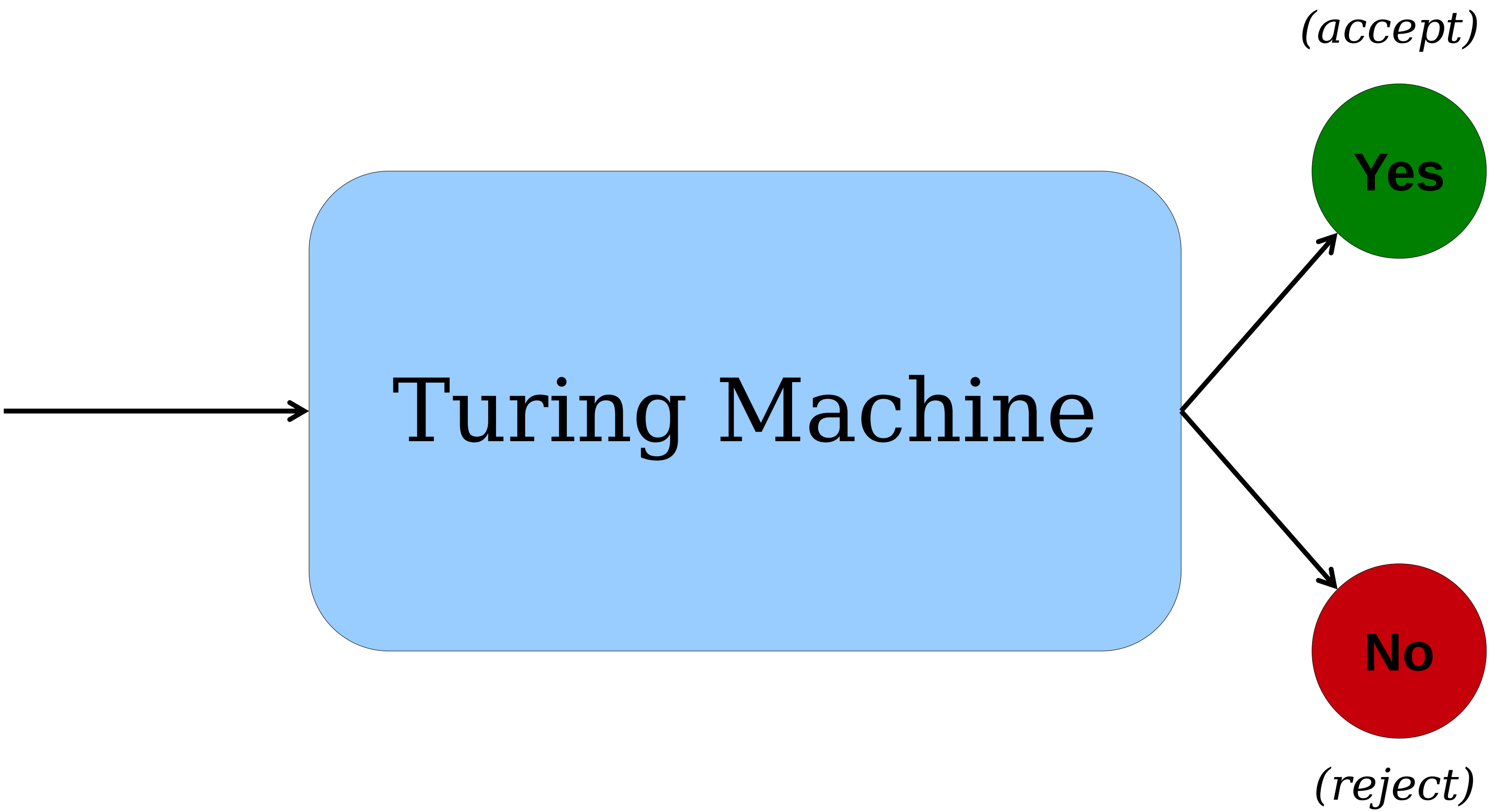
A Model for Solving Problems



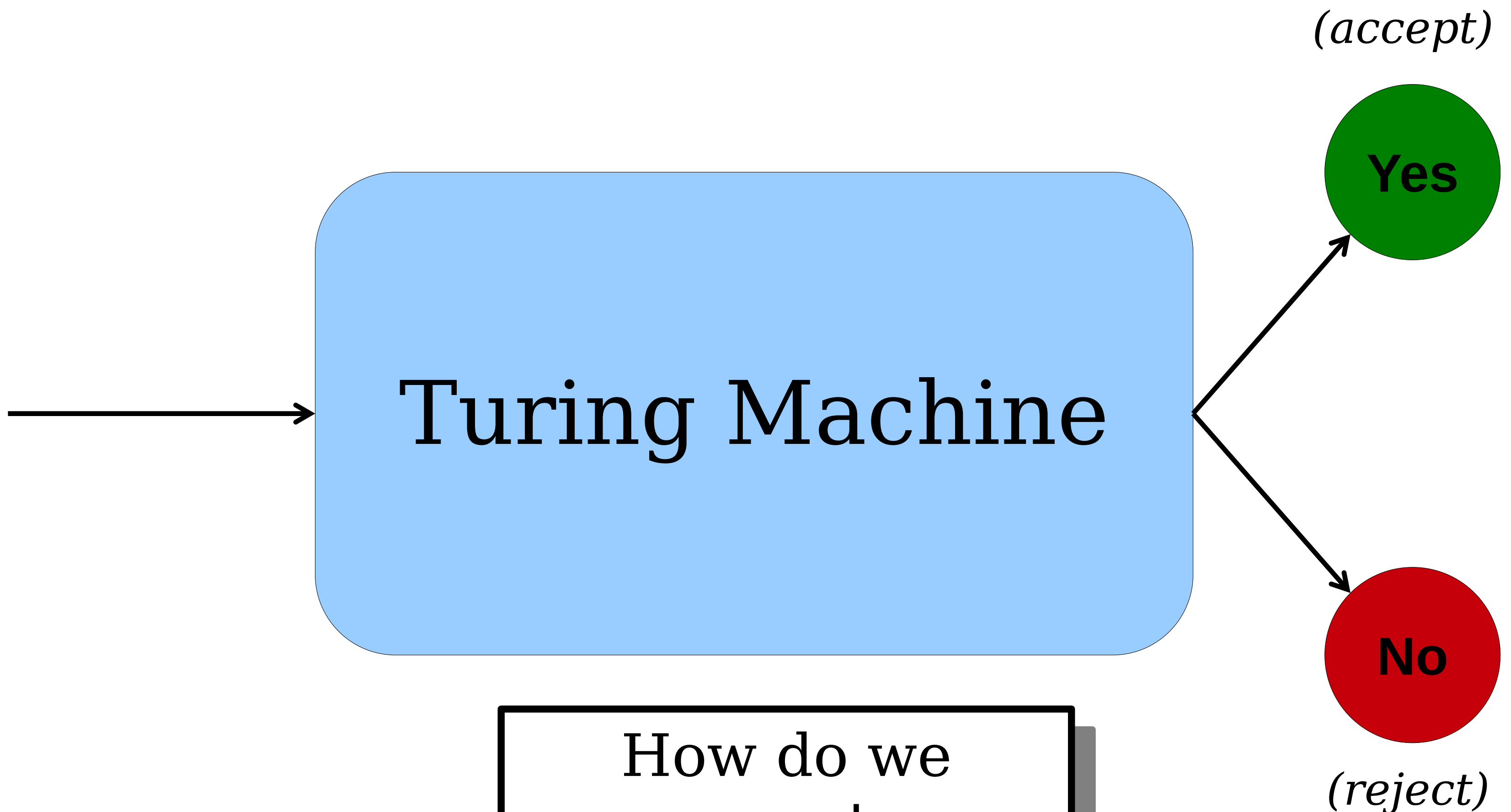
A Model for Solving Problems



A Model for Solving Problems



A Model for Solving Problems



How do we represent our inputs?

Humbling Thought:

Everything on your computer is a string over {0, 1}.

Strings and Objects

Think about how my computer encodes the image on the right.

Internally, it's just a series of zeros and ones sitting on my hard drive.

Strings and Objects

A different sequence of 0s and 1s gives rise to the image on the right.

Every image can be encoded as a sequence of 0s and 1s, though not all sequences of 0s and 1s correspond to images.

Object Encodings

If *Obj* is some mathematical object that is *discrete* and *finite*, then we'll use the notation **⟨*Obj*⟩** to refer to some way of encoding that object as a string.

Think of ⟨*Obj*⟩ like a file on disk – it encodes some high-level object as a series of characters.

⟨  ⟩

= 11011100101110111100010011…110

Object Encodings

If *Obj* is some mathematical object that is *discrete* and *finite*, then we'll use the notation **⟨*Obj*⟩** to refer to some way of encoding that object as a string.

Think of ⟨*Obj*⟩ like a file on disk – it encodes some high-level object as a series of characters.

⟨  ⟩ = 00110101000101000101000100…001

Object Encodings

For the purposes of what we're going to be doing, we aren't going to worry about exactly *how* objects are encoded.

For example, we can say ⟨137⟩ to mean "some encoding of 137" without worrying about how it's encoded.

Analogy: do you need to know how the `int` type is represented in C++ to do basic C++ programming? That's more of a CS107 question.

We'll assume, whenever we're dealing with encodings, that some Smart, Attractive, Witty person has figured out an encoding system for us and that we're using that encoding system.

Encoding Groups of Objects

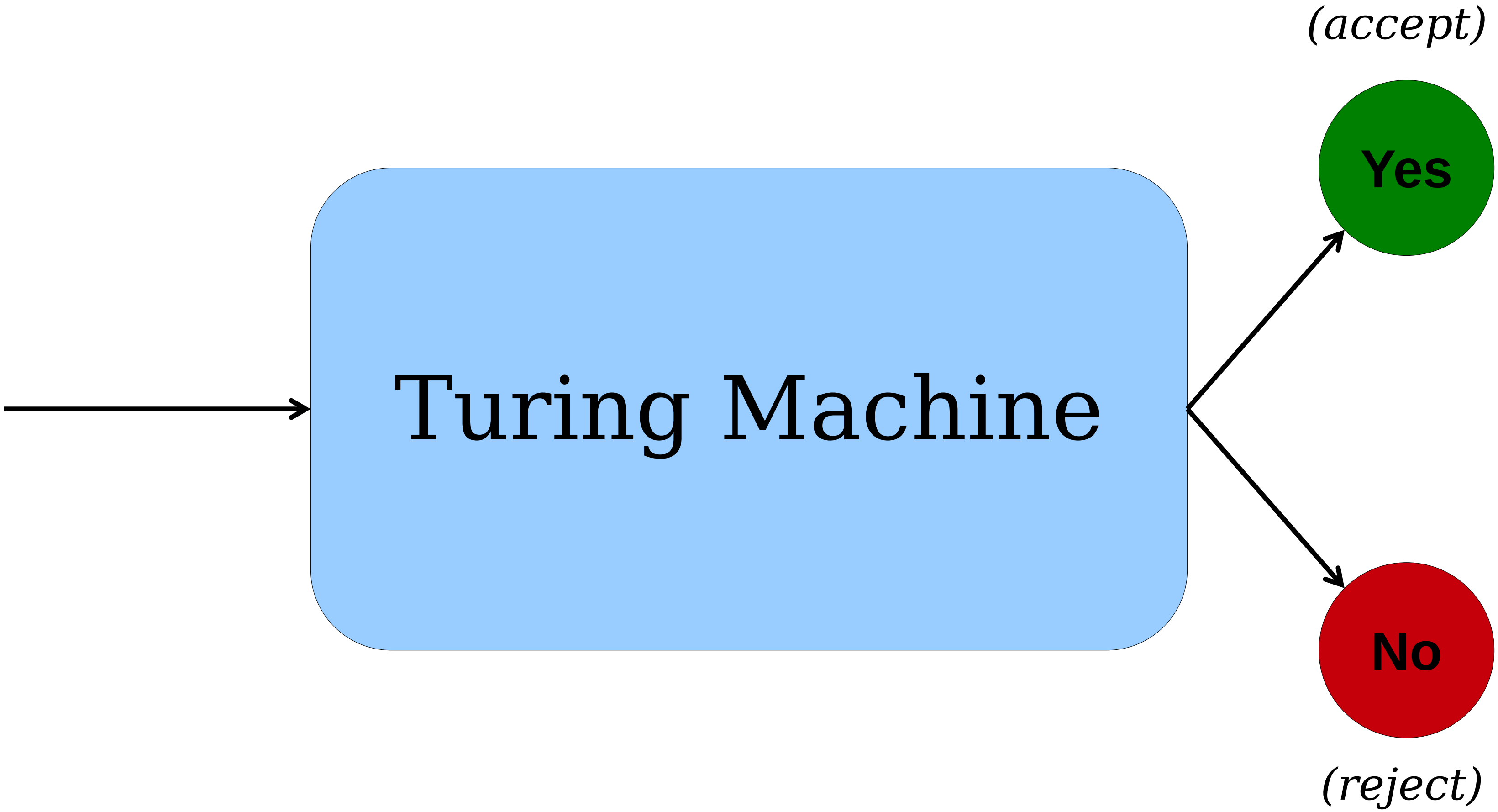
Given a group of objects Obj_1, Obj_2, …, Obj_n, we can create a single string encoding all these objects.

- Think of it like a .zip file, but without the compression.

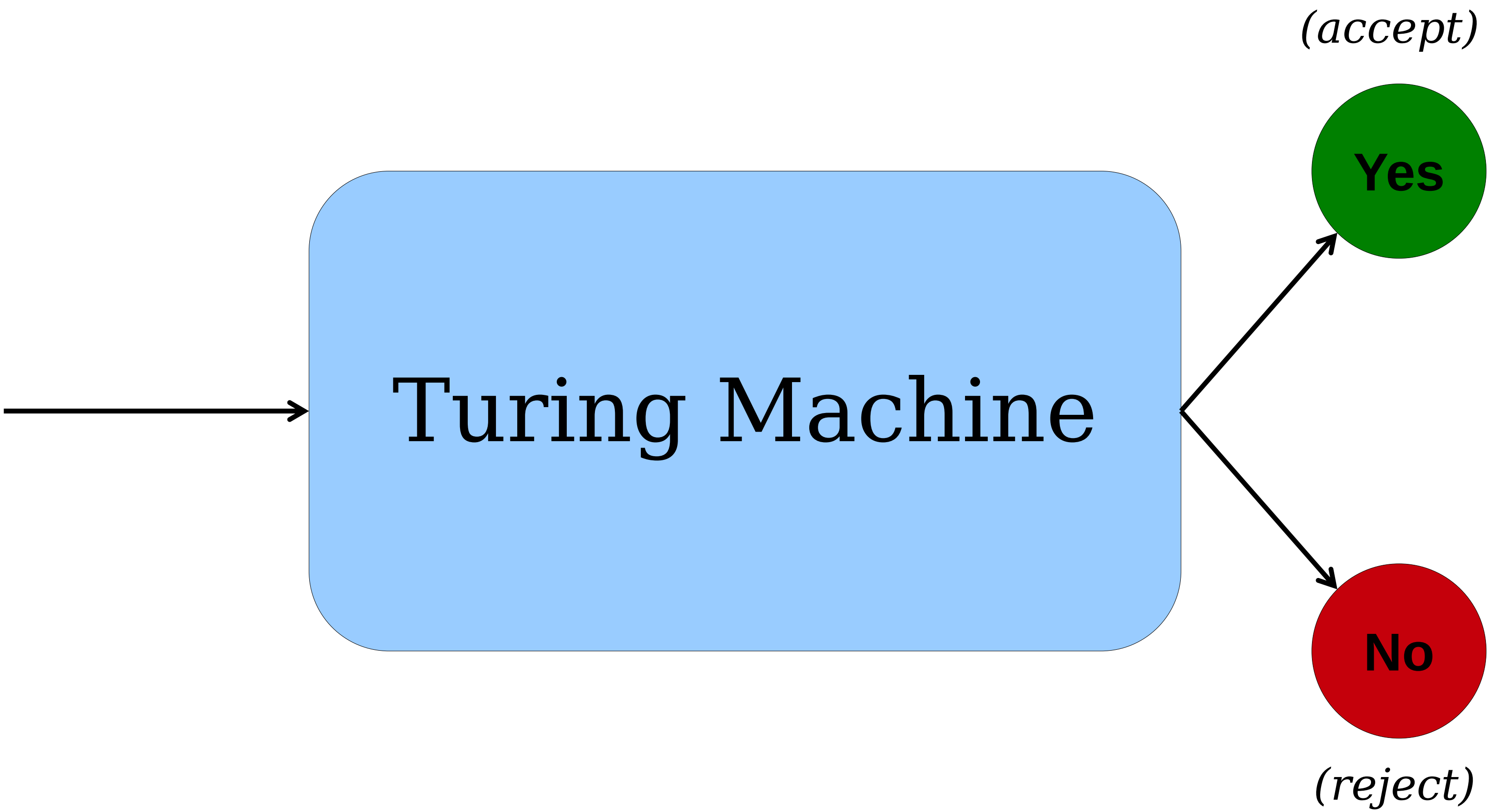
We'll denote the encoding of all of these objects as a single string by **$\langle Obj_1, \ldots, Obj_n \rangle$**.

This lets us feed multiple inputs into our computational device at the same time.

A Model for Solving Problems



A Model for Solving Problems



What problems can we solve with a computer?

Emergent Properties

Emergent Properties

An ***emergent property*** of a system is a property that arises out of smaller pieces that doesn't seem to exist in any of the individual pieces.

Examples:

- Individual neurons work by firing in response to particular combinations of inputs. Somehow, this leads to consciousness, love, and ennui.
- Individual atoms obey the laws of quantum mechanics and just interact with other atoms. Somehow, it's possible to combine them together to make iPhones and pumpkin pie.

Emergent Properties of Computation

- All computing systems equal to Turing machines exhibit several surprising emergent properties.
- If we believe the Church-Turing thesis, these emergent properties are, in a sense, “inherent” to computation. Computation can’t exist without them.
- These emergent properties are what ultimately make computation so interesting and so powerful.
- As we'll see, though, they're also computation's Achilles heel – they're how we find concrete examples of impossible problems.

Two Emergent Properties

There are two key emergent properties of computation that we will discuss:

- ***Universality***: There is a single computing device capable of performing any computation.
- ***Self-Reference***: Computing devices can ask questions about their own behavior.

As you'll see, the combination of these properties leads to simple examples of impossible problems and elegant proofs of impossibility.

Universal Machines

An Observation

When we've been discussing Turing machines, we've talked about designing specific TMs to solve specific problems.

Does this match your real-world experiences? Do you have one computing device for each task you need to perform?

Can we make a “reprogrammable Turing machine?”

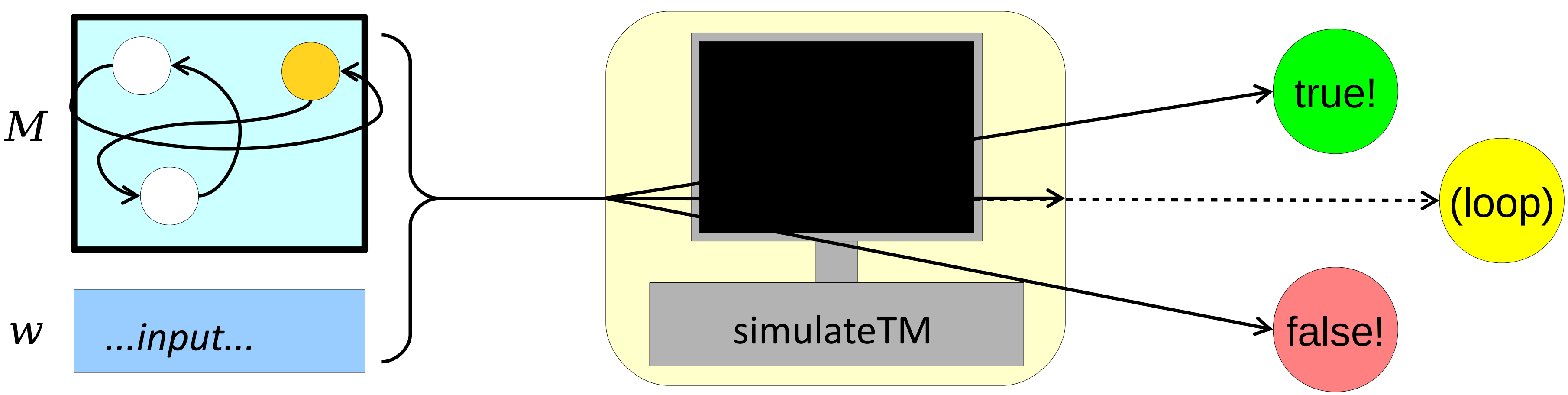
A TM Simulator

- It is possible to program a TM simulator on an unbounded-memory computer.
- We could imagine it as a method

boolean simulateTM(TM M, string w)

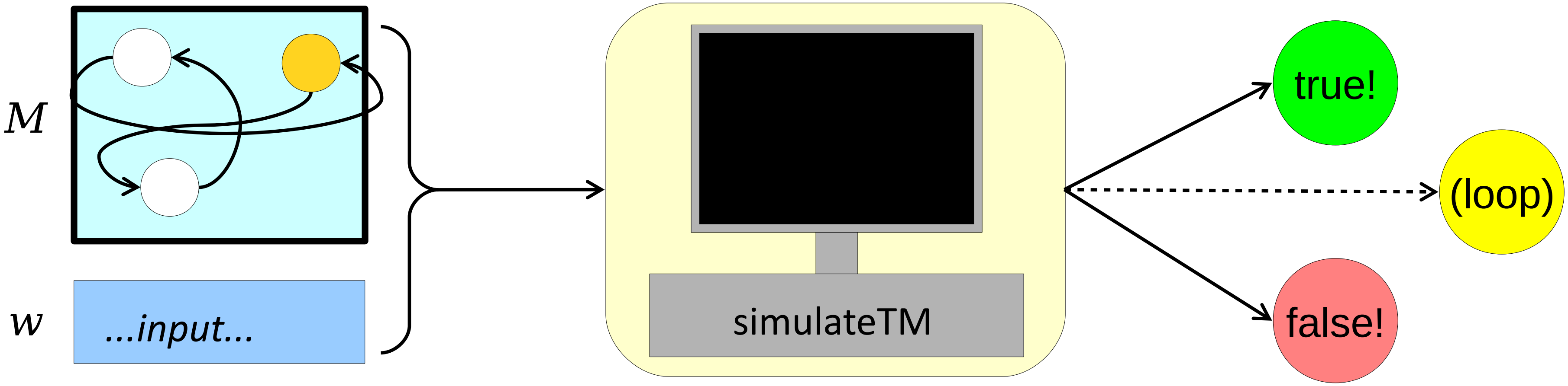
with the following behavior:

- If M accepts w, then simulateTM(M, w) returns **true** .
- If M rejects w, then simulateTM(M, w) returns **false** .
- If M loops on w, then simulateTM(M, w) loops infinitely.



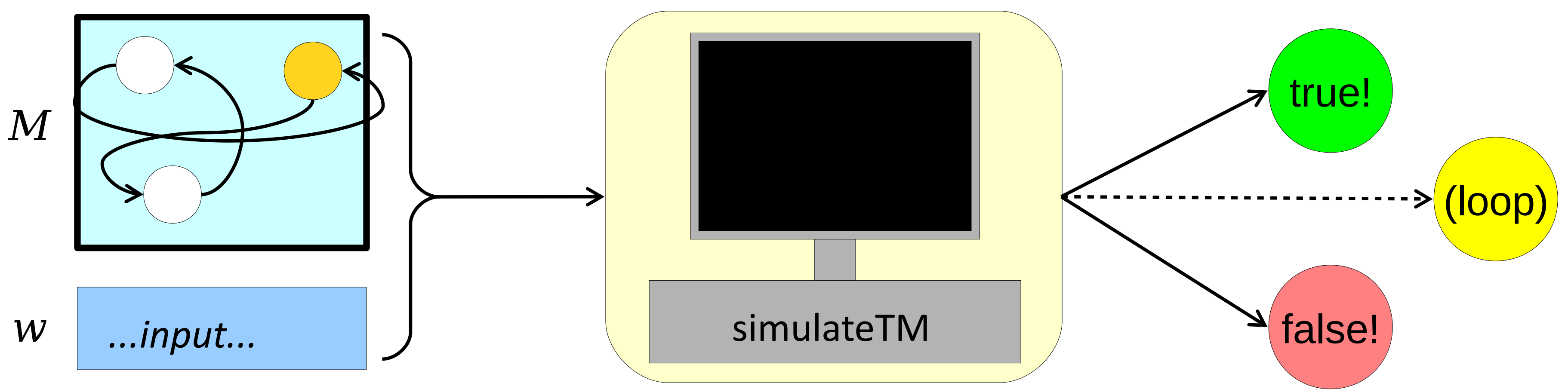
A TM Simulator

- It is known that anything that can be done with an unbounded-memory computer can be done with a TM.



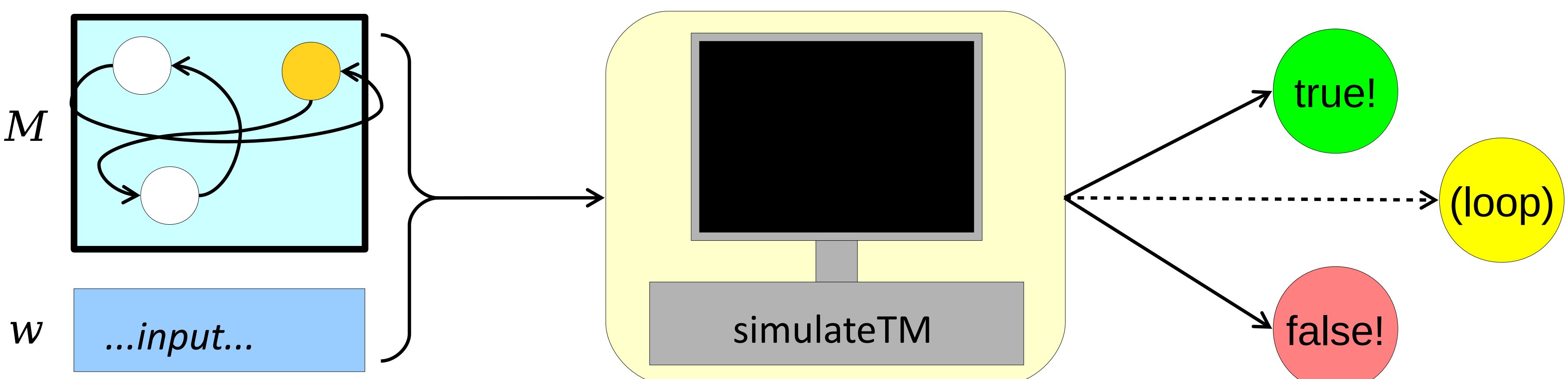
A TM Simulator

- It is known that anything that can be done with an unbounded-memory computer can be done with a TM.
- This means that there must be some TM that has the behavior of this simulateTM method.



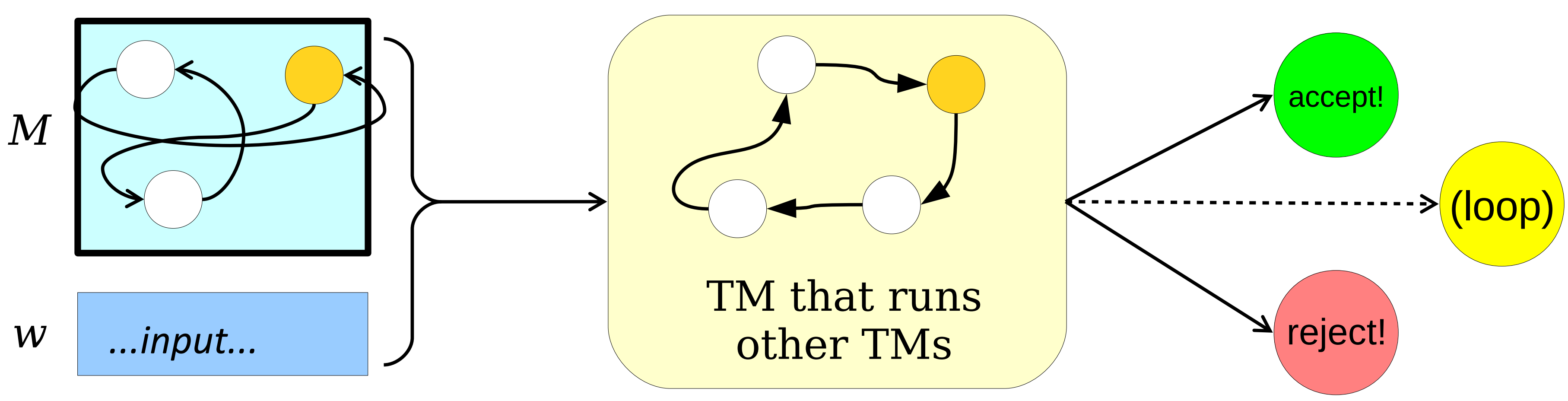
A TM Simulator

- It is known that anything that can be done with an unbounded-memory computer can be done with a TM.
- This means that there must be some TM that has the behavior of this simulateTM method.
- What would that look like?



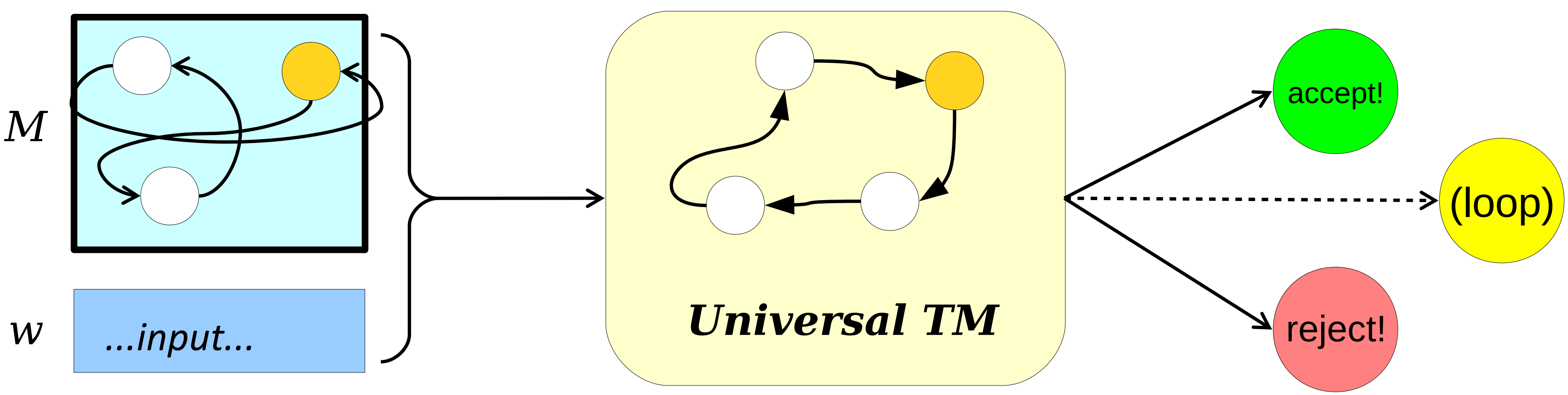
A TM Simulator

- It is known that anything that can be done with an unbounded-memory computer can be done with a TM.
- This means that there must be some TM that has the behavior of this simulateTM method.
- What would that look like?



A TM Simulator

- It is known that anything that can be done with an unbounded-memory computer can be done with a TM.
- This means that there must be some TM that has the behavior of this simulateTM method.
- What would that look like?



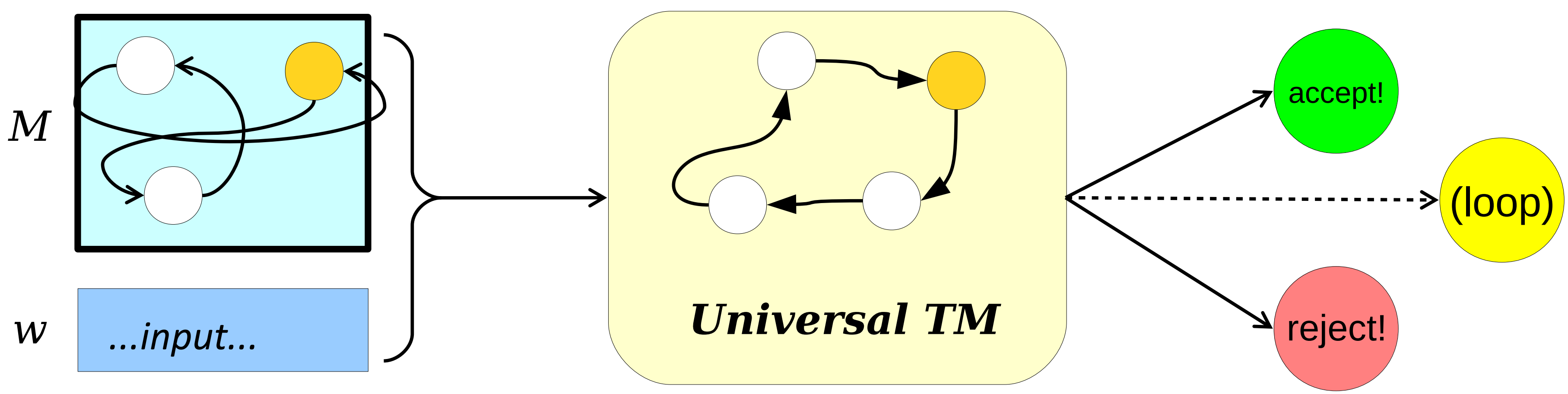
The Universal Turing Machine

- ***Theorem (Turing, 1936)****:* There is a Turing machine $\mathbf{U_{TM}}$ called the ***universal Turing machine*** that, when run on an input of the form $\langle M, w\rangle$, where M is a Turing machine and w is a string, simulates M running on w and does whatever M does on w (accepts, rejects, or loops).
- The observable behavior of U_{TM} is the following:
  - If M accepts w, then U_{TM} accepts $\langle M, w\rangle$.
  - If M rejects w, then U_{TM} rejects $\langle M, w\rangle$.
  - If M loops on w, then U_{TM} loops on $\langle M, w\rangle$.

M does to w

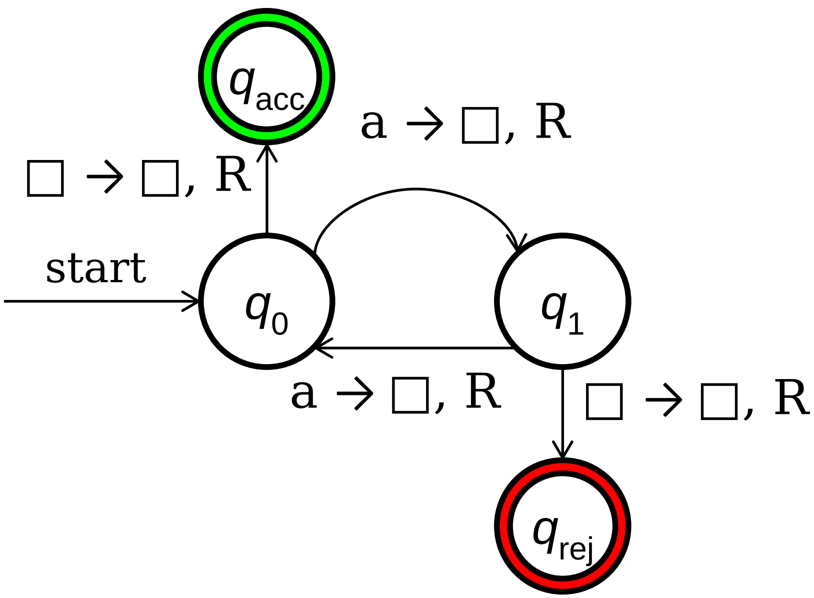
what

U_{TM} does to $\langle M, w\rangle$.

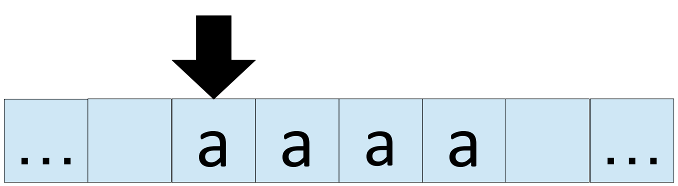


U_{TM}, Schematically

Machine M



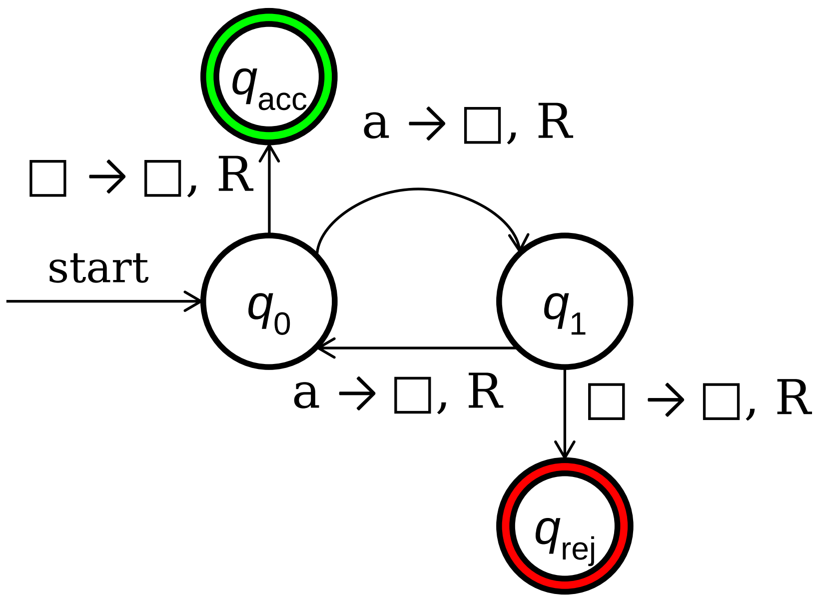
Input w



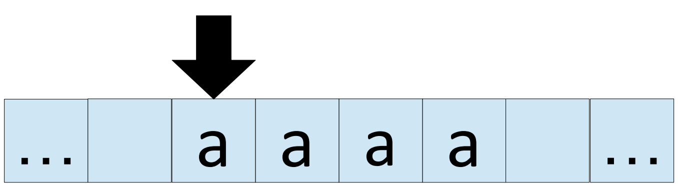
Imagine you have some machine *M* (like a program) that you want to run on input *w.*

U_{TM}, Schematically

Machine *M*

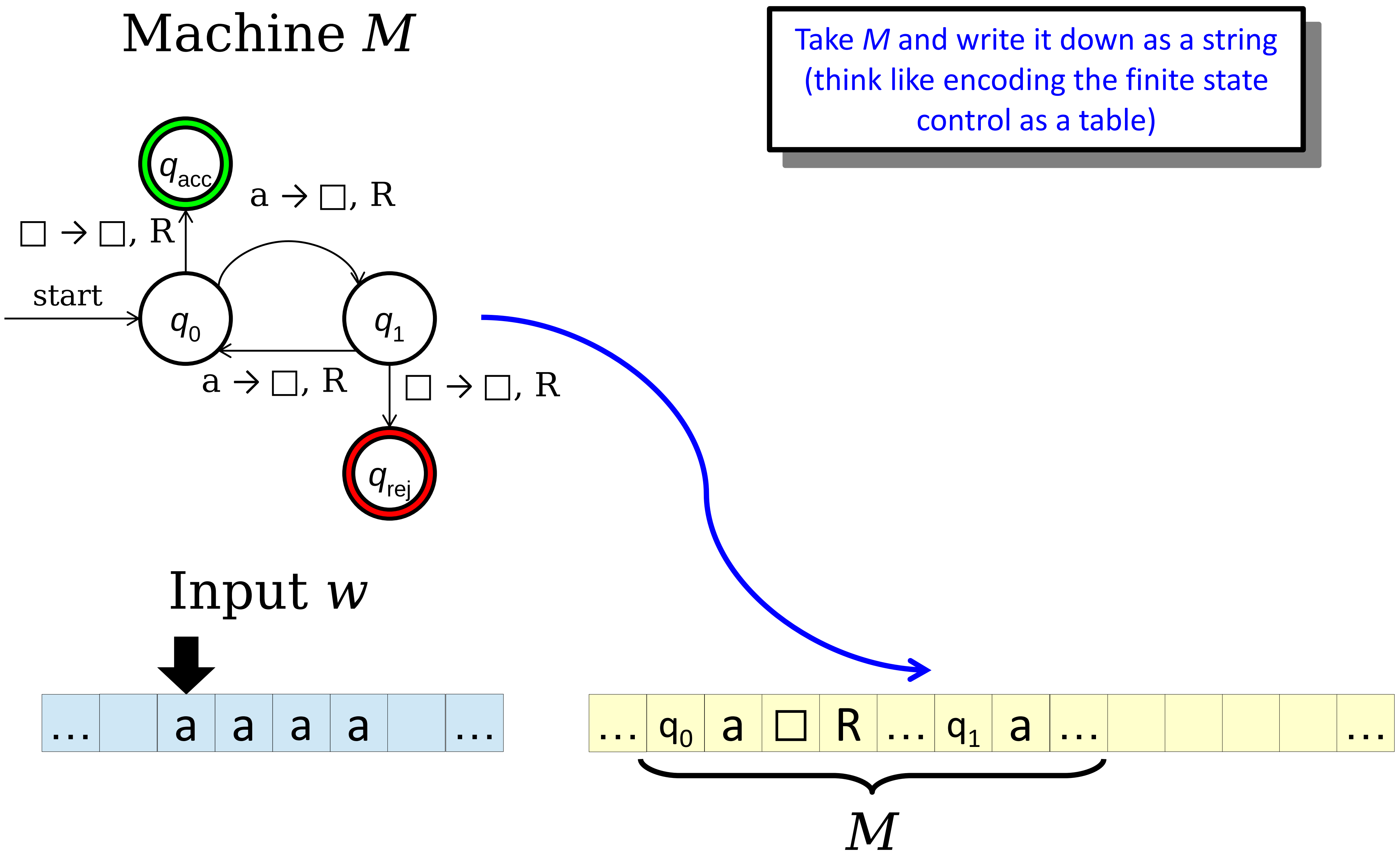


Input *w*



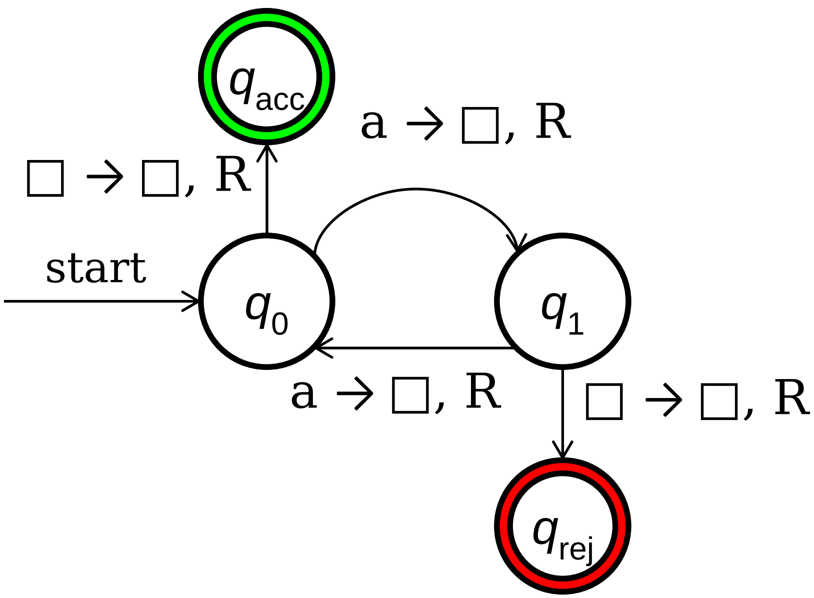
Take *M* and write it down as a string (think like encoding the finite state control as a table)

U_{TM}, Schematically



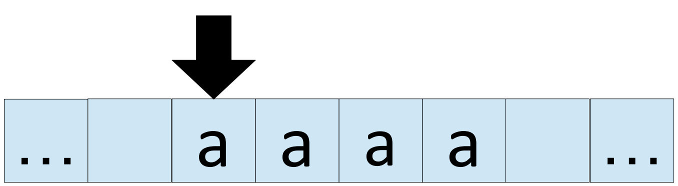
U_{TM}, Schematically

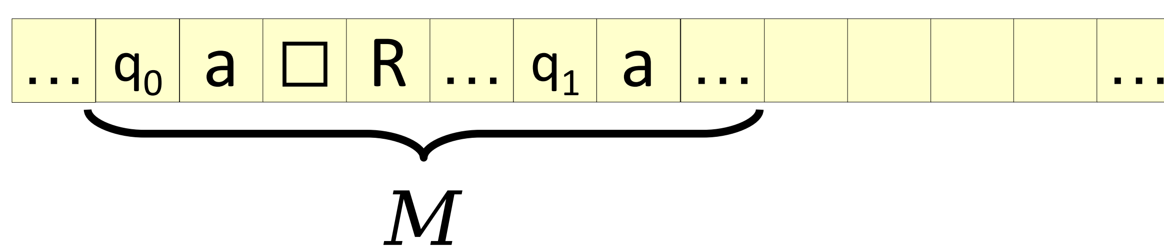
Machine M



Now take your input w and write it down too.

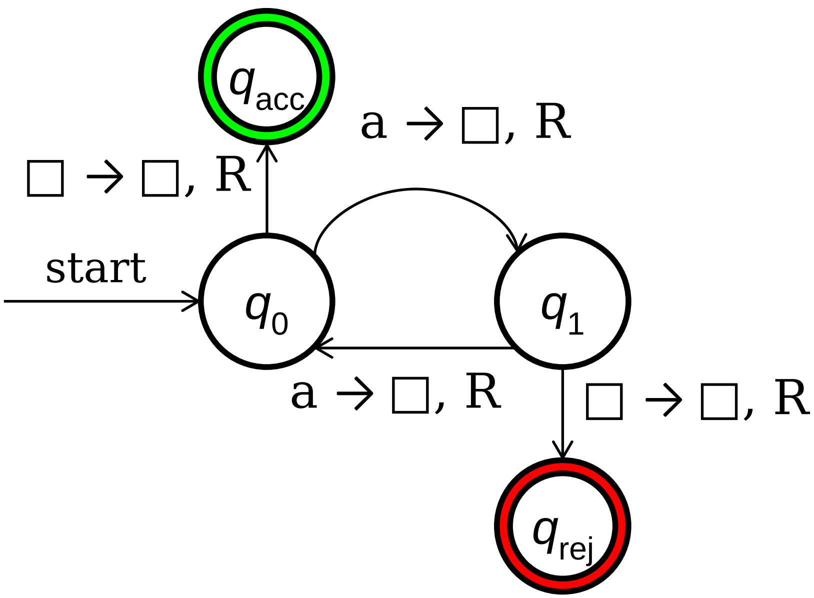
Input w





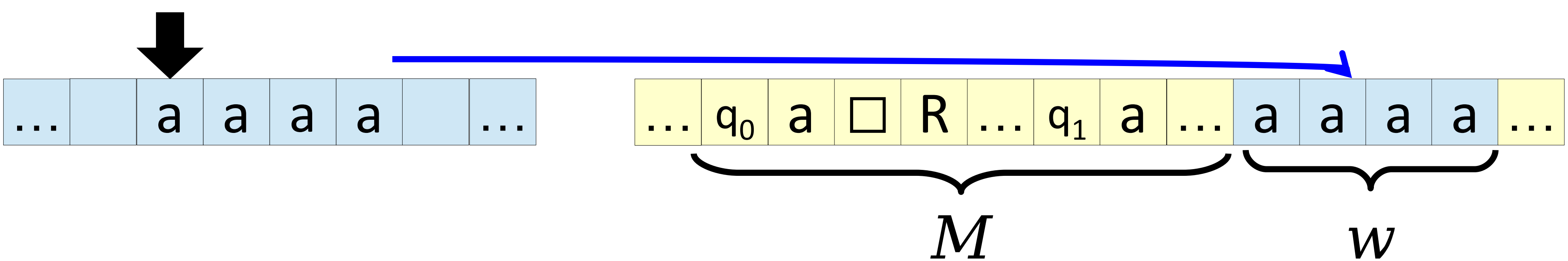
U_{TM}, Schematically

Machine *M*



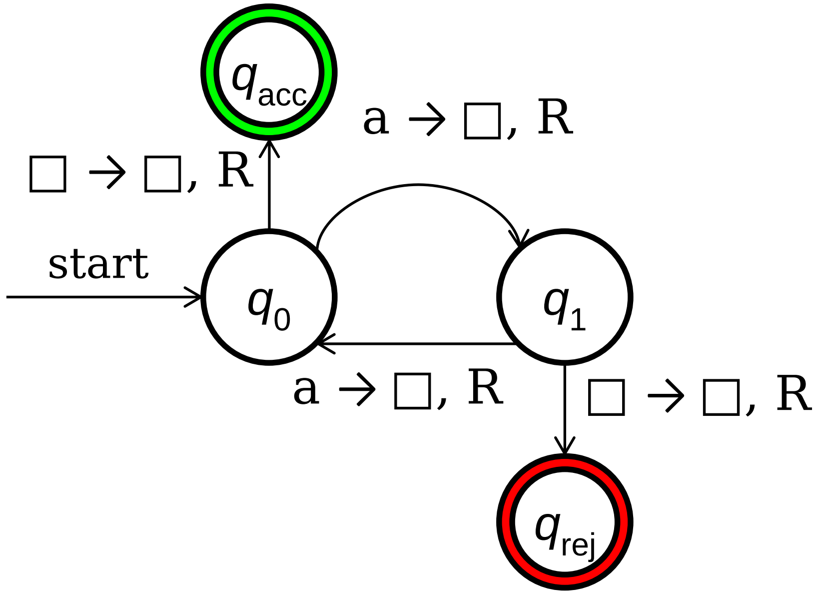
Now take your input *w* and write it down too.

Input *w*



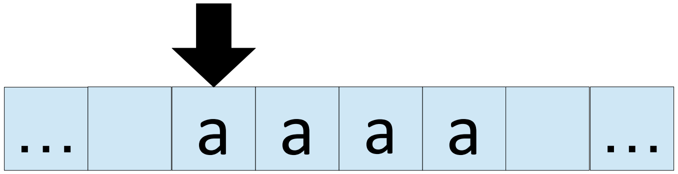
U_{TM}, Schematically

Machine M

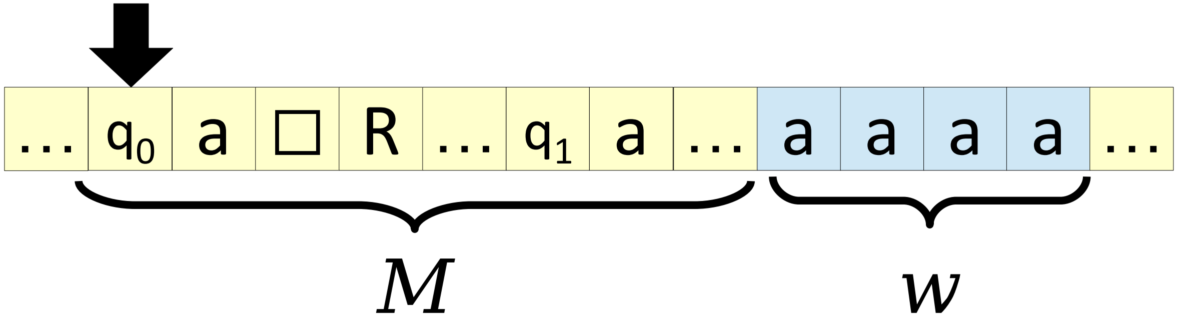


Feed this into U_{TM}.

Input w

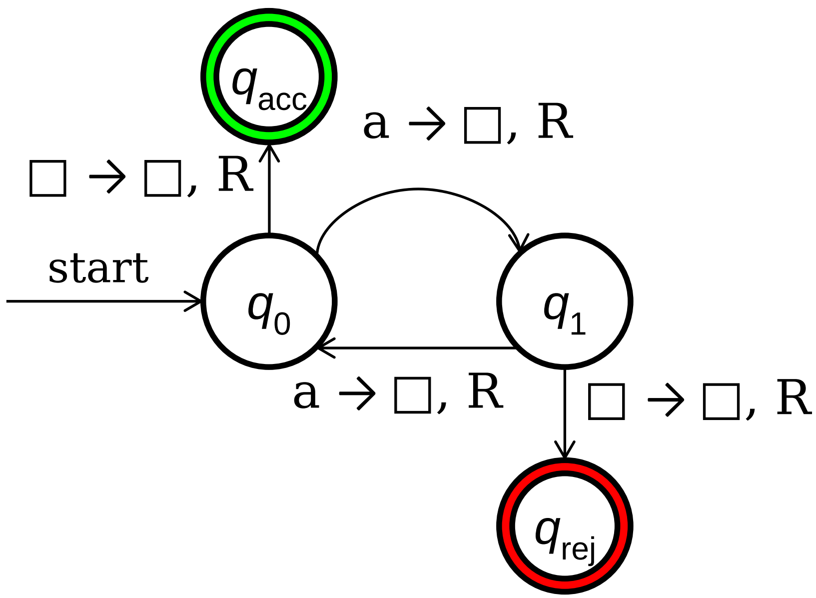


Input ⟨M, w⟩

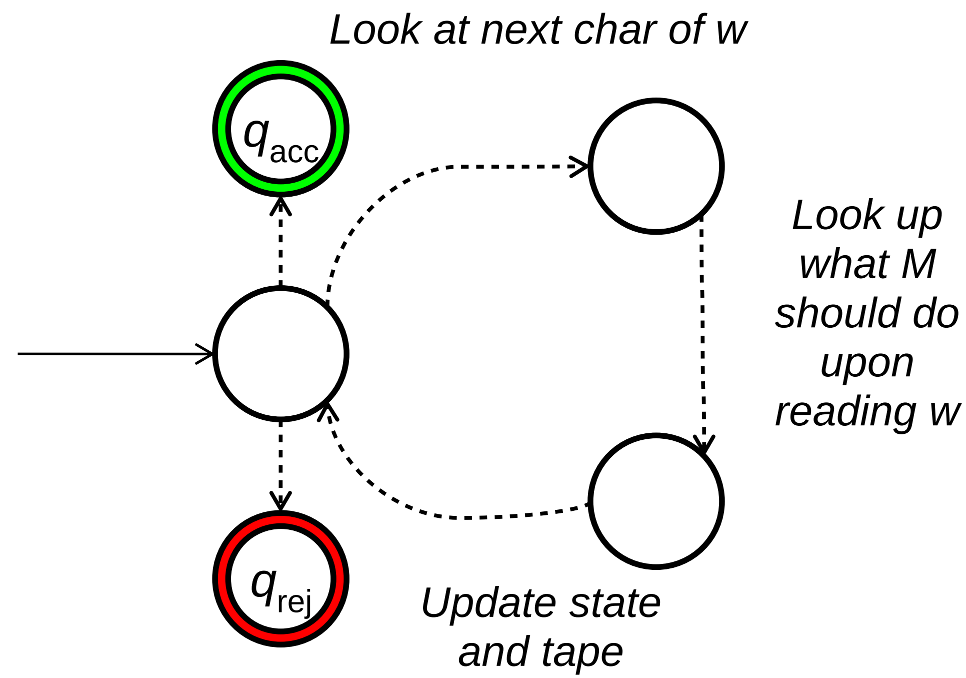


U_{TM}, Schematically

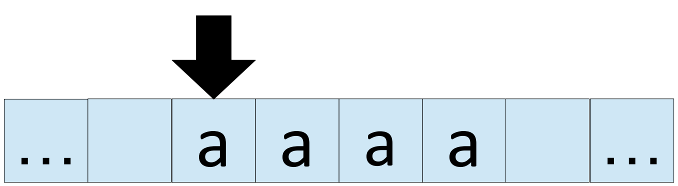
Machine M



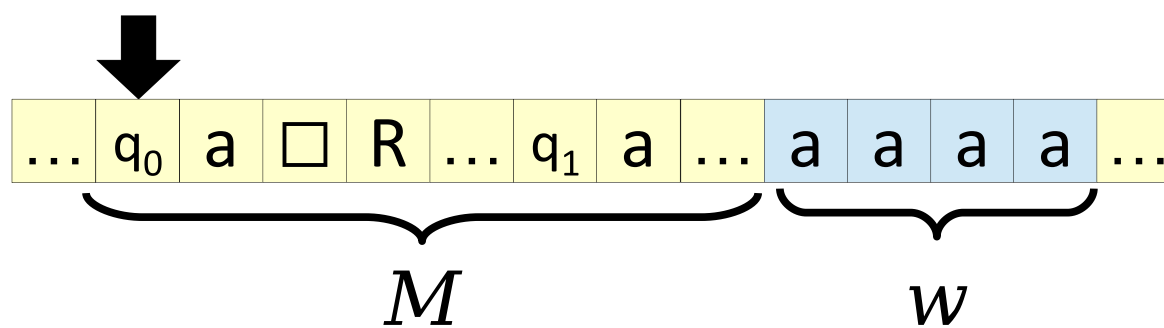
U_{TM}



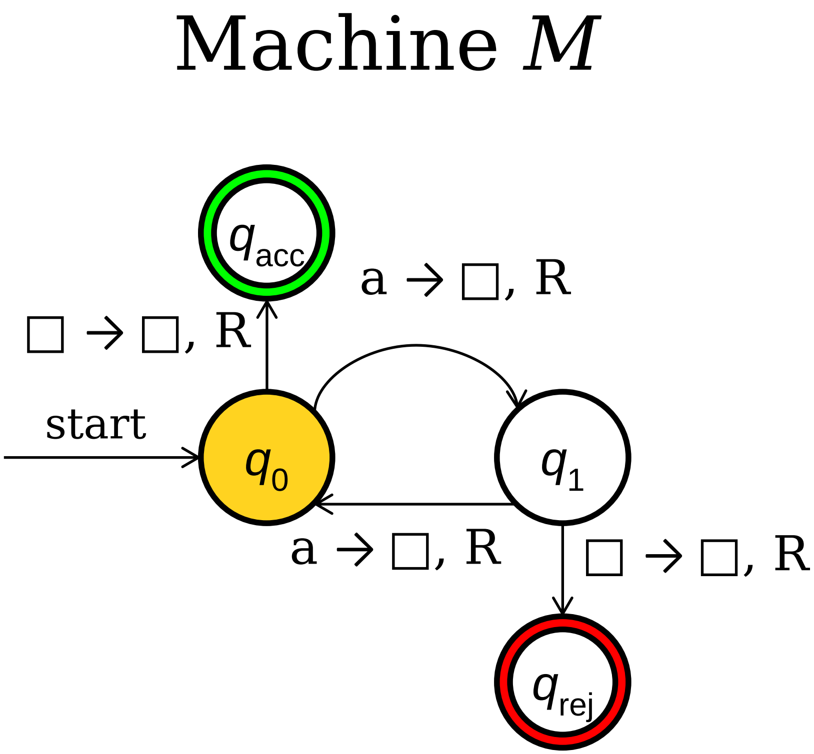
Input w

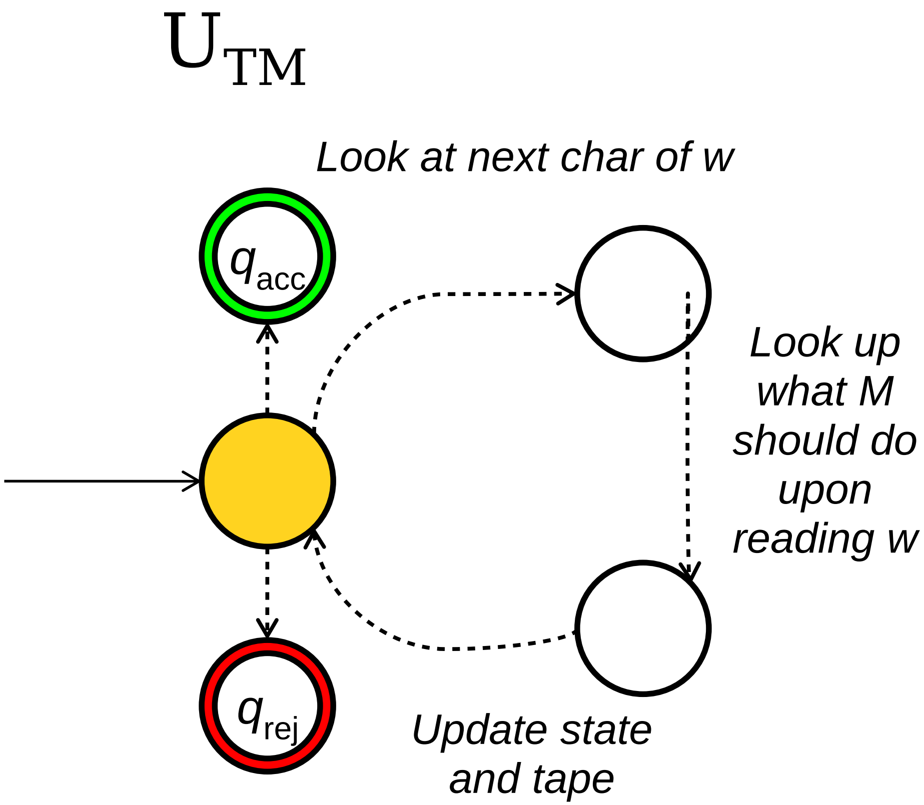


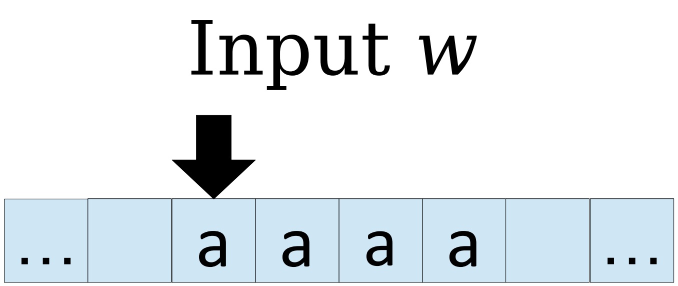
Input $\langle M, w \rangle$

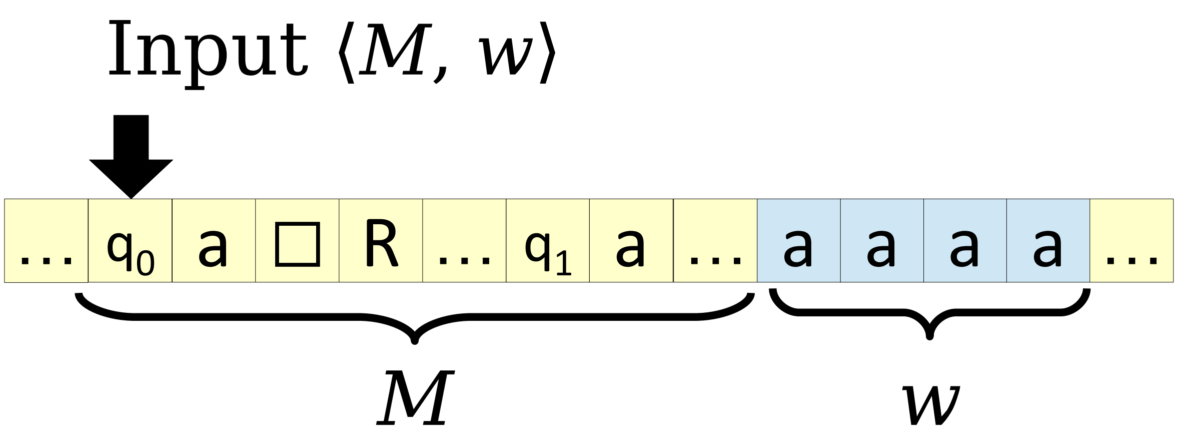


U_{TM}, Schematically

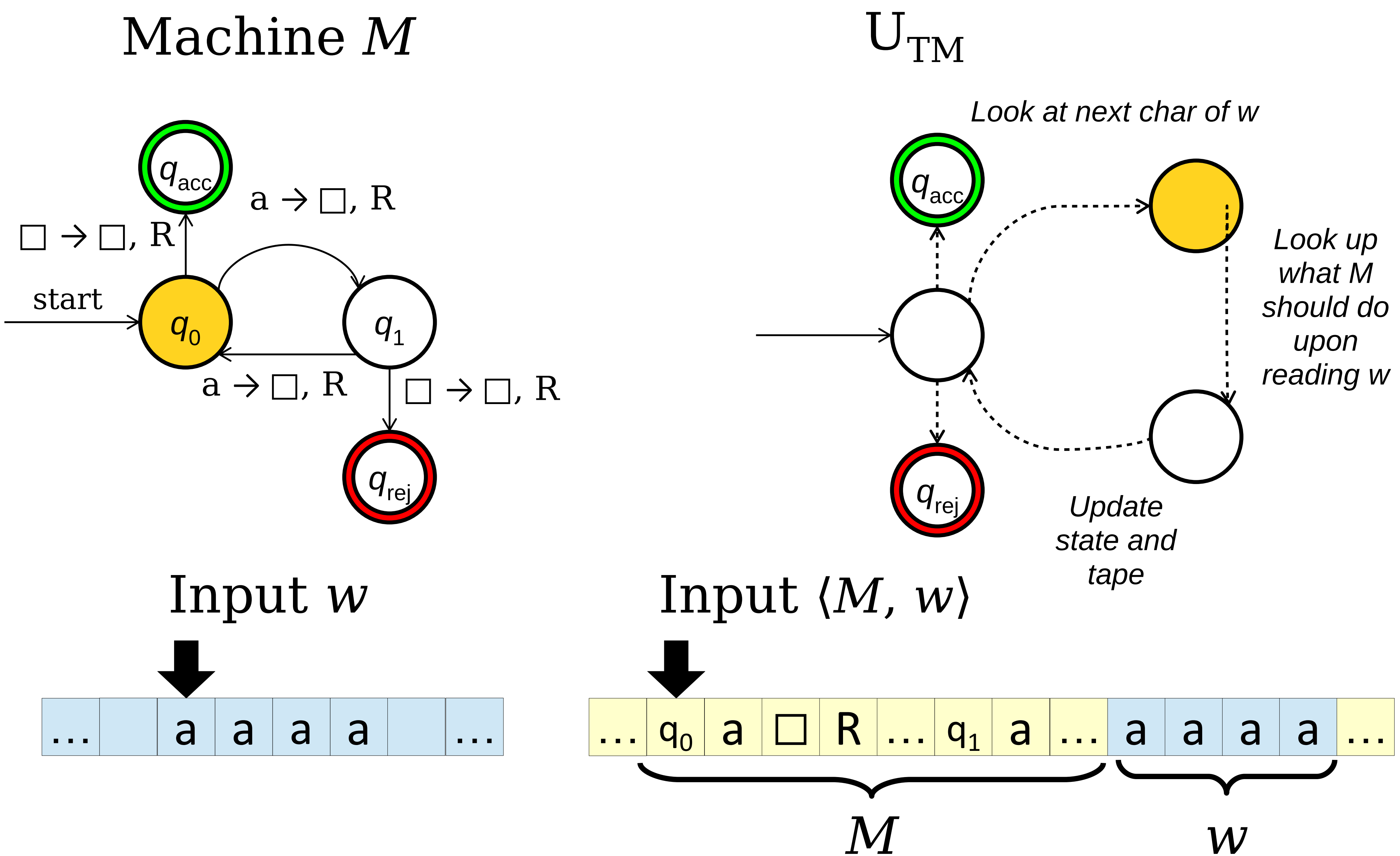






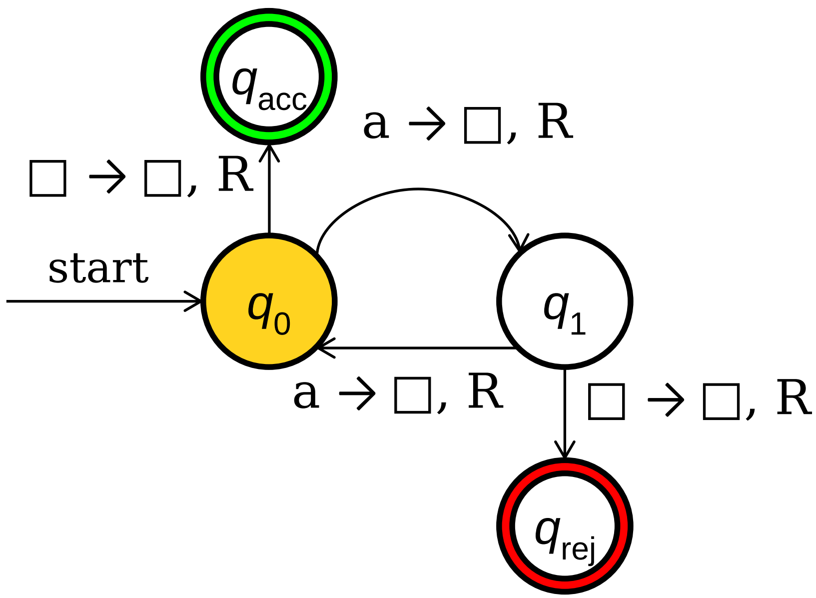


U_{TM}, Schematically

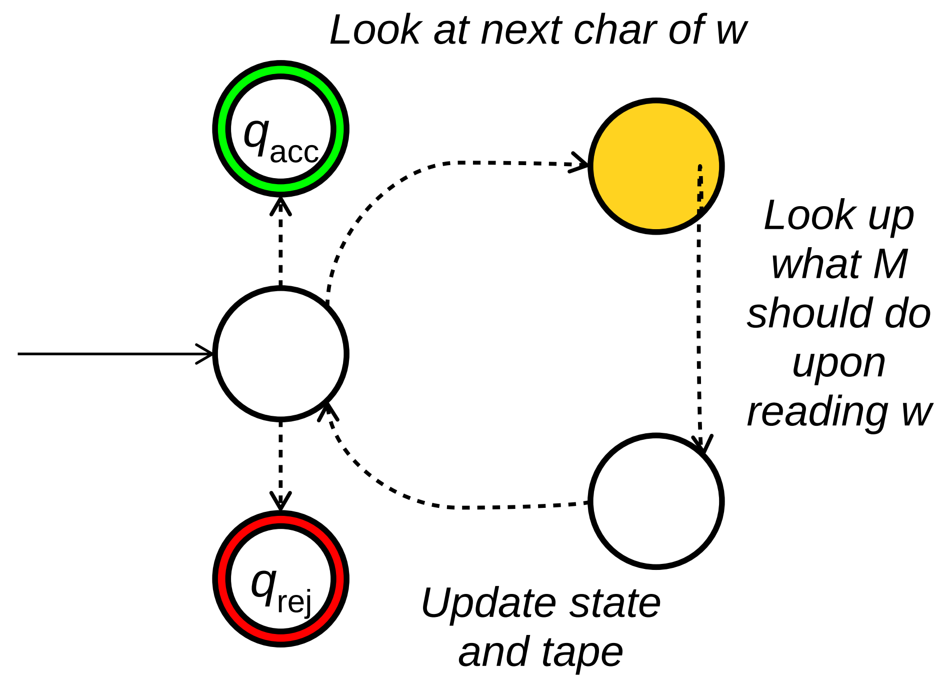


U_{TM}, Schematically

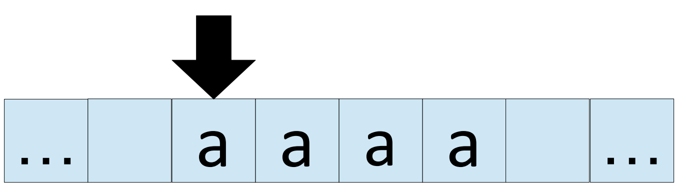
Machine M



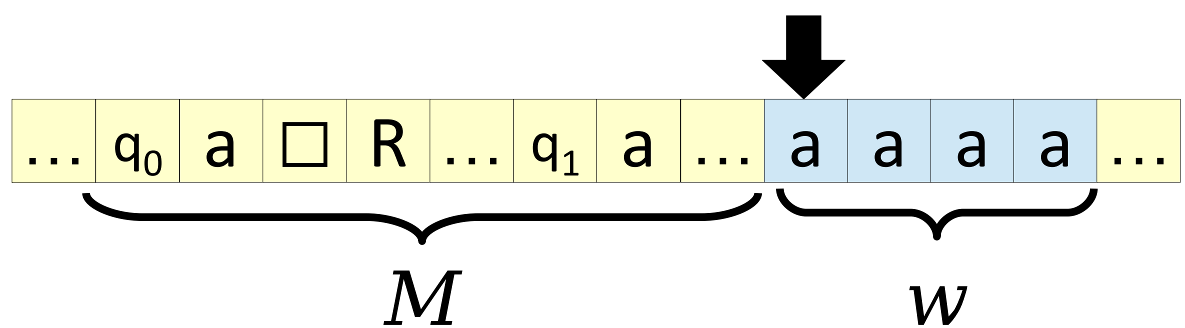
U_{TM}



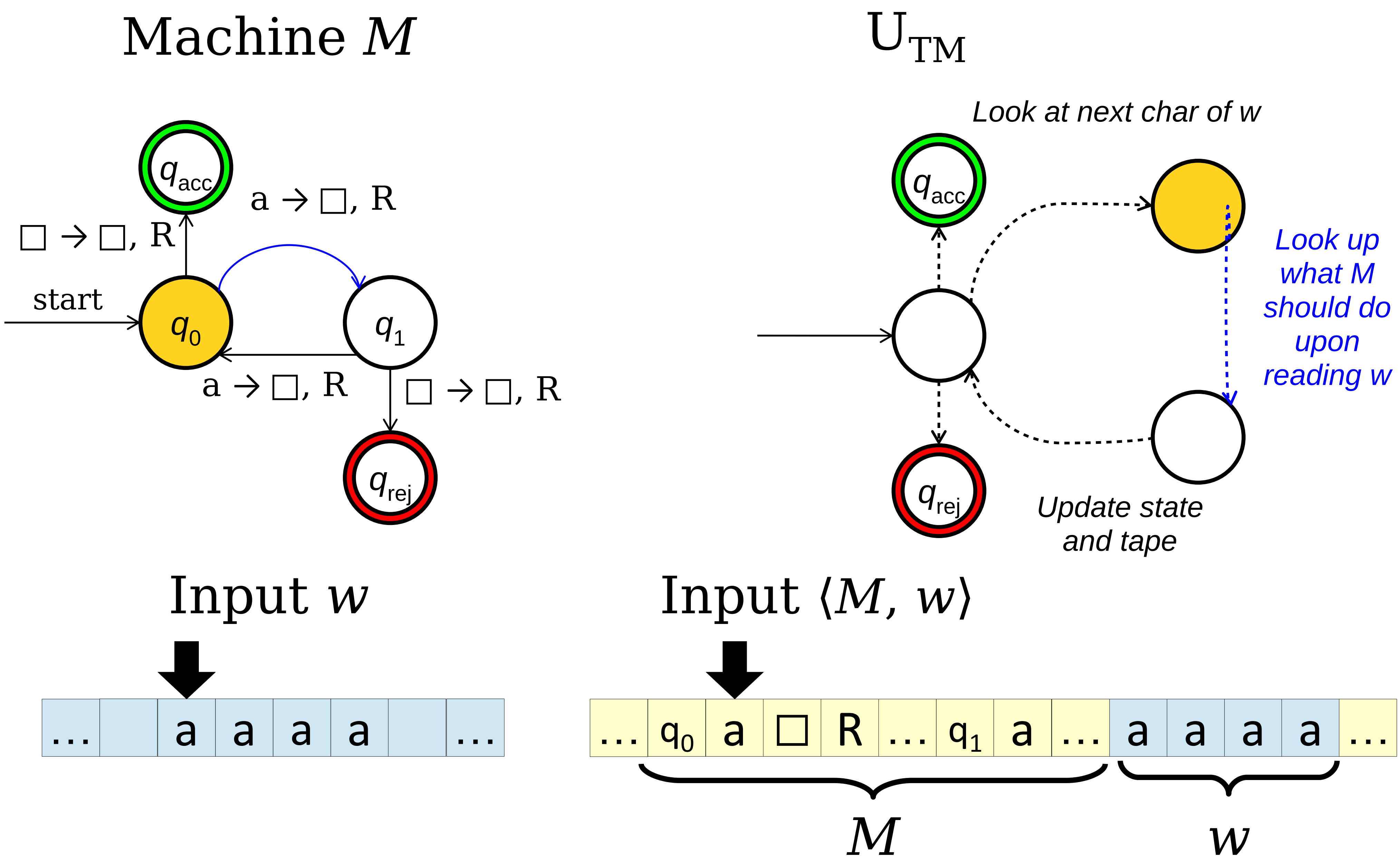
Input w



Input $\langle M, w \rangle$

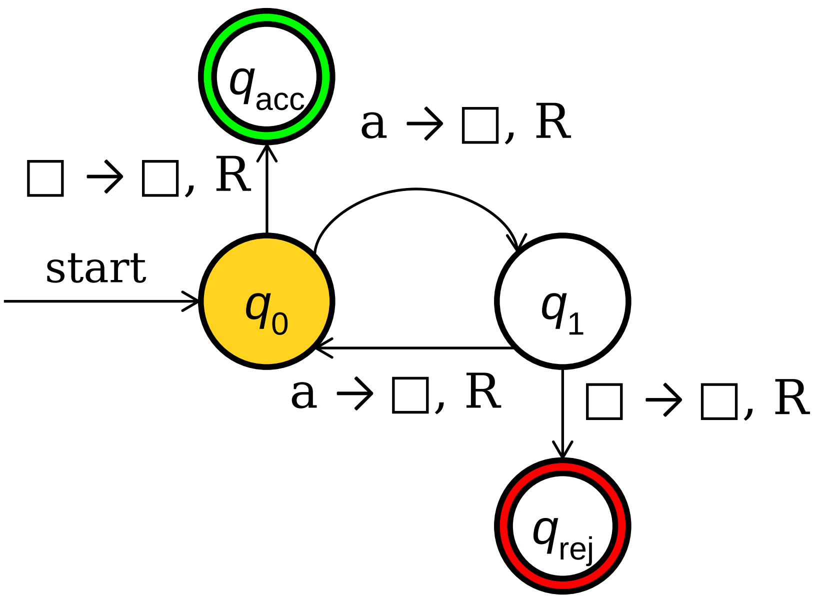


U_{TM}, Schematically

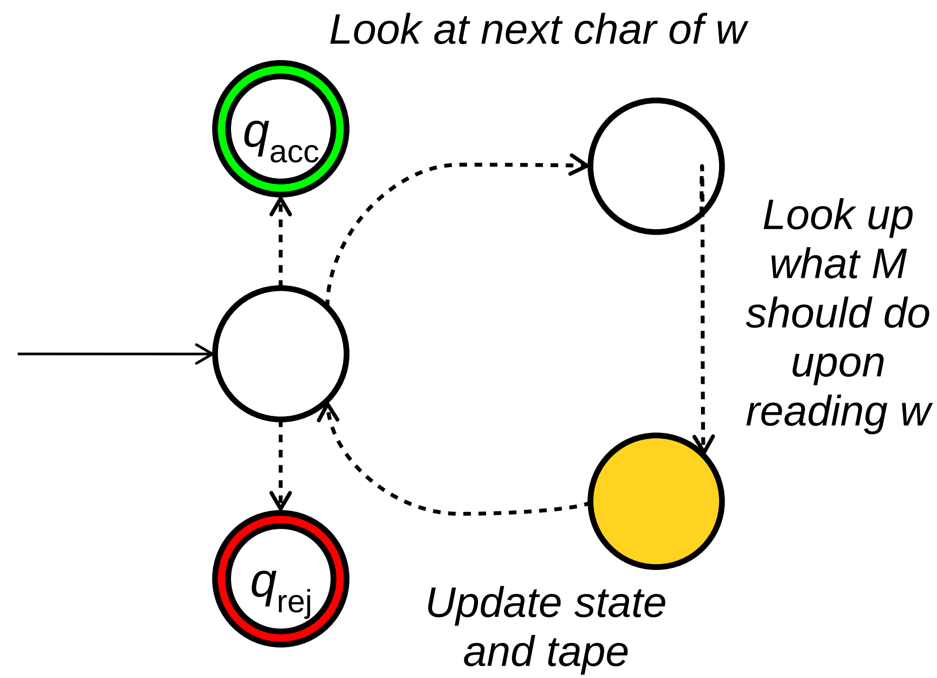


U_{TM}, Schematically

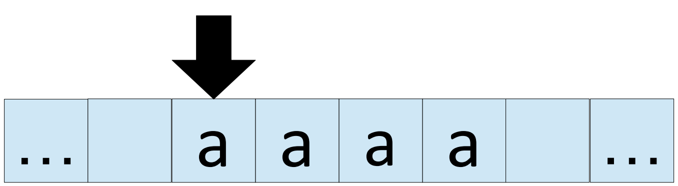
Machine M



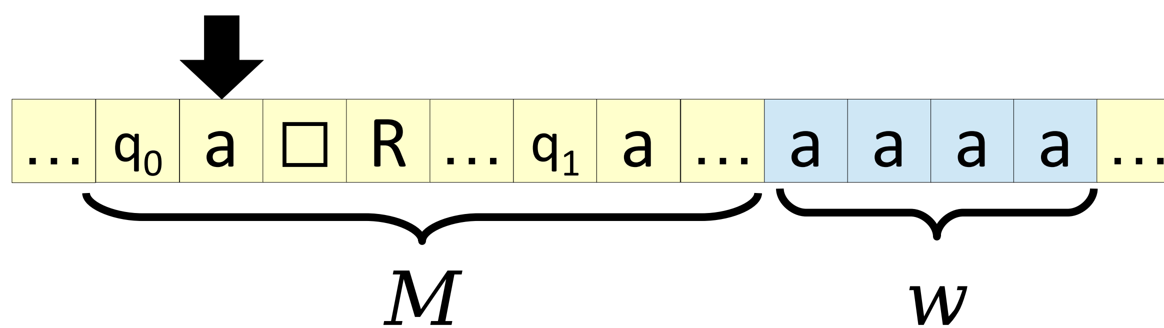
U_{TM}



Input w

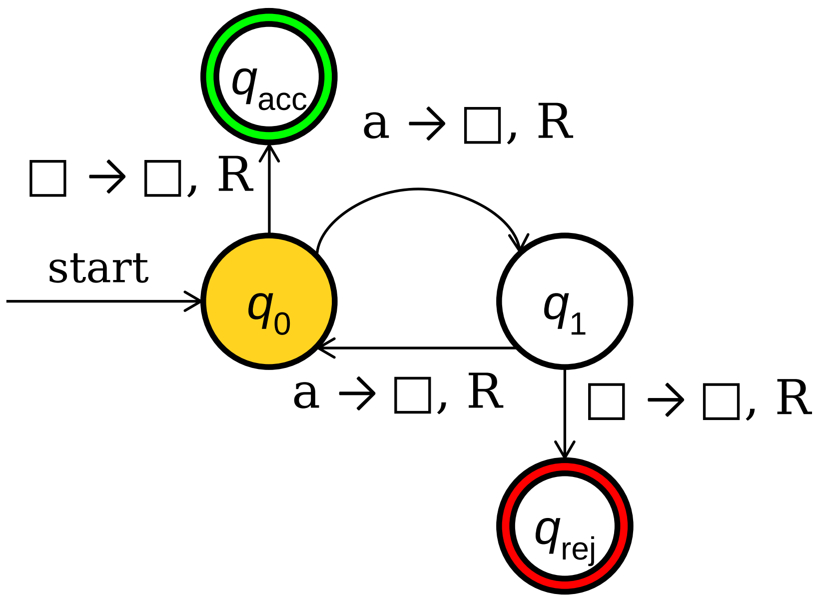


Input ⟨M, w⟩

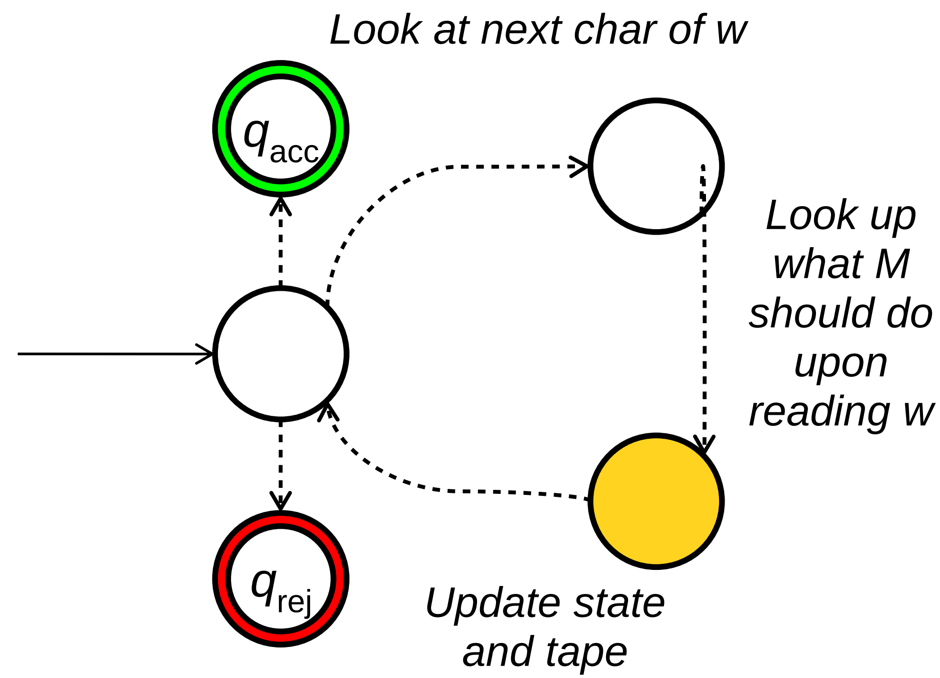


U_{TM}, Schematically

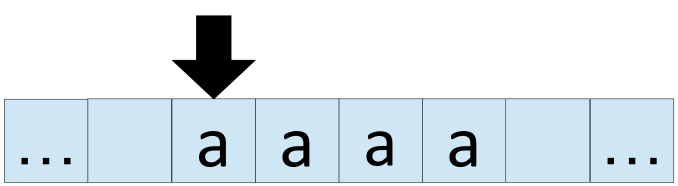
Machine M



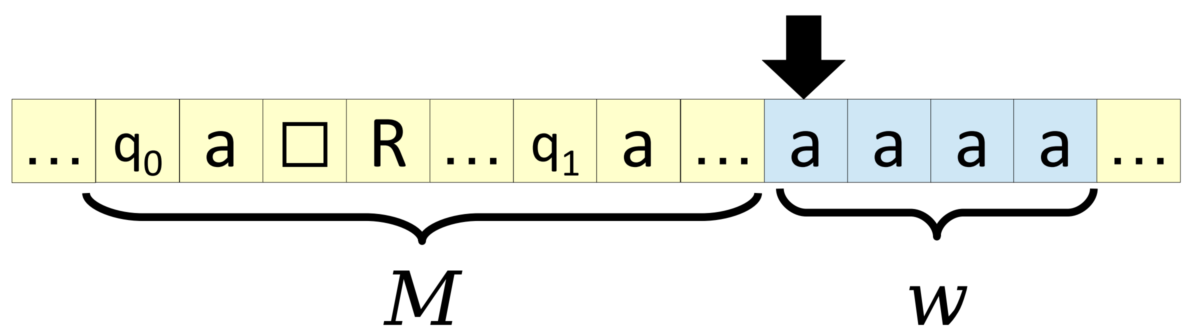
U_{TM}



Input w

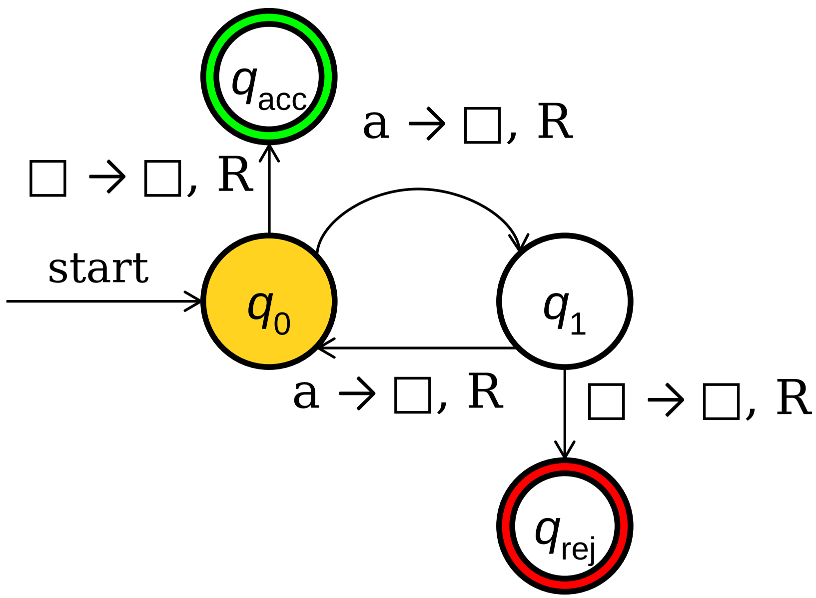


Input ⟨M, w⟩

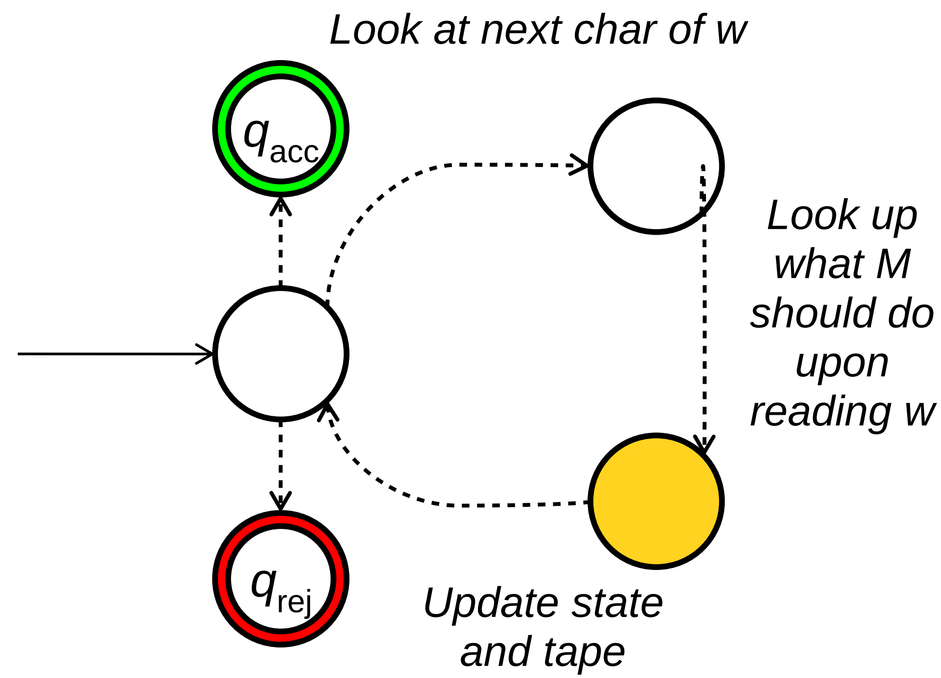


U_{TM}, Schematically

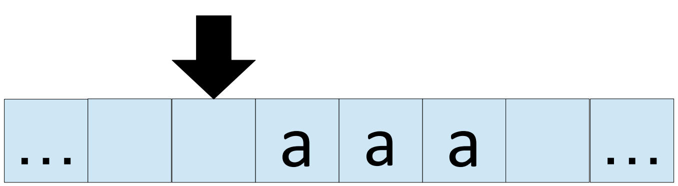
Machine M



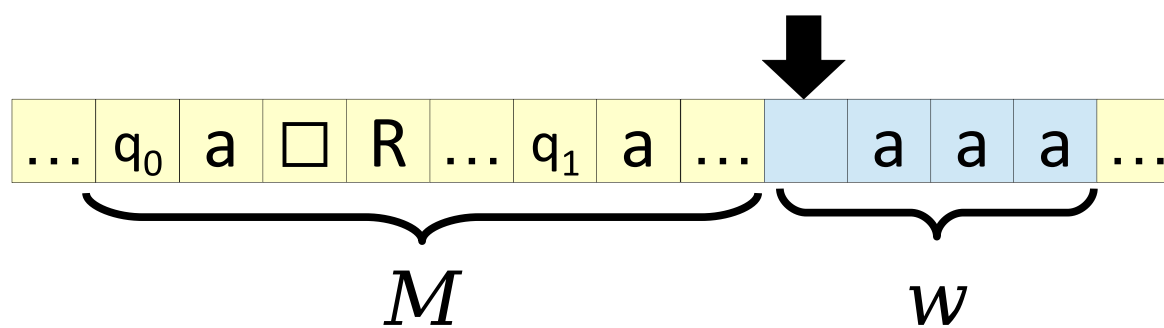
U_{TM}



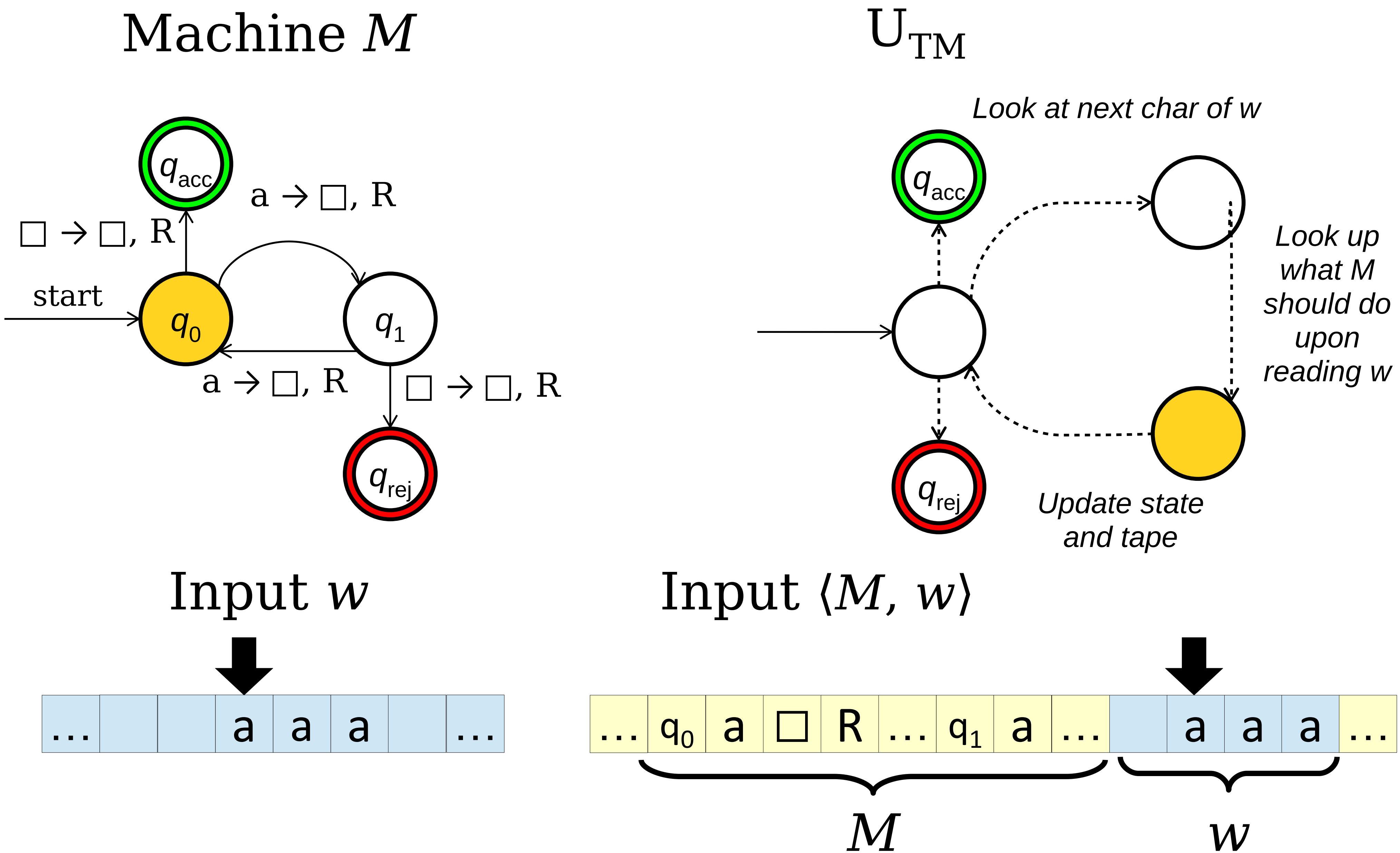
Input w



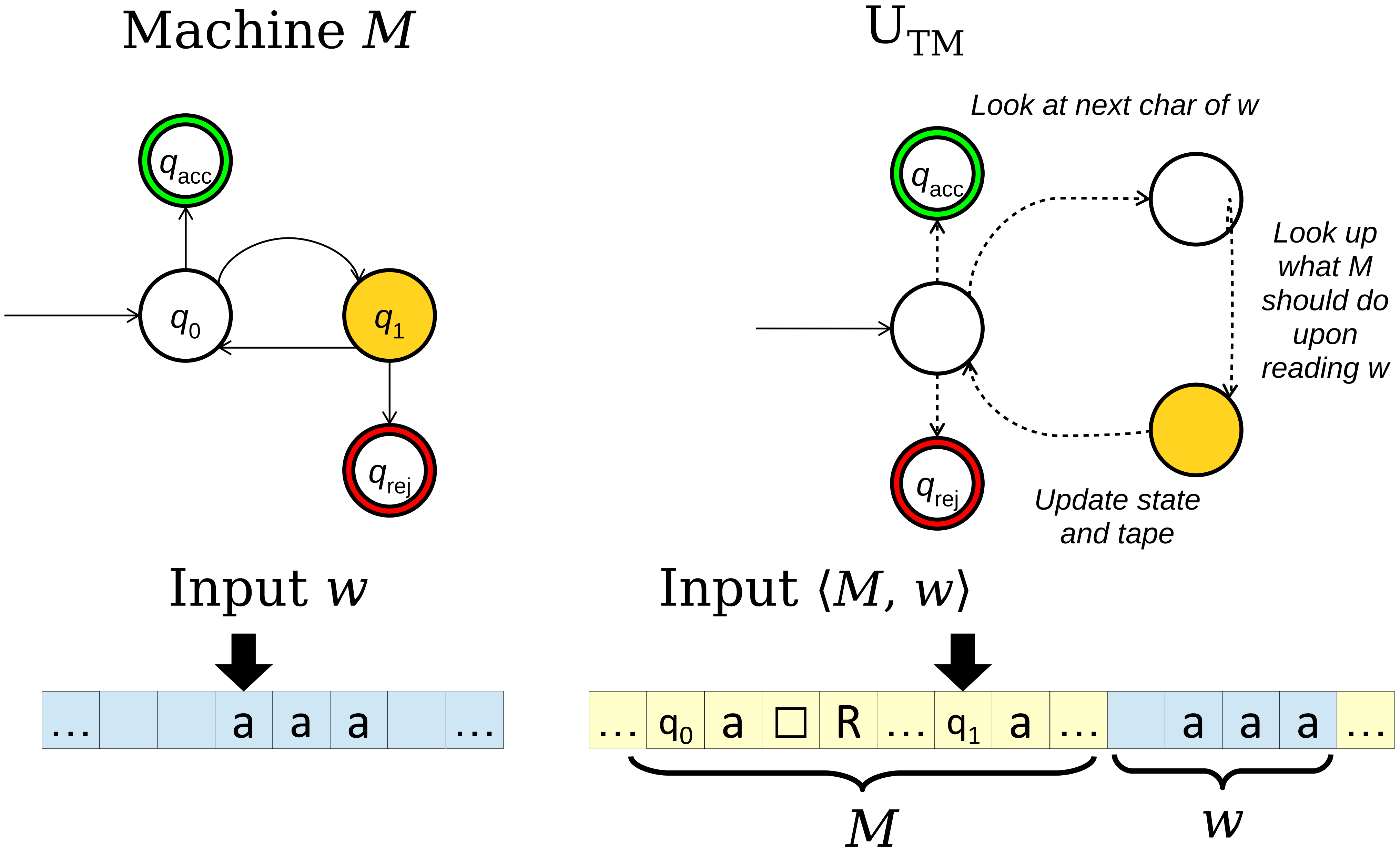
Input $\langle M, w \rangle$



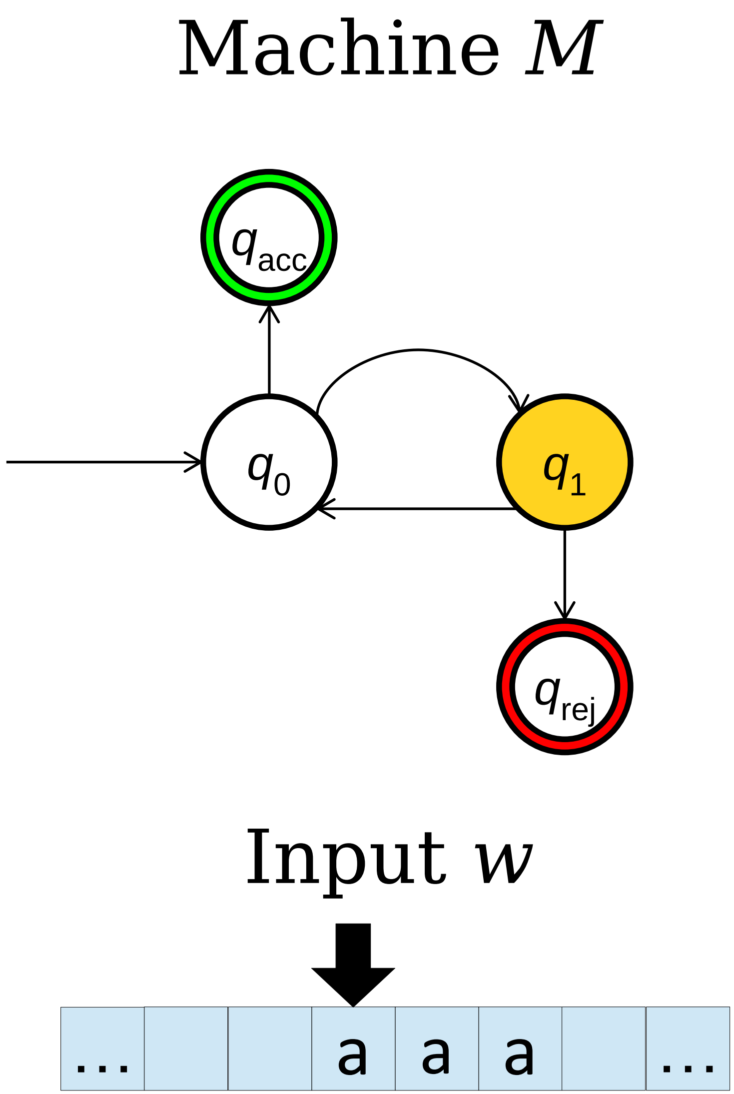
U_{TM}, Schematically

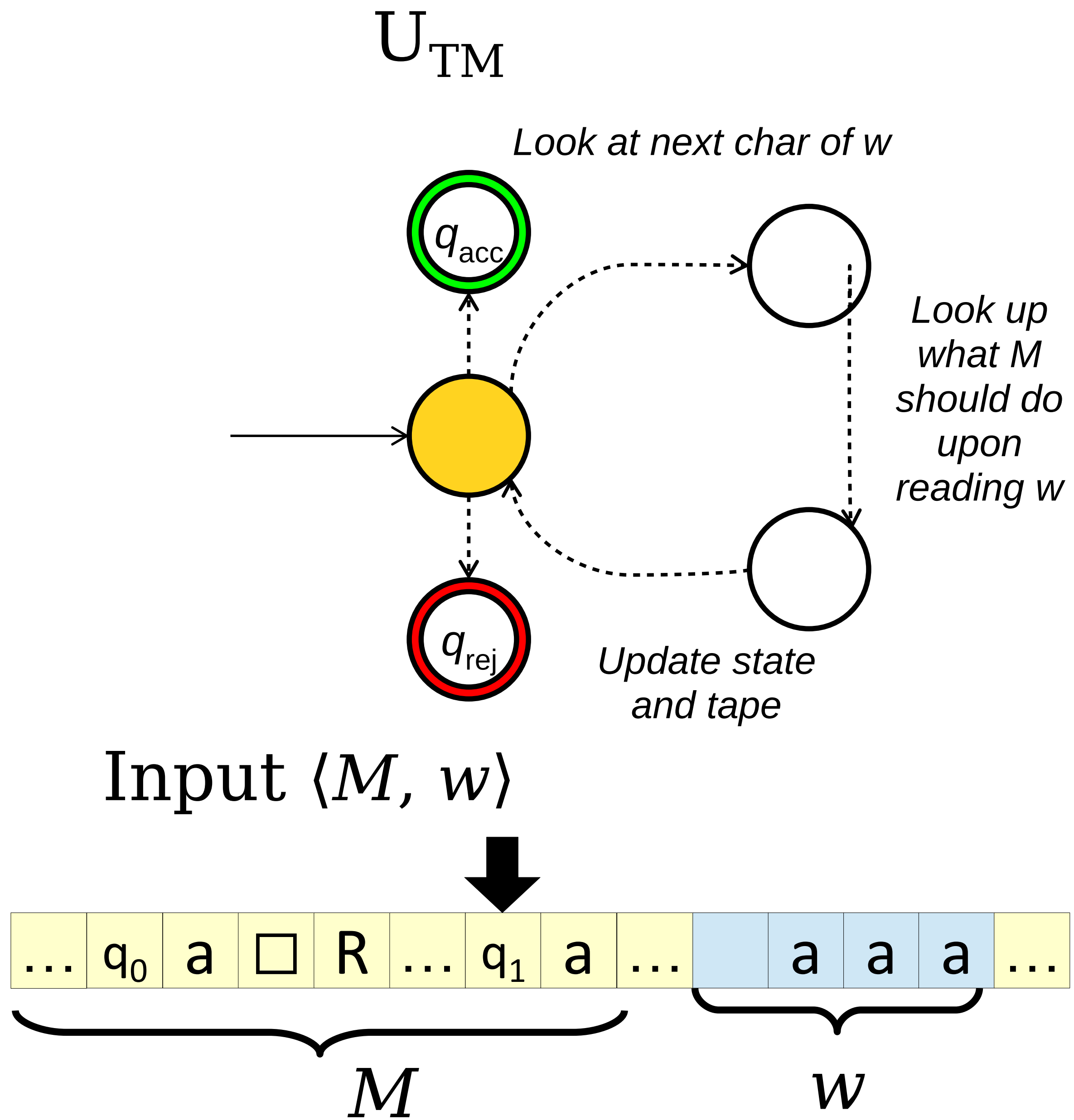


U_{TM}, Schematically



U_{TM}, Schematically





Since U_{TM} is a TM, it has a language.

What is the language of the universal Turing machine?

The Language of U_{TM}

Recall that the language of a TM is the set of all strings that TM accepts.

U_{TM}, when run on a string $\langle M, w \rangle$, where M is a TM and w is a string, will

… accept $\langle M, w \rangle$ if M accepts w,

… reject $\langle M, w \rangle$ if M rejects w, and

… loop on $\langle M, w \rangle$ if M loops on w.

The Language of U_{TM}

Recall that the language of a TM is the set of all strings that TM accepts.

U_{TM}, when run on a string $\langle M, w \rangle$, where M is a TM and w is a string, will

… accept $\langle M, w \rangle$ if M accepts w,

~~… reject $\langle M, w \rangle$ if M rejects w, and~~

~~… loop on $\langle M, w \rangle$ if M loops on w.~~

$$\mathscr{L}(U_{TM}) = \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \}$$

$$= \{ \langle M, w \rangle \mid M \text{ is a TM and } w \in \mathscr{L}(M) \}$$

The Language A_{TM}

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

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

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

Useful fact:

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

Because $A_{TM} = \mathscr{L}(U_{TM})$, we know that $A_{TM} \in$ **RE**.

Great Question to Ponder

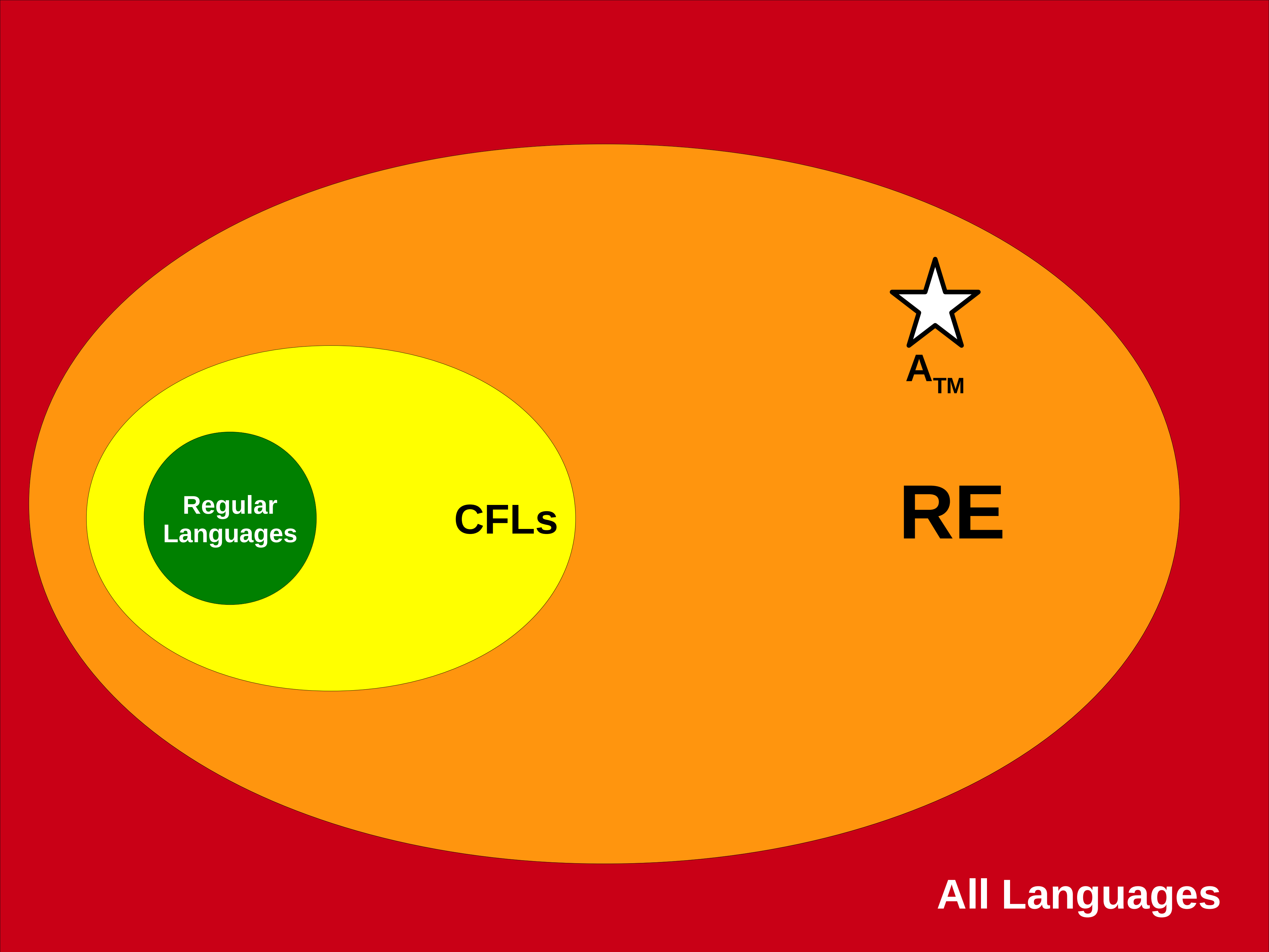
Simplify this expression:

$$\langle U_{TM}, \langle U_{TM}, \langle U_{TM}, \langle U_{TM}, \langle M, w \rangle\rangle\rangle\rangle\rangle \in A_{TM}.$$

If you can do this, you probably understand how things fit together.

If you're having trouble, no worries! It might be easier to start with this expression:

$$\langle U_{TM}, \langle M, w \rangle\rangle \in A_{TM}.$$



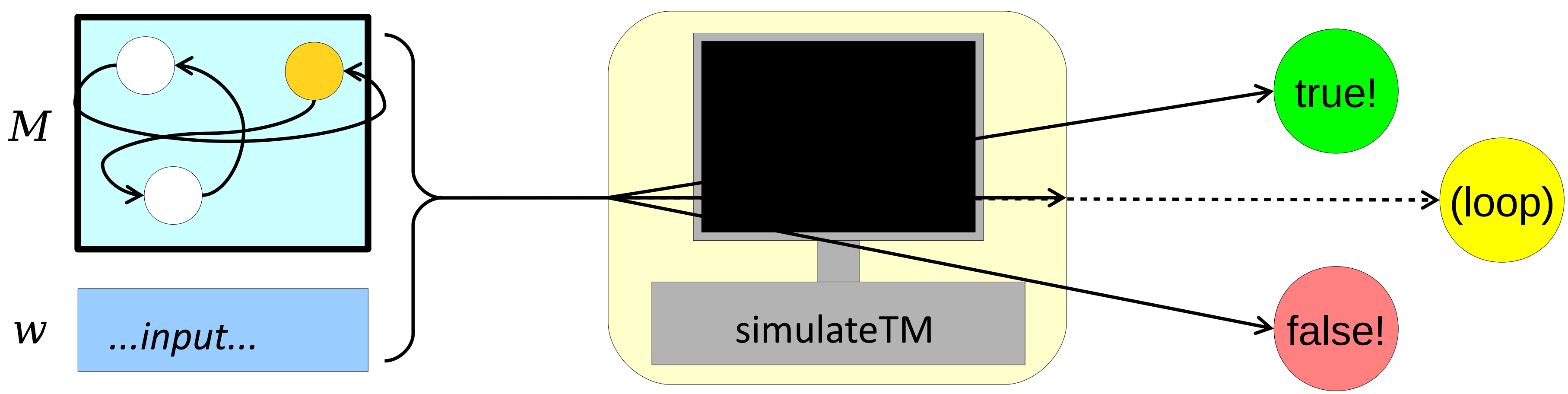
A_{TM}
Regular Languages
CFLs
RE
All Languages

Uh... so what?

Universality of computation has
practical consequences.

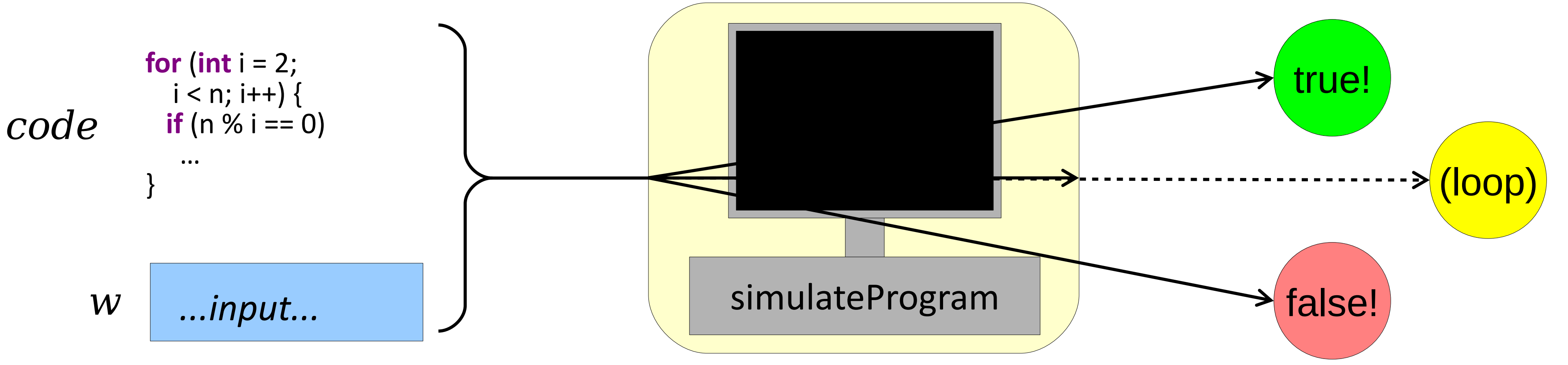
Why Does This Matter?

- The existence of a universal Turing machine has both theoretical and practical significance.
- For a practical example, let's review this diagram from before.
- Previously we replaced the *computer* with a TM. (This gave us the universal TM.)
- What happens if we replace the *TM* with a computer program?



Why Does This Matter?

- The existence of a universal Turing machine has both theoretical and practical significance.
- For a practical example, let's review this diagram from before.
- Previously we replaced the *computer* with a TM. (This gave us the universal TM.)
- What happens if we replace the *TM* with a computer program?



Programs Simulating Programs

The fact that there's a universal TM, combined with the fact that computers can simulate TMs and vice-versa, means that it's possible to write a program that simulates other programs.

These programs go by many names:

- An ***interpreter***, like the Java Virtual Machine or most implementations of Python.
- A ***virtual machine***, like VMWare or VirtualBox, that simulates an entire computer.

Why Does This Matter?

- The key idea behind the universal TM is that idea that TMs can be fed as inputs into other TMs.
  - Similarly, an interpreter is a program that takes other programs as inputs.
  - Similarly, an emulator is a program that takes entire computers as inputs.
- This hits at the core idea that ***computing devices can perform computations on other computing devices.***

Time-Out for Announcements!

Problem Sets

- Problem Set Five is due tomorrow night.
- Late period extends this to Saturday.
- This is the last assignment you can take a late period on.
- If you submitted the CFG before we linked to it, you need to submit again.

Final Exam Logistics

- Our final exam is next Friday.
- The exam is cumulative. You’re responsible for topics from PS0 – PS6 and all of the lectures up through Unsolvable Problems (this Friday).
- The exam is the same style as the midterm. More details on a Campuswire post going up today.

Your Questions

"Why did you decide to coterm (versus double major, minor, enter industry before grad school, etc) and what do you wish you had known before coterming?"

Your Questions

“Tabs or spaces?”

Your Questions

“What made you interested in CS?”

Your Questions

Have more questions?

Go to **sli.do** and put in code **G517**.

The event is closed, but you can still click on it and add questions / vote.

Let’s take a five minute break!

Teaser #1:

This language A_{TM} has some interesting properties beyond what we've seen here.

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 ∈ {**a**, **b**}* | w has the same number of **a**'s and **b**'s }:

```
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:

```
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 $\{ \mathbf{a}^n\mathbf{b}^n \mid n \in \mathbb{N} \}$ might be represented in software as a method with this signature:

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

Similarly, a decider for $\{ \langle m, n \rangle \mid m, n \in \mathbb{N} \text{ and } m \text{ is a multiple of } n \}$ 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! ■

Self-Defeating Objects

The general template for proving that x is a self-defeating object is as follows:

- Assume that x exists.
- Construct some object $f(x)$ from x.
- Show that $f(x)$ has some impossible property.
- Conclude that x doesn't exist.

The particulars of what x and $f(x)$ are, and why $f(x)$ has an impossible property, depend on the specifics of the proof.

An Important Point

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

Claim: There is a largest integer.

Proof: Assume x is the largest integer.

Notice that $x > x - 1$.

So there's no contradiction. ■

How do we know there's no contradiction? We just checked one case.

Self-Defeating Objects

You ***cannot*** show that a self-defeating object x ***does exist*** by using this line of reasoning:

- Suppose that x exists.
- Construct some object $g(x)$ from x.
- Show that $g(x)$ has ***no*** undesirable properties.
- Conclude that x exists.

The fact that $g(x)$ has no bad properties doesn't mean that x exists. It just means you didn't look hard enough for a counterexample. 😃

Teaser #3:

Certain Turing machines can’t exist, as they’d be self-defeating objects.

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 $w \in \mathscr{L}(M)$.

How could we do this?

Observation:

If you want to know whether this is true…

$$w \in \mathscr{L}(M)$$

if and only if

M accepts w.

… you can try to determine whether this is true.

Learning About a String

Option 1: Run M on w.

What could happen?

- M could accept w. Great! We know $w \in \mathscr{L}(M)$.
- M could reject w. Great! We know $w \notin \mathscr{L}(M)$.
- M could loop on w. Hmmm. We've learned nothing.

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

Observation:

If you want to know whether this is true…

$$w \in \mathscr{L}(M)$$

if and only if

$$M \text{ accepts } w$$

if and only if

$$\langle M, w \rangle \in \mathrm{A}_{\mathrm{TM}}.$$

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

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

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! Th

What if we used a ***decider***, not a ***recognizer***?

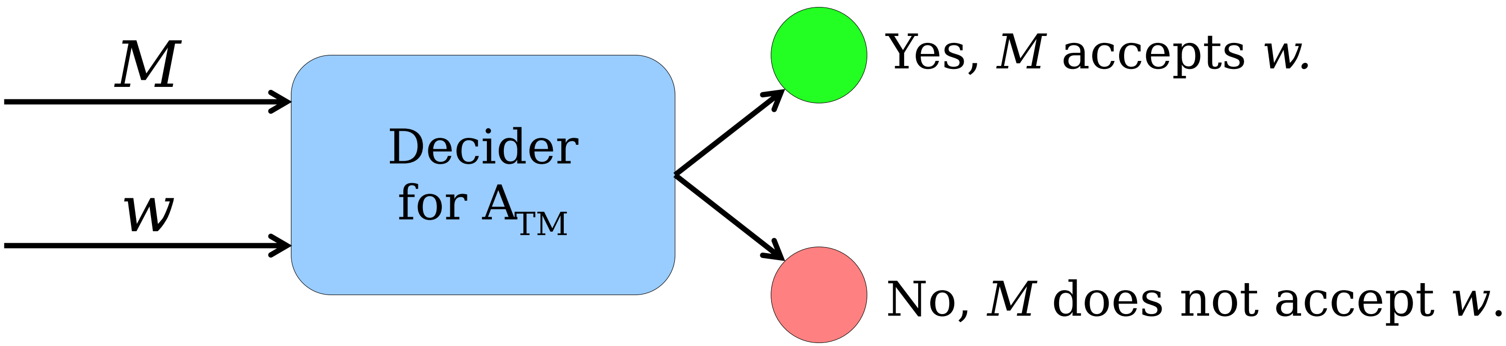
U_{TM} could reject $\langle M, w\rangle$. Great! Then $w \notin \mathscr{L}(M)$.

U_{TM} could loop on $\langle M, w\rangle$. Hmmm. We've learned nothing.

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

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 $w \in \mathscr{L}(M)$.

The decider could reject $\langle M, w\rangle$. Then $w \notin \mathscr{L}(M)$.

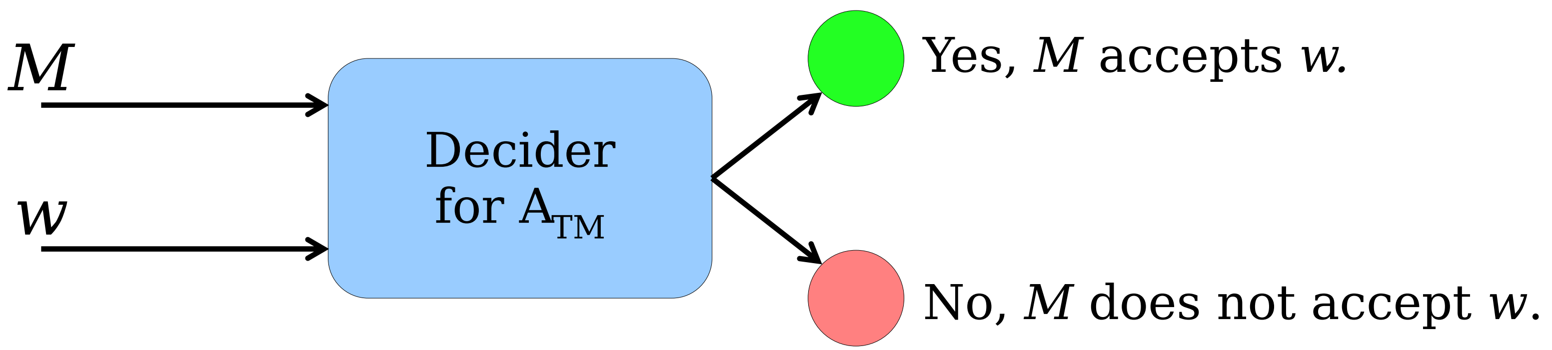
Question: How do we build this decider?

Claim: A decider for A_{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:



We could represent this decider in software as a method

```
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?

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

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();
        }
}
```

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?

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?

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?

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?

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?

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!

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!

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?

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?

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?

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?

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?

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?

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?

```
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?

```
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"

"Using n to get n + 1"

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! ■

```
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?

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 in

Notice that *n* <

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

Contradiction! ■-*ish*

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

int main() {
          string me = mySource();
          tInput();

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

Assume there exists this object *x* which has these properties that are too powerful to actually work.

What does this program do?

Use the purported properties of *x* against itself to create a contradiction.

Theorem: The ... 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*

```
... 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?

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 i

Notice that $n < n + 1$.

But then n isn't the largest integer.

Contradiction! ■-*ish*

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

Thus, this object *x* cannot exist!

```
                          ource();
                          tInput();

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

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*

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

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

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

Theorem: $A_{TM} \notin \mathbf{R}$.

Theorem: $A_{TM} \notin \mathbf{R}$.

Proof: By contradiction; assume that $A_{TM} \in \mathbf{R}$.

Theorem: $A_{TM} \notin \mathbf{R}$.

Proof: By contradiction; assume that $A_{TM} \in \mathbf{R}$. 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.

Theorem: $A_{TM} \notin \mathbf{R}$.

Proof: By contradiction; assume that $A_{TM} \in \mathbf{R}$. 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*:

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

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

Theorem: $A_{TM} \notin \mathbf{R}$.

Proof: By contradiction; assume that $A_{TM} \in \mathbf{R}$. 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:

```
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.

Theorem: $A_{TM} \notin \mathbf{R}$.

Proof: By contradiction; assume that $A_{TM} \in \mathbf{R}$. 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:

```
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.

Theorem: $A_{TM} \notin \mathbf{R}$.

Proof: By contradiction; assume that $A_{TM} \in \mathbf{R}$. 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:

```
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.

Theorem: $A_{TM} \notin \mathbf{R}$.

Proof: By contradiction; assume that $A_{TM} \in \mathbf{R}$. 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:

```
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.

Theorem: $A_{TM} \notin \mathbf{R}$.

Proof: By contradiction; assume that $A_{TM} \in \mathbf{R}$. 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:

```
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.

Theorem: $A_{TM} \notin \mathbf{R}$.

Proof: By contradiction; assume that $A_{TM} \in \mathbf{R}$. 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:

```
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: $A_{TM} \notin \mathbf{R}$.

Proof: By contradiction; assume that $A_{TM} \in \mathbf{R}$. 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:

```
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, $A_{TM} \notin \mathbf{R}$.

Theorem: $A_{TM} \notin \mathbf{R}$.

Proof: By contradiction; assume that $A_{TM} \in \mathbf{R}$. 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.

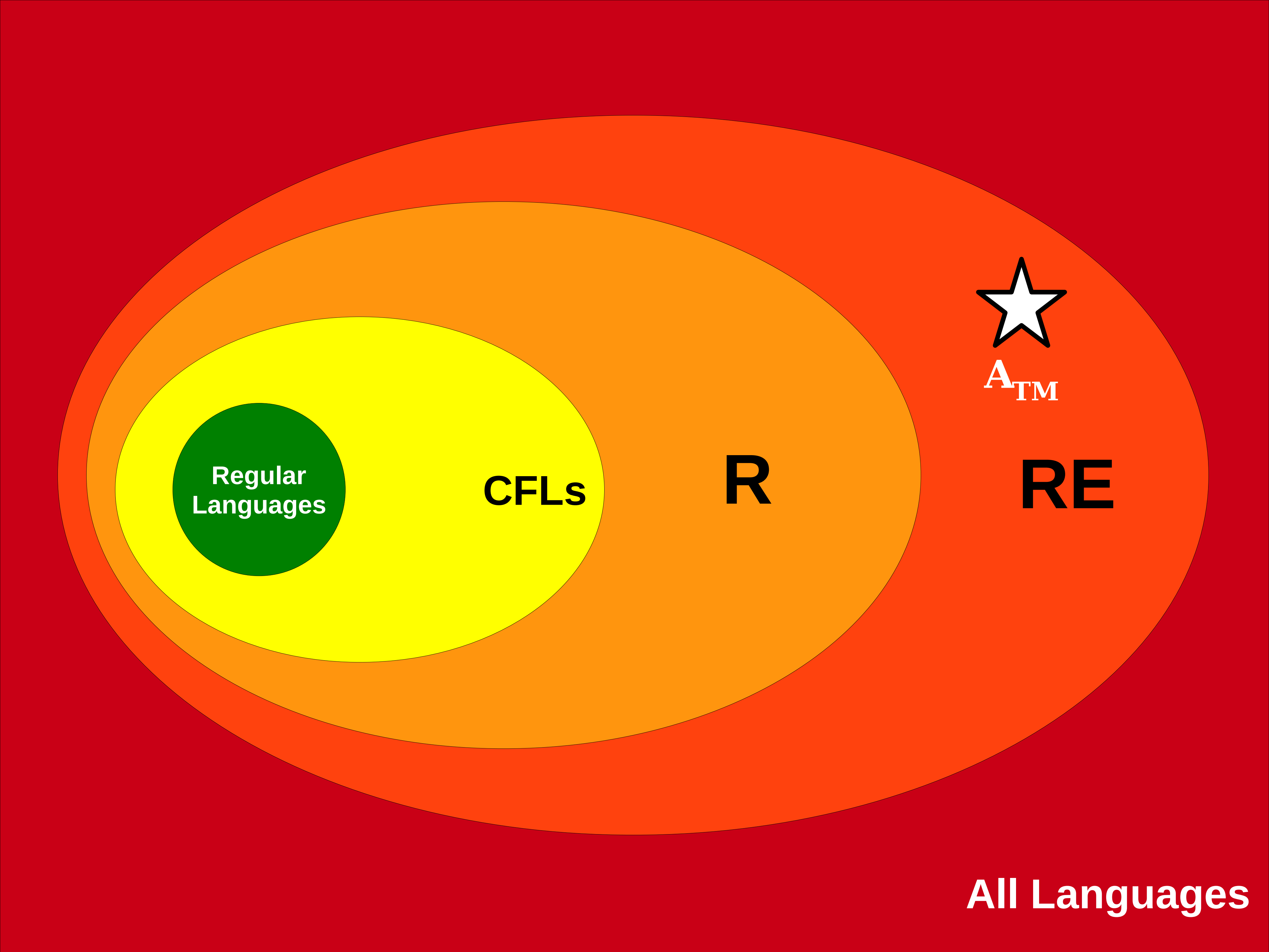
Given this, we could then construct this program P:

```
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, $A_{TM} \notin \mathbf{R}$. ■



Regular Languages
CFLs
R
RE
A_{TM}
All Languages

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.
- $\mathbf{R} \neq \mathbf{RE}$, because $A_{TM} \notin \mathbf{R}$ but $A_{TM} \in \mathbf{RE}$.

What do these three statements really mean? As in, why should you care?

Self-Defeating Objects

The fact that a decider for A_{TM} is a self-defeating object is analogous to this classic philosophical question:

If you know what you are fated to do, can you avoid your fate?

If we have a decider for A_{TM}, we could use it to build a TM that determines what it's supposed to do next, then chooses to do the opposite!

$A_{TM} \notin \mathbf{R}$

The proof we've done says that

There is no algorithm that can determine whether a program will accept an input.

Our proof just assumed there was some decider for A_{TM} and didn't assume anything about how that decider worked. No matter how you try to implement a decider for A_{TM}, you can never succeed!

$A_{TM} \notin \mathbf{R}$

What exactly does it mean for A_{TM} to be undecidable?

Intuition: The only general way to find out what a program will do is to run it.

As you'll see, this means that it's provably impossible for computers to be able to answer questions about what a program will do.

$A_{TM} \notin \mathbf{R}$

At a more fundamental level, the existence of undecidable problems tells us the following:

There is a difference between what is true and what we can discover is true.

Given a TM *M* and a string *w*, one of these two statements is true:

M* accepts *w

M* does not accept *w

But since A_{TM} is undecidable, there is no algorithm that can always determine which of these statements is true!

R ≠ RE

Because **R ≠ RE**, there is a difference between decidability and recognizability:

In some sense, it is fundamentally harder to solve a problem than it is to check an answer.

There are problems where when you have the answer, you can confirm it (build a recognizer), but where if you don't have the answer, you can't come up with it in a mechanical way (build a decider).

Next Time

More Undecidable Problems

Problems truly beyond the limits of algorithmic problem-solving!

Consequences of Undecidability

Why does any of this matter outside of a computer science course?