So far we’ve seen two computational models:

– ordinary finite automata (DFA) and their generalizations (NFA, $\epsilon$-NFA), which recognize the regular languages;

– pushdown automata, which recognize the context-free languages.

Now it’s time to see a third model, which is much more powerful than the other two: the Turing machine (TM).
Oddly enough, the Turing machine preceded the other two models.

It was invented c. 1936, by Turing.

The finite automaton, by contrast, was invented c. 1943, by McCulloch and Pitts.

And the pushdown automaton was invented c. 1961, by Oettinger.
Turing’s goal was not to model an electronic computer (since they didn’t even exist then).

It was to model a *human computer*.

In 1936, the term “computer” meant a *human* who calculated something (often with a desk calculator).

He realized a person has only a finite memory (analogous to the finite control of a TM) but has access to unbounded memory in the form of pieces of paper and a pencil.

Every calculation a human makes on a piece of paper consists of writing down symbols, erasing, and copying.

This is what a Turing machine can do.
Meaning of "computer"
There are many different flavors of Turing machine in the literature.

As it turns out, the computational power of all of them are the same (in terms of the class of languages recognized).

So it doesn’t really matter which flavor we use, but I will present the one in Prof. Watrous’s notes.

You should be aware that if you read other textbooks, you will almost certainly see minor variations on this model.
The Turing machine

For us, a basic Turing machine is

– a *deterministic* finite control

– a single tape that is unbounded in both directions, divided into cells

– a tape head that can *both* read from the tape and write to the tape

– and the tape head can move both to the left and to the right.
A PDA *can only move right* through its input. A TM can move *both* left and right.

A PDA is nondeterministic by default. A TM is *deterministic* by default; we’ll see nondeterministic TM’s later.

A PDA has its extra storage in the form of a stack. A TM has its extra storage on the same tape as the input (we’ll see variations later).

A PDA *cannot* change what is on the tape. A TM can arbitrarily change any cell of its tape.
How a Turing machine computes

When a computation begins, we assume the (finite) input is written on the tape, but all other cells contain a special blank symbol \( \_ \).

The single tape is used both to hold the input and to do any computations.

There is one accept state, which we usually call \( q_{\text{acc}} \), and one reject state, which we usually call \( q_{\text{rej}} \).
What a Turing machine can do

At every step, based on its current state, and the contents of the cell currently being scanned, the Turing machine

- rewrites the contents of the current cell, with either the same symbol or a different one;
- changes state to a possibly different state (unless it is in $q_{\text{acc}}$ or $q_{\text{rej}}$);
- moves its tape head either left or right.

In particular this means that we require that there always be a next move in every configuration of the TM, except if it is in $q_{\text{acc}}$ or $q_{\text{rej}}$.

(Some books and papers allow the possibility of having no next move in some configurations. We don’t.)
Movement of a Turing machine always refers to movement of the \textit{tape head}, not the tape itself.

We think of the tape as \textit{stationary}, and the read/write head moving left and right on it.
The computation continues until the TM enters either $q_{\text{acc}}$ or $q_{\text{rej}}$. At this point computation stops, and it is said to “halt”. If the TM is in $q_{\text{acc}}$, it accepts the input, and if in $q_{\text{rej}}$ it rejects the input.
On an input, there are three possible behaviors of a TM:

- It can halt by entering $q_{\text{acc}}$;
- It can halt by entering $q_{\text{rej}}$;
- It can not halt.

“Not halting” is also called “failing to halt” or “running forever” or “looping forever”.

When we use the term “loop” there is actually no implication that exactly the same sequence of moves is used over and over.

Furthermore, a TM can fail to halt in a “bounded way” (where it never goes further than a bounded distance from the first cell), or an “unbounded way” (where it goes arbitrarily far away from the first cell).
#1: When the TM enters $q_{\text{acc}}$, the input that is accepted is the part of the input that has been read so far.

This is not the case. At the beginning of the computation the TM is scanning the blank immediately to the left of the input $x$, which is defined to be the contents of the tape up until (but not including) the next blank to the right. If it enters $q_{\text{acc}}$, then $x$ is the string accepted. In particular, a TM does not have to read the entire input to accept it. For example, it can accept all strings starting with 0 by reading the first symbol and then entering $q_{\text{acc}}$ if it is a 0.

So this is different from the behavior of DFA’s and PDA’s.
Misconceptions about Turing machines

#2: The Turing machine knows the number of the cell it is scanning.

Cells are not numbered at all, actually. We may occasionally think of them as being numbered in order to talk about them, but the TM has no access to the numbers we make up. One can, of course, have the program keep track of the numbers of cells somehow, in ways we will talk about later.
Example #1 of a TM

Let’s build a TM for one of our favorite languages, $L = \{a^n b^n : n \geq 0\}$.

The idea is one of the basic techniques of “programming” a TM: checking off symbols. Namely, as we read each $a$, we convert it to a blank $\_\_$, and then try to match it to a $b$ in the second half of the string, which we also convert to a blank. If, eventually, there are no symbols left, we accept the input.

We will use the following notation for transitions:

$a, b \leftarrow$ means if we read an $a$, change it to $b$ and move left;

$a, b \rightarrow$ means if we read an $a$ change it to $b$ and move right.
We start in state $q_0$, and immediately move to the right on the tape. If we see an $a$, we change it to $\_\_$ and move right to the end of the string, find a $b$ at the end, change it to $\_\_$, and move to the left, back to the first $a$ again. In this way, we have changed $a^n b^n$ to $a^{n-1} b^{n-1}$, and the process can begin again.
Ultimately we have matched every $a$ against a $b$, and the result is the empty string.

At that point we can go to $q_{\text{acc}}$.

If at any point we read a symbol that shouldn’t be there (for example, a $b$ that starts the string) we go to $q_{\text{rej}}$. 
Example #2 of a TM

Next, let’s build a TM for the language \( \{a^{2^n} : n \geq 1\} \), which is not a CFL.

What’s the strategy? On each pass from left to right, the TM changes every other a that it sees to X, thus effectively dividing the number of a’s on the tape by two.

If the number of a’s is odd, but \( > 1 \), we reject.

If the number of a’s is one, then we accept.
Example #2 of a TM
Formal definition of a TM

Now we’re ready for the formal definition of a TM:

It is a 7-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$, where

- $Q$ is a finite set of states;
- $\Sigma$ is the finite input alphabet;
- $\Gamma$ is the finite tape alphabet (and $\Sigma \subseteq \Gamma$);
- $q_0$ is the initial state;
- $q_{\text{acc}}$ is the accepting state;
- $q_{\text{rej}}$ is the rejecting state.
- $\delta$ is the transition function.

The domain of $\delta$ is $(Q - \{q_{\text{acc}}, q_{\text{rej}}\}) \times \Gamma$ and the range of $\delta$ is $Q \times \Gamma \times \{\leftarrow, \rightarrow\}$.

Now we also need the analogue of instantaneous description for PDA. For TM it is called a “configuration”.
Configuration of a TM

The definition of configuration is a string of the form \( u(q, a)v \) where

\[ u, v \in \Gamma^* \]
\[ a \in \Gamma \]
\[ q \in Q \]

Here the contents of the tape is \( uav \) and the TM is in state \( q \) and is currently scanning the particular \( a \) in \( uav \).

The convention is that \( u \) doesn’t begin with a blank and \( v \) doesn’t end with a blank. If they do, we can truncate them to remove the appropriate blanks.

(Prof. Watrous’s notes does this in a more formal and careful way, but I think it’s not really necessary and just makes things more complicated than needed.)
Formal definition of moves of a TM

We need to say how we go from one configuration of a TM to another.

1. (Right move) Suppose $p \in Q$, $p \not\in \{q_{\text{acc}}, q_{\text{rej}}\}$ and $
\delta(p, a) = (q, b, \rightarrow)$. Then

$$u(p, a)cv \vdash ub(q, c)v$$

$$u(p, a) \vdash ub(q, \omega)$$

for each $c \in \Gamma$, $u, v \in \Gamma^*$.

2. (Left move) Suppose $p \in Q$, $p \not\in \{q_{\text{acc}}, q_{\text{rej}}\}$ and
\n$
\delta(p, a) = (q, b, \leftarrow)$. Then

$$uc(p, a)v \vdash u(q, c)bv$$

$$(p, a)v \vdash (q, \omega)bv$$

for each $c \in \Gamma$, $u, v \in \Gamma^*$.
Now we define the symbol $\vdash^*$ as we did for PDA’s: namely $c_1 \vdash^* c_2$ if we can go from configuration $c_1$ to configuration $c_2$ by a sequence of 0 or more moves of $\vdash$.

Now **acceptance** of $x$ by a TM is defined by

$$(q_0, \omega)x \vdash^* u(q_{\text{acc}}, a)v$$

for some strings $u, v \in \Gamma^*$ and $a \in \Gamma$.

**Rejection** of $x$ is defined by

$$(q_0, \omega)x \vdash^* u(q_{\text{rej}}, a)v$$

for some strings $u, v \in \Gamma^*$ and $a \in \Gamma$. 
Formal definition of acceptance

If neither the accept state nor the reject state is reached, then $M$ runs forever on input $x$.

Notice: if $M$ accepts an input $x$, then $M$ is guaranteed to halt on $x$.

If $M$ doesn’t accept $x$, then $M$ could either halt and reject, or run forever.

So there is an asymmetry between acceptance and rejection.
The language $L(M)$ recognized by a TM $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$ is now defined to be

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

$$= \{ x \in \Sigma^* : (q_0, \varepsilon)x \vdash u(q_{\text{acc}}, a)v \text{ for some } u, v \in \Gamma^* \text{ and } a \in \Gamma \}$$

This is a weak notion of acceptance, because it allows $M$ to run forever on inputs it does not accept. This is not desirable behavior (but we’ll see that it is unavoidable in some cases!).

A stronger notion of acceptance is **decidability**.

We say the TM $M$ **decides** a language $L$ if (a) $L = L(M)$ and (b) $M$ halts on every input.
Two flavors of Turing machine languages

If there exists $M$ such that $L = L(M)$, then we say that $L$ is Turing-recognizable or recursively enumerable or r.e. or computably enumerable.

The class of all Turing-recognizable languages is written RE.

If there exists $M$ such that $L = L(M)$ and $M$ decides $L$, then we say that $L$ is Turing-decidable or recursive or computable.

The class of all Turing-decidable languages is written REC.
The Church-Turing thesis says, roughly speaking, that everything that can be computed, can be computed by a Turing machine.

This is called a “thesis” and not a “theorem” because it is about what is actually computable by physical machines. It is not susceptible to mathematical proof.

Evidence for it includes the fact that all reasonable mathematical models of computation can be simulated by Turing machines, and all actually-realized physical computers obey it.

It is not contradicted by alternate models of computation, such as DNA computers or quantum computers.
Thinking about TM’s

Learning to program TM’s is kind of like learning to program in a very low-level language like assembly language, except that there are even fewer instructions.

There are no built-in instructions for adding numbers, or comparing strings. Everything you wish to do must be built up out of the primitive operations of reading and writing single symbols.

One of our goals in studying TM’s is to convince you that a TM can do anything a modern programming language can do.

Once you believe that, in almost all cases, when you think about what a TM can do, you can substitute “what you can do in Java”, or “what you can do in Python”, or any programming language that you like.
Thinking about TM’s

In particular, think about writing a Java program that can decide the membership problem for a language $L$, that is, given $x$, is $x \in L$?

You’d definitely want such a program to always eventually halt and tell you “yes” or “no”.

You would find it quite unsatisfactory if sometimes the program just sat there and looped forever. You would never know when to hit interrupt!

When we talk about Turing-decidable languages, we have the “good” behavior in mind: the TM *must* halt on all inputs and either accept or reject.

When we talk about Turing-recognizable languages, we have the “bad” behavior in mind: the TM is allowed to loop forever if $x \notin L$. Of course, it doesn’t *have* to loop forever; it is just allowed to.
Thinking about TM’s

As an example of the “bad” kind of program, consider writing a program to answer the question, given a positive integer $n$, are there two primes $p_1, p_2$ such that $n = p_1 - p_2$?

If we try to solve this by checking, for each prime $p$, whether $p + n$ is a prime, then this check will run forever if the answer for $n$ is “no”.

For example, suppose $n = 496562420542$. What is the first prime $p$ such that $p + n$ is prime?

Currently nobody knows if there is a “good” program for this problem (one that halts on all inputs $n$).