St. Francis Xavier University
Department of Computer Science

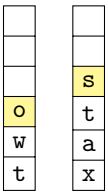
CSCI 356: Theory of Computing
Lecture 3: Beyond Context-Free Languages
Fall 2022

1 Turing Machines

When we introduced pushdown automata, we saw that augmenting our machine with a stack gave it a rudimentary form of memory, and it could therefore recognize a larger set of languages. However, in spite of the stack having an unbounded capacity (and therefore being able to store as many symbols as it wants), we're still limited by the fact that it's a stack, meaning it can only access the *top* symbol at any given time.

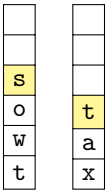
So, how do we overcome this limitation? Let's try adding *two stacks* to our machine! It may not sound like a huge or meaningful change—after all, what can we get with another stack that we didn't already have with the first stack?—but, just like with heads, it turns out that two stacks are indeed better than one.

Suppose we now have two stacks available for us to use. For the purposes of this example, the stacks have been initialized with some symbols already in them.

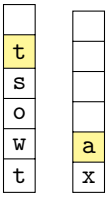


Even with two stacks, we can still only access the top symbol of each stack: in the first stack, we can access the symbol `o`, and in the second stack, we can access the symbol `s`.

However, what happens if we use the two stacks in tandem? Suppose we pop one symbol from a stack, say `s`, and push it to the other stack.



Now, we have access to the symbol `t` that was beneath `s` in the second stack, and we haven't lost the symbol `s`, since it's safely stored in the first stack! Similarly, if we pop that symbol `t` from the second stack and push it to the first stack, we can get access to the symbol `a` in the second stack:



It's looking like having two stacks is far more meaningful than we might've initially thought. We can push and pop symbols between the two stacks in order to get access to *any* symbol we've stored in the machine's memory, instead of only the most recent symbol at the top.

Indeed, if we took our two stacks and we aligned them horizontally in such a way that the "bottoms" of each stack were on the left and right edges...

t	w	o	s	t	a	x

...we would get a new form of storage: a *tape*.

With a tape, we can move left and right through each cell and access each symbol stored on the tape whenever we want. (This is exactly what we were doing when we pushed and popped symbols between our two stacks.) Additionally, just like our stacks had unbounded capacity, our tape has infinite length. Therefore, we can write as many symbols to the tape as we want, and we can write them to either the left side or the right side of the tape.[1]

···			i	s	a	t	a	p	e			···

Since we can now access any symbol of the tape that we want, tape cells that do not contain a symbol become an important consideration. With stacks, we didn't need to worry about blank spaces; we only pushed to or popped from the top of the stack, so we would never encounter a situation where two symbols were separated by a blank space. With tapes, we must account for blank spaces, so we'll denote a blank space on a tape by the symbol $\sqcup$.

···	$\sqcup$	$\sqcup$	i	s	a	t	a	p	e	$\sqcup$	$\sqcup$	···

Having defined this notion of a tape, we can now augment our machine with a tape instead of a stack. In doing so, we will obtain one of the most powerful abstract models of computation possible; even more powerful than any real-world computer!

1.1 Definition

The focus of this lecture, the *Turing machine*, is a model of computation that consists of two components: a finite-state control (much like the states of a finite automaton or a pushdown automaton) and an infinite-length tape. The finite-state control keeps track of where we are in the computation, while the tape serves as the machine's memory throughout the computation.

At the beginning of a computation, the tape holds the input word given to the Turing machine, and all other cells of the tape are blank. Since the input word is initially stored on the tape, we can assume that the input alphabet Σ is a subset of the tape alphabet Γ. The input head of the Turing machine starts on the leftmost symbol of the input word. It can move along the tape, and it can both read from and write to cells of the tape. This means that we can use the tape both to store (and potentially modify) not only the input word, but also any auxiliary information we need to use during the computation.

To model the Turing machine's input head movement along the tape, we must account for the direction of movement in the transition function. To figure out the next step of the computation, our transition function will take as input our current state and the tape symbol the input head reads in the current cell, and it will produce as output the state we are transitioning to, the tape symbol the input head will write to the current cell, and the direction the input head will move: one cell leftward (L) or one cell rightward (R).

One other key difference that sets Turing machines apart from finite automata and pushdown automata is in how it accepts or rejects input words. Unlike finite automata or pushdown automata, which can "run out" of input by reaching the ends of their input words, a Turing machine could theoretically read the symbols on its tape as many times as it wants. Therefore, we must fix two special "accept" and "reject" states such that, whenever the computation of the Turing machine enters one of those states, it immediately halts the computation and accepts or rejects the input word accordingly. Note that, if the Turing machine doesn't visit either of these states during its computation, then its computation will continue indefinitely.

[1] If we take the "two stacks" perspective of thinking about a tape, then writing a symbol to the right side of the tape would require us to move all existing symbols to the first stack and push the new symbol to the second stack. Writing to the left side of the tape is similar.

Apart from these changes, then, the formal definition of a Turing machine is quite similar to our definitions of finite automata and pushdown automata.

Definition 1 (Turing machine)**.** A Turing machine is a tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, where

- Q is a finite set of *states*;
- Σ is the *input alphabet* (where $\sqcup \notin \Sigma$);
- Γ is the *tape alphabet* (where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$);
- $\delta : (Q \setminus \{q_{\text{accept}}, q_{\text{reject}}\}) \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the *transition function*;
- $q_0 \in Q$ is the *initial* or *start state*;
- $q_{\text{accept}} \in Q$ is the *final* or *accepting state*; and
- $q_{\text{reject}} \in Q$ is the *rejecting state*.

Example 2. Consider the context-free language $L_{\text{a=b}} = \{\texttt{a}^n\texttt{b}^n \mid n \geq 1\}$. Even though we know the language is context-free (and is therefore recognized by a pushdown automaton), let's construct a Turing machine recognizing this language.

The idea behind our Turing machine is as follows. Given an input word of the form $\texttt{a}^n\texttt{b}^n$ on the tape, the input head will move back and forth, replacing all `a`s with `X`s and all `b`s with `Y`s. In a sense, the input head is "marking" `a`s and `b`s as it sees them. Each time the input head replaces an `a` with an `X`, it will move rightward in an attempt to find a matching `b` that it can replace with a `Y`. The input head will then move leftward and repeat the process until no more `a`s remain.

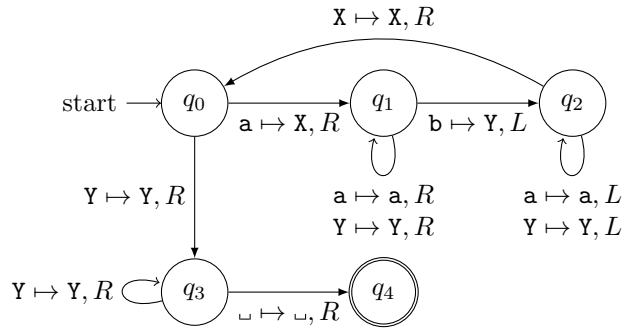
We formally define the Turing machine as follows:

- $Q = \{q_0, q_1, q_2, q_3, q_4, q_R\}$;
- $\Sigma = \{\texttt{a}, \texttt{b}\}$;
- $\Gamma = \{\texttt{a}, \texttt{b}, \texttt{X}, \texttt{Y}, \sqcup\}$;
- $q_0 = q_0$;
- $q_{\text{accept}} = q_4$;
- $q_{\text{reject}} = q_R$; and
- δ is specified by the following table:

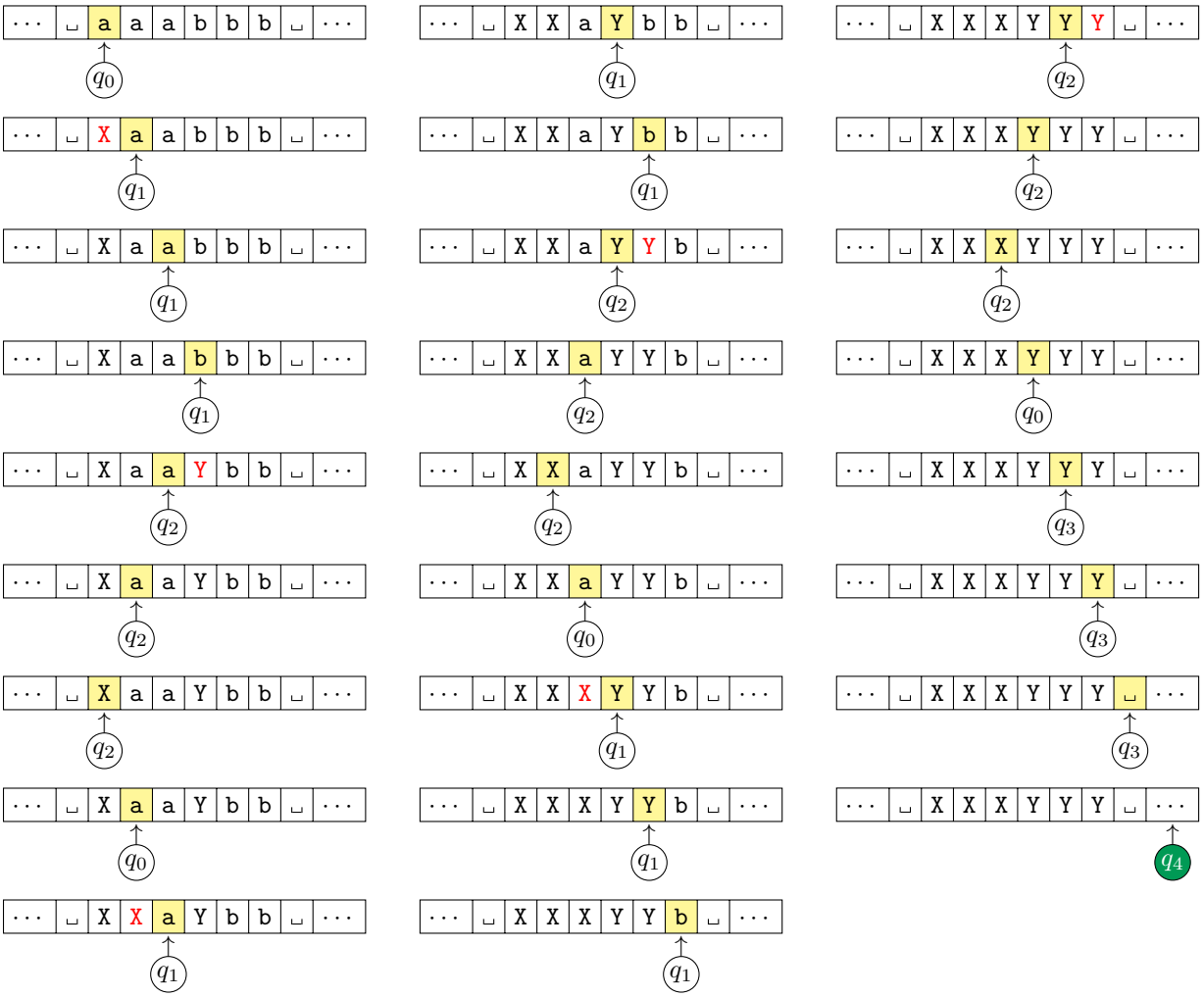
Γ	`a`	`b`	`X`	`Y`	$\sqcup$
q_0	$(q_1, \texttt{X}, R)$	—	—	$(q_3, \texttt{Y}, R)$	—
q_1	$(q_1, \texttt{a}, R)$	$(q_2, \texttt{Y}, L)$	—	$(q_1, \texttt{Y}, R)$	—
q_2	$(q_2, \texttt{a}, L)$	—	$(q_0, \texttt{X}, R)$	$(q_2, \texttt{Y}, L)$	—
q_3	—	—	—	$(q_3, \texttt{Y}, R)$	$(q_4, \sqcup, R)$
q_4	×	×	×	×	×
q_R	×	×	×	×	×

Note that we can't have any transitions from either state q_4 or q_R, since those are the accepting and rejecting states, respectively. Instead of writing transitions to state q_R, we just assume that any undefined transition in the table (—) automatically leads to state q_R.

We can draw this Turing machine graphically, just like a finite automaton or a pushdown automaton. To reduce the number of transitions we need to draw, we will omit the state q_R and all transitions leading to it, and we will just assume (again) that all transitions not included automatically lead to state q_R. The Turing machine looks like the following:



Now, let's suppose we give the input word `aaabbb` to this Turing machine. The input head will start its computation in state q_0 on the leftmost symbol of the input word. Moving from the top to the bottom of each column, we will highlight the state of the machine and the input head's current tape cell at each step.



Since the computation halted in the accepting state q_4, we know that the machine accepts the word `aaabbb`.

1.2 Configurations, Accepting, and Rejecting

You may have noticed in the previous example that the computation of the Turing machine on the given input word took up a *lot* of space on the page. This is because we had to draw the current state, the entire tape, and the position of the input head on the tape, all at each step. Fortunately, however, there is a more concise way to present this information.

All we need to specify a particular stage of the computation is the current state, the current tape contents, and the current input head position, and we can represent this using a single sequence of symbols. This sequence is called a *configuration* of the Turing machine. If the Turing machine is currently in a state q, its tape contains the symbols $X_1X_2\cdots X_{i-1}X_i\cdots X_{n-1}X_n$, and its input head is at cell i of the tape, then the configuration of the Turing machine at this moment is written

$$X_1X_2\cdots X_{i-1}qX_i\cdots X_{n-1}X_n.$$

If we can get from a configuration C_i to a configuration C_{i+1} in a single computation step, then we say that C_i *yields* C_{i+1} and we write $C_i \vdash C_{i+1}$. Formally, given $a, b, c \in \Gamma$, $u, v \in \Gamma^*$, and $q_i, q_j \in Q$, we say that uaq_ibv yields $uacq_jv$ if $\delta(q_i, b) = (q_j, c, R)$. We can define the notion of "yields" for leftward moves similarly.

Example 3. Recalling the Turing machine's computation from the previous example, the first five configurations of the machine are

$$q_0\texttt{aaabbb} \vdash \texttt{X}q_1\texttt{aabbb} \vdash \texttt{Xa}q_1\texttt{abbb} \vdash \texttt{Xaa}q_1\texttt{bbb} \vdash \texttt{Xa}q_2\texttt{aYbb} \vdash \cdots$$

By our earlier definition, we know that a Turing machine has dedicated accept and reject states, and we know that the machine's computation either accepts or rejects once it enters the appropriate state. With the notion of configurations, we can specify exactly what it means for a Turing machine to accept or reject its input word.

If we have a Turing machine $\mathcal{M}$ and an input word w, then we say that the *start configuration* of $\mathcal{M}$ on w is q_0w. In this configuration, $\mathcal{M}$ is in its initial state q_0, and the input head of $\mathcal{M}$ is on the leftmost symbol of the input word.

Likewise, an *accepting configuration* is one where the current state of $\mathcal{M}$ is q_{accept}, and a *rejecting configuration* is one where the current state of $\mathcal{M}$ is q_{reject}. Note that, in either configuration, we only care about the state and not the tape contents. This is because once we enter the accepting or rejecting state, the computation immediately halts, and so the tape contents don't have any effect on the accepting or rejecting configuration.

We can now formally define what it means for the Turing machine $\mathcal{M}$ to accept its input word w.

Definition 4 (Accepting computation of a Turing machine)**.** Let $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ be a Turing machine, and let w be an input word. The Turing machine $\mathcal{M}$ accepts the input word w if there exists a sequence of configurations $C_1, C_2, \ldots, C_n$ satisfying the following conditions:

1. C_1 is the start configuration of $\mathcal{M}$ on w;
2. $C_i \vdash C_{i+1}$ for all $1 \leq i \leq (n-1)$; and
3. C_n is an accepting configuration.

We can write a similar definition for a rejecting computation of a Turing machine by considering rejecting configurations in the third condition.

Lastly, the set of all input words accepted by a Turing machine $\mathcal{M}$ is referred to as the language of the machine $\mathcal{M}$, written $L(\mathcal{M})$.

1.3 Deciding and Semideciding

We noted earlier on that, unlike with finite automata and pushdown automata, a Turing machine cannot "run out" of input symbols. Indeed, it can read the symbols on its tape as many times as it wants. The machine's computation immediately halts once the machine enters either the accepting or the rejecting state, but there's no guarantee that it will ever enter either of these states during its computation. We must account for the possibility that the machine simply never ends its computation; that is, the machine enters a *loop*.

Every Turing machine recognizes some language, but the class to which that language belongs is determined by the machine's behaviour on each input word. Let's first focus on the scenario where the machine always either accepts or rejects every input word its given.

If the Turing machine accepts all words that belong to its language and rejects all words that don't belong to its language, then we say that the machine *decides* its language.

Definition 5 (Decidable language)**.** Given a Turing machine $\mathcal{M}$, we say that the language $L(\mathcal{M})$ is decidable if,

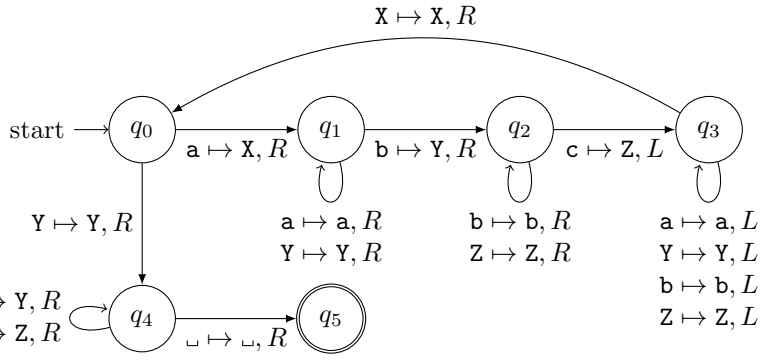
- whenever $w \in L(\mathcal{M})$, then $\mathcal{M}$ accepts w; and
- whenever $w \notin L(\mathcal{M})$, then $\mathcal{M}$ rejects w.

We denote the class of decidable languages by D. In the literature, the class of decidable languages is sometimes called the class of *recursive languages*, where the term "recursive" comes from the origins of computer science and its connections to recursive functions and sets.

Example 6. Consider the language $L_{\mathtt{a=b=c}} = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 1\}$. We know that this language is non-context-free, so we can't construct a pushdown automaton recognizing the language. Let's instead construct a Turing machine recognizing the language.

Our Turing machine will function in much the same way as the machine we constructed to recognize words of the form $\mathtt{a}^n\mathtt{b}^n$; moving from left to right, we will match symbols up to one another by replacing the symbols on the tape.

Suppose our alphabets are $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ and $\Gamma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{X}, \mathtt{Y}, \mathtt{Z}, \sqcup\}$. The Turing machine, then, will look like the following:



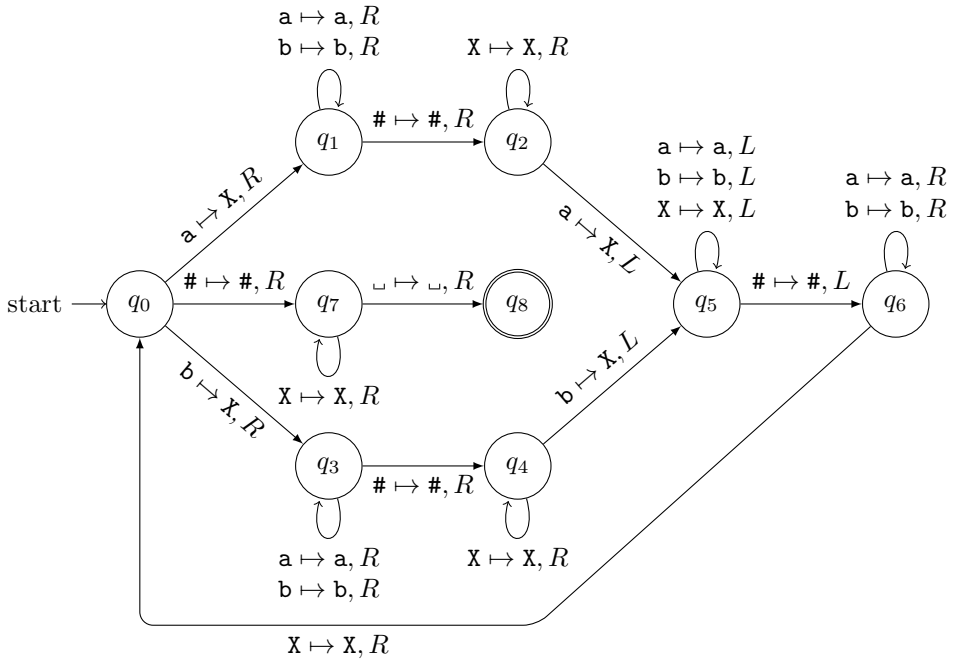
This Turing machine decides the language $L_{\mathtt{a=b=c}}$ because (i) if some input word w belongs to this language, then the machine will accept it; and (ii) if some input word w does not belong to this language, then the machine will (implicitly) go to the rejecting state q_{reject}.

Example 7. Consider the language $L_{\text{double\#}} = \{w\#w \mid w \in \Sigma^*\}$. This language is very similar to the language L_{double}, which we previously showed was non-context-free. Let's therefore construct a Turing machine for the language $L_{\text{double\#}}$.

The idea behind this Turing machine is to move back and forth between each copy of w, marking off each symbol in the first copy and using the states of the machine to "remember" this symbol as we move to

the second copy. If the machine is able to match every symbol in both copies of the word, then it accepts. Otherwise, it (implicitly) rejects.

Suppose our alphabets are $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ and $\Gamma = \{\mathtt{a}, \mathtt{b}, \mathtt{X}, \sqcup\}$. The Turing machine will look like the following:



This Turing machine decides the language $L_{\text{double\#}}$ because it accepts all words of the form $w\#w$ and (implicitly) rejects all other words.

Of course, there's no requirement that a Turing machine always either accepts or rejects every input word it's given. As we mentioned before, it's possible that the machine could enter a loop. In this case, then, we can only get a definite answer from the machine if the input word belongs to the machine's language. If a Turing machine accepts all and only those words that belong to its language, then we say that the machine *semidecides* its language.[2]

Definition 8 (Semidecidable language)**.** Given a Turing machine $\mathcal{M}$, we say that the language $L(\mathcal{M})$ is semidecidable if,

- whenever $w \in L(\mathcal{M})$, then $\mathcal{M}$ accepts w; and
- whenever $w \notin L(\mathcal{M})$, then either $\mathcal{M}$ rejects w or $\mathcal{M}$ enters a loop.

We denote the class of semidecidable languages by SD. In the literature, the class of semidecidable languages is sometimes called the class of *recognizable languages* or *recursively enumerable languages*. The term "recursively enumerable" again comes from a connection to mathematics and the notion of recursively enumerable sets, but in a machine-oriented context, a recursively enumerable language is one for which a Turing machine can list (or *enumerate*) every word in the language.

We can come up with any number of artificial examples of semidecidable languages, but it's often the case that making a small change to the Turing machine results in those languages also becoming decidable. In the next lecture, we will focus on some more natural examples of semidecidable languages. By "natural", we mean that the language models some inherent property or quality of the machine that recognizes it.

[2]We say the language is "semidecidable" because the Turing machine is only able to decide in the positive case; that is, when the input word belongs to the machine's language.

1.4 Variants of Turing Machines

The definition of a Turing machine that we gave earlier in this lecture is by no means a canonical definition. When we chose to give our machine a deterministic transition function, or a single tape, or a two-way infinite tape, we made those choices simply to fix *some* definition of a Turing machine. Since we are just introducing Turing machines for the first time in this lecture, we decided to go with a relatively easy-to-understand definition: the single, two-way infinite tape allowed us to use our "two stacks" perspective to motivate the definition, and deterministic transition functions are more straightforward to reason about.

Just like we defined different types of finite automata, nothing is stopping us from modifying our definition of a Turing machine. One of the most remarkable results in this area, though, is that we can make pretty much *any* modification to our definition of a Turing machine, and it won't affect the recognition power of the model. The Turing machine is just so powerful that even our easy-to-understand definition is sufficient to recognize the large classes of decidable and semidecidable languages.

Here, we will make a number of modifications to our Turing machine definition, and we will then prove that each modified definition is equivalent to our original definition. This will give us a wide array of variant Turing machine models that we can choose from when we're trying to recognize certain languages.

Nondeterministic Turing Machines

The first natural modification we can make to our definition is to take the transition function δ to be nondeterministic. Just like with our other models of computation, a nondeterministic transition function for a Turing machine will map a pair of state and tape symbol to the power set of tuples of state, tape symbol, and input head movement.

Definition 9 (Nondeterministic Turing machine)**.** A nondeterministic Turing machine is a tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, where everything is defined as in Definition 1 except for the transition function, which is

$$\delta : (Q \setminus \{q_{\text{accept}}, q_{\text{reject}}\}) \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\}).$$

We saw that deterministic and nondeterministic finite automata are equivalent in terms of recognition power, while nondeterministic pushdown automata are able to recognize more languages than deterministic pushdown automata. With Turing machines, we return to equivalence: adding nondeterminism to a Turing machine does not give it more recognition power.

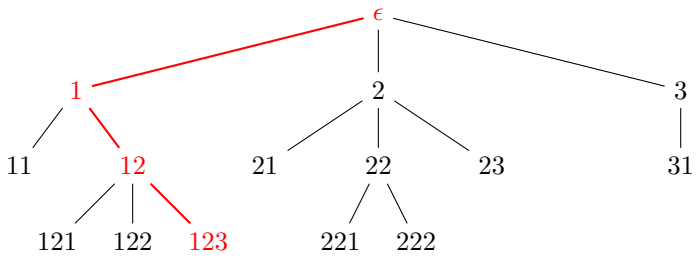
Theorem 10. *Given a nondeterministic Turing machine $\mathcal{M}$, we can convert it to a deterministic Turing machine $\mathcal{M}'$.*

Proof. Let $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ be a nondeterministic Turing machine. We will construct a deterministic Turing machine $\mathcal{M}'$ that uses three tapes to simulate the nondeterministic computation of $\mathcal{M}$.

- The first tape will be called the *input tape*, and it will contain the input word given to $\mathcal{M}$. The contents of the input tape will not be changed during the computation.
- The second tape will be called the *simulation tape*, and it will simulate the contents of $\mathcal{M}$'s tape as $\mathcal{M}$ performs its nondeterministic computation.
- The third tape will be called the *address tape*, and it will keep track of where we are in the nondeterministic computation tree of $\mathcal{M}$.

Before we proceed, let's consider how we can represent the nondeterministic computation tree of $\mathcal{M}$ on a linear form of storage like a tape. If we take b to denote the maximum number of branches in the tree (that is, the maximum number of nondeterministic transitions $\mathcal{M}$ can follow at any given point in its computation), then we can assign a unique address to each vertex of the tree over the alphabet $\Sigma_b = \{1, 2, \ldots, b\}$. The address is determined by tracing the branches we must follow in order to get from the root to that vertex; for example, the vertex with address 123 can be reached by starting at the root of the tree and taking the first

branch, followed by the second branch, followed by the third branch. As a consequence of this convention, the root of the tree receives the address ϵ.



The address tape, then, contains symbols from the alphabet Σ_b. Each symbol on the address tape will tell $\mathcal{M}'$ which branch of the nondeterministic computation tree it must follow in its next computation step. Note that the contents of the address tape do not necessarily need to correspond to a vertex of the tree; if the address tape contains an invalid address, then $\mathcal{M}'$ simply aborts its attempted simulation of that branch.

Having defined the three tapes, we can describe how $\mathcal{M}'$ simulates the computation of $\mathcal{M}$:

1. Initialize the tapes in the following way:
   (a) Write to the input tape the input word w given to $\mathcal{M}$.
   (b) Leave both the simulation and address tapes empty.
2. Copy the contents of the input tape to the simulation tape.
3. Use the simulation tape to perform the following steps:
   (a) For each computation step of $\mathcal{M}$, read the next symbol of the address tape to determine which branch of the nondeterministic computation tree to follow.
   (b) If there are no more symbols remaining on the address tape, or if the address tape contains an invalid address, or if $\mathcal{M}$ enters a rejecting configuration, then abort the attempted simulation of this branch and go to step 4.
   (c) If $\mathcal{M}$ enters an accepting configuration, then accept w.
4. Write to the address tape the sequence of symbols over Σ_b that comes next in lexicographic order and go to step 2. □

Since every deterministic Turing machine is a nondeterministic Turing machine that doesn't use nondeterminism during its computation, we immediately obtain the other direction of the relationship between these models. Therefore, we can conclude that deterministic and nondeterministic Turing machines are equivalent.

Multitape Turing Machines

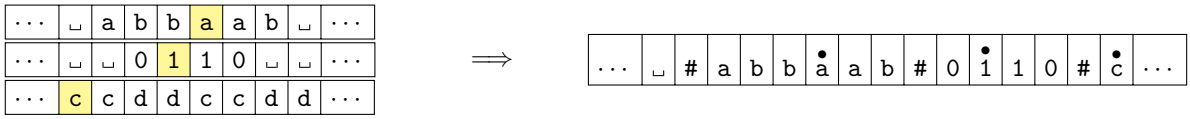
In the proof of Theorem 10, we saw that we could simulate the computation of a nondeterministic Turing machine using a deterministic Turing machine with multiple tapes. It's natural to wonder whether including additional tapes gave us some kind of advantage in this simulation process—after all, giving our computational model two stacks instead of one stack led us to the idea of a Turing machine—but as it turns out, we can perform exactly the same computations using only a single tape.

First, let's define our *multitape Turing machine* model. Again, we only need to modify the transition function: instead of mapping a pair of state and *one* tape symbol to a tuple of state, *one* tape symbol, and *one* input head movement, we will transition on k tape symbols and k input head movements, where k denotes the number of tapes used by the machine.

Definition 11 (k-tape Turing machine)**.** A k-tape Turing machine is a tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, where everything is defined as in Definition 1 except for the transition function, which is

$$\delta : (Q \setminus \{q_{\text{accept}}, q_{\text{reject}}\}) \times \Gamma^k \to Q \times \Gamma^k \times \{L, R\}^k.$$

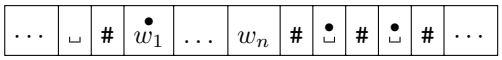
The idea allowing us to establish one direction of the equivalence between multitape and single-tape Turing machines is that we can simulate having many tapes T_i by storing all of the contents on a single tape T and separating "tape i" from the other "tapes" using a special symbol. Since our single tape has only one input head, we will also simulate the position of each input head on "tape i" using a special marker on the corresponding tape symbol to act as a virtual input head.



Theorem 12. *Given a k-tape Turing machine $\mathcal{M}$, we can convert it to a single-tape Turing machine $\mathcal{M}'$.*

Proof. Suppose the given Turing machine $\mathcal{M}$ has k tapes, and the tape alphabet is denoted by Γ. Our single-tape Turing machine $\mathcal{M}'$ will simulate $\mathcal{M}$'s computation on an input word $w = w_1 \dots w_n$ in the following way:

1. Take the tape alphabet of $\mathcal{M}'$ to be $\Gamma' = \Gamma \cup \dot{\Gamma} \cup \{\#\}$, where $\dot{\Gamma}$ consists of all alphabet symbols of Γ augmented with a dot and # is a special boundary marker.
2. Write the boundary marker # and the symbols of w to the tape of $\mathcal{M}'$, including the dotted symbol w_1. Then, write $k + 1$ copies of # each separated by a dotted blank space.



3. For each step of the computation, $\mathcal{M}'$ scans its entire tape from the first occurrence of # to the $(k+1)$st occurrence of # to read the symbols on all k tapes of $\mathcal{M}$. Then, $\mathcal{M}'$ makes a second pass along its tape to update any symbols that were changed by the transition function of $\mathcal{M}$. This update includes changing occurrences of dotted symbols in accordance with the changed positions of each virtual input head.

   If any of the virtual input heads of $\mathcal{M}'$ move onto an occurrence of #, then $\mathcal{M}$ must have moved the corresponding input head of that tape onto a blank space. In this case, $\mathcal{M}'$ writes a blank space to this cell of its tape and shifts the symbols of all subsequent cells rightward by one position. □

Naturally, a k-tape Turing machine can simulate the computation of a single-tape Turing machine by using only one of its k tapes. This again gives us the other direction of the relationship between the models, and again establishes the equivalence of the models.

One-Way Infinite Tape Turing Machines

In our definition of a Turing machine, we assumed that the tape used by the machine is *two-way infinite*; that is, there is an infinite number of cells to the left and to the right of the input head, and the input head can therefore move to an infinite number of positions of the tape in either direction.

We didn't have to define our storage in this way, though. We could have alternatively defined the tape to act more like the stack of a pushdown automaton: just like the stack has a fixed bottom boundary forcing us to push symbols only above that boundary, the tape could have a fixed left boundary forcing us to write symbols only to the right of that boundary. We call such a tape *one-way infinite*, since at any position along the tape, the input head has a finite number of cells to its left and an infinite number of cells to its right.

Because of the way the computation of the Turing machine is initialized, the initial position of the input head of a Turing machine with a one-way infinite tape will be on the first symbol of the input word, and