To make our decision, we need to strip away all of the real-world baggage associated with the computers we use every day and focus on the abstract. Thus, we won't care exactly how powerful a processor is, or how many cores it has—we will only require that there exists a processor that can execute instructions. Likewise, we won't care exactly how much memory or storage our computer has—in fact, we will assume that our computer has an *infinite* amount of memory. The only criteria we care about, therefore, are that the computer can process instructions and that we can measure what it does by some metric—say, by the number of instructions executed as a measure of time, or by the number of memory cells used as a measure of space.

There are many abstract *models of computation* that allow us to reason about problems, algorithms, and the limits of what we can compute. You may have heard of some of them before; for example, a finite automaton is a very simple model of computation. The model we will use in this course—the *Turing machine*—is quite a bit more powerful than a finite automaton, but nonetheless it remains a very simple model for us to work with and reason about. It encapsulates nicely what it means for us to "compute" something, it allows us to concretely count the number of computation steps we perform as a measure of time, and its tape gives us a way to count the amount of memory used by a computation as a measure of space.

2.1 Deterministic Turing Machines

A Turing machine consists of two components: a finite-state control 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. In this way, we can use the tape to store and modify not only the input word, but also any auxiliary information we need to use during the computation.

To model the movement of the Turing machine's input head 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 will transition to, the tape symbol the input head will write to the current cell, and the direction in which the input head will move: one cell leftward (L) or one cell rightward (R).

Note that, at least initially, we will take our Turing machine to be deterministic; that is, for each tuple of current state and tape symbol being read, we will have at most one transition to a given tuple of state, tape symbol being written, and input head movement.

Since a Turing machine can move its input head back and forth along the tape, it could theoretically read the symbols on its tape as many times as it wants. Therefore, we fix two special "accept" and "reject" states where, 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 it will continue to compute indefinitely.

Taken together, we get the formal definition of a deterministic Turing machine.

Definition 4 (Deterministic Turing machine)**.** A deterministic 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*.

Variants of Turing Machines

The Turing machine model is quite robust, and it turns out that we can make many changes to its definition without affecting its computational power.

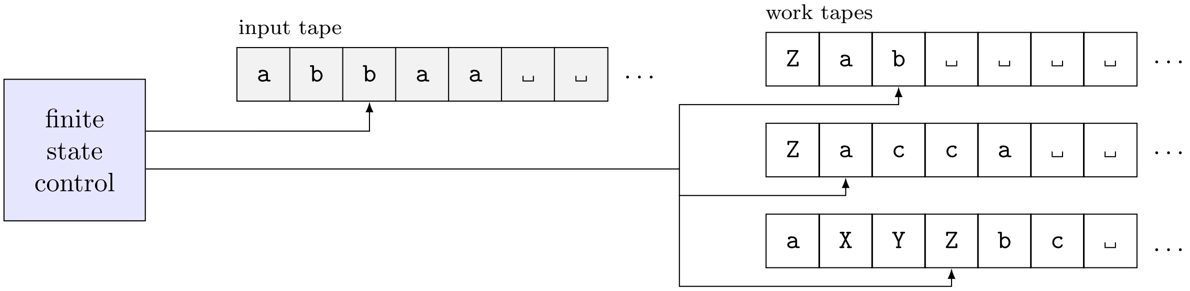
For example, consider the following: the deterministic Turing machine as written in our definition uses one tape for everything. We begin our computation with the input word written to the tape, we write all of our auxiliary information to the tape, and we need to differentiate between everything as we go forward with the execution of our algorithm. Must we restrict ourselves to using just this one tape? Thankfully, no! In the spirit of making our job easier, we can define a variant Turing machine model that uses *multiple tapes*.

Likewise, when we said that a Turing machine has an infinite-length tape, we neglected to mention whether that tape was *one-way infinite* or *two-way infinite*. In both cases, the tape contains an infinite number of cells, but a one-way infinite tape has a fixed left boundary and all of the cells extend infinitely to the right whereas a two-way infinite tape has cells extending infinitely in both directions. Again, striving for the easier option, we will assume that we're dealing with one-way infinite tapes.

Thus, the specific model we will use here is a one-way k-tape deterministic Turing machine. As you might have guessed by the name, this model consists of three components:

1. a finite-state control, as before;
2. one read-only input tape; and
3. $k - 1$ readable and writable work tapes.

Each tape has its own input head, and that input head interacts only with its associated tape. At each step of its computation, the Turing machine can read each of the k symbols being scanned by each of the input heads, it can write to each of the cells on any of the work tapes, and it can move each of the input heads left or right independently.



As before, we can formalize our definition in the following way:

Definition 5 (k-tape deterministic Turing machine)**.** A k-tape deterministic Turing machine is a tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, where everything is defined as in Definition 4 except for the transition function, which is

$$\delta : (Q \setminus \{q_{\text{accept}}, q_{\text{reject}}\}) \times \Gamma^k \to Q \times \Gamma^{k-1} \times \{L, R\}^k.$$

The only significant change here is that, 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 $k - 1$ tape symbols and k input head movements.

Although we move all k input heads, we need only write $k - 1$ tape symbols because the input tape is read-only. Note also that we don't need to incorporate into our definition the fact that each of the k tapes is one-way infinite, since this ultimately doesn't affect the way the Turing machine performs its computation.

Configurations and Accepting Configurations

Speaking of how we perform a computation, it would be good for us to devise some kind of notation to represent each individual computation step performed by a Turing machine. While we could write out every minute detail of the Turing machine at each discrete step from start to finish, this would quickly become tedious and it would take up a lot of paper. Fortunately, there is a much more concise way to summarize such information.

All we need to specify a particular stage of some computation is the current state, the current tape contents, and the current input head position, and we can represent all of this using sequences of symbols. These sequences taken together give us a *configuration* of the Turing machine.

At the beginning of the computation of some Turing machine $\mathcal{M}$, the input word w is stored on the input tape t_1 and the input head for this tape is positioned over the first symbol of w. Additionally, for $2 \leq i \leq k$, each work tape t_i is initially filled with blank symbols, and the ith input head is positioned over the leftmost blank cell of the tape. This is the *start configuration* of $\mathcal{M}$, and we write it in shorthand as

$$(q_0 w,\ \underbrace{q_0,\ \ldots,\ q_0}_{k-1 \text{ times}})$$

This notation is essentially a tuple where the ith element is the contents of tape t_i. If these contents are $u_i q v_i$, where $q \in Q$ and $u_i, v_i \in \Gamma^*$, then this indicates that we are in state q and the input head is positioned over the first symbol of the subword v_i. Thus, in the start configuration, we are in state q_0 with the input heads positioned exactly where we said they would be. (Since none of the work tapes have symbols written to them, we write only q_0 with no blank spaces.)

Example 6. Consider our figure of a 4-tape deterministic Turing machine from earlier. If we suppose that the finite-state control of this Turing machine is in state q, then we can write the current configuration of the machine as

$$(\mathtt{ab}q\mathtt{baa}, \mathtt{Za}q\mathtt{b}, \mathtt{Z}q\mathtt{acca}, \mathtt{aXY}q\mathtt{Zbc}).$$

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.

To go along with our start configuration, 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 7 (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.

Finally, 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 $L(\mathcal{M})$.

2.2 Nondeterministic Turing Machines

Each time we execute a computation on a deterministic Turing machine on the same input, we will follow the same sequence of configurations and end up with the same output. Sometimes, this is desirable behaviour, since it allows us to trace the behaviour of an algorithm—indeed, it's so desirable that we included determinism as the third condition of Definition 1!

At other times, though, we might not want determinism because it imposes on our computation a behaviour that is too stringent. Behaving in an entirely deterministic manner might make certain computations take a long time to complete; say, if we need to check every combination sequentially in a given set, or if we need to follow every possible path through a graph.

Fortunately, we can make a small modification to the Turing machine model that effectively allows it to "make guesses" as it runs. These "guesses" allow the Turing machine to produce potentially different computations and outputs on the same input, which renders the machine *nondeterministic*.

Definition 8 (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 4 except for the transition function, which is

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

We can see that the only change needed to make a Turing machine nondeterministic is to allow the transition function δ to be multivalued: instead of mapping to exactly one tuple of state, tape symbol, and input head direction, we map to the power set $\mathcal{P}\left(Q \times \Gamma \times \{L, R\}\right)$. Thus, reading the same tape symbol from the same state could lead to potentially many outcomes, and the nondeterministic Turing machine can "guess" which of these outcomes it should follow. In practice, though, the machine follows all of these outcomes at once in a manner that simulates unlimited parallelism.

As you might expect, we can modify the definition of our k-tape Turing machine to be nondeterministic in a similar way.

Definition 9 (k-tape nondeterministic Turing machine)**.** A k-tape 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 5 except for the transition function, which is

$$\delta : (Q \setminus \{q_{\text{accept}}, q_{\text{reject}}\}) \times \Gamma^k \to \mathcal{P}\left(Q \times \Gamma^{k-1} \times \{L, R\}^k\right).$$

Nondeterminism also affects the definitions of configurations, computations, and accepting computations, though only in a minor way, so we will only discuss each change at a high level.

While configurations themselves remain the same in a nondeterministic computation, it is no longer the case that one single configuration must yield one other single configuration. Instead, some configuration in a state q and reading symbols $\sigma_1, \ldots, \sigma_k$ on each of the k tapes may yield any configuration corresponding to an element in the set $\delta(q, \sigma_1, \ldots, \sigma_k)$ defined by the transition function.

Likewise, when we speak of an accepting computation on a nondeterministic Turing machine, we are no longer speaking of a single sequence of configurations leading to an accepting configuration. Instead, since nondeterministic Turing machines make "guesses" and follow all outcomes in parallel, we must consider all sequences of configurations originating from the start configuration. If just one of these sequences halts in an accepting configuration, then we say that the entire nondeterministic computation is an accepting computation.