3

Turing Machines

A Turing machine is essentially a deterministic finite automaton equipped with updatable storage. This updatable storage is represented by an array of cells, unbounded to the right. The content of that array is accessible via a pointer: the pointer points to a specific cell, and can be moved to the left and to the right. Only the cell pointed to by that point can be read or written. For historical reasons, this array is called a *tape*, and the pointer is called the *tape head*. Instead of an input string that gets scanned from left to right automatically as the machine progresses, the input string for a Turing machine is initially provided on the tape, and the machine must read from it explicitly.

Formal Definitions

A *deterministic Turing machine* is a tuple

$$M = (Q, \Sigma, \Gamma, \vdash, \sqcup, \delta, s, acc, rej)$$

where

- Q is a finite set of *states*
- Σ is a finite *input alphabet*
- Γ is a finite *tape alphabet* ($\Sigma \subseteq \Gamma$)
- $\vdash$ is the *leftmost marker symbol* ($\vdash \in \Gamma \backslash \Sigma$)

- $\sqcup$ is the *blank symbol* indicating an empty tape cell ($\sqcup \in \Gamma \backslash \Sigma$)
- δ is the *transition function*, $\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R\}$
- s is the *start state*
- *acc* is the *accept state*
- *rej* is the *reject state*

The transition function takes a state in Q and a symbol in tape alphabet Γ (intuitively, the symbol written in the tape cell where the tape head is), and tells you the new state to which the machine transitions, the symbol to write in the cell where the tape head is, and then whether to move the tape head to the left (L) or to the right (R).

The tape is infinite to the right. In its first (leftmost) cell, we assume that the leftmost marker $\vdash$ is written. Empty cells are assumed to contain the symbol $\sqcup$. We assume that the machine can never move the tape head to the left of the leftmost marker. (This is enforced by having the transition function always make the tape head move right upon reading the leftmost marker symbol.)

Examples

A Turing machine M computes as follows when trying to accept string $w = a_1 \dots a_k \in \Sigma^*$:

1. Put $\vdash\ a_1\ \dots\ a_k$ on the leftmost cells of the tape.
2. The tape head is initially on the leftmost cell containing $\vdash$.
3. The state is initially s, the start state.
4. If the machine is in state p and the symbol in the cell where the tape head is is a, and $\delta(p, a) = (q, b, d)$, then the machine writes b in the cell where the tape head is, moves the tape head in direction d, and moves to state q.
5. Repeat step 4 until the state is either *acc* (in which case the machine accepts) or *rej* (in which case the machine rejects).

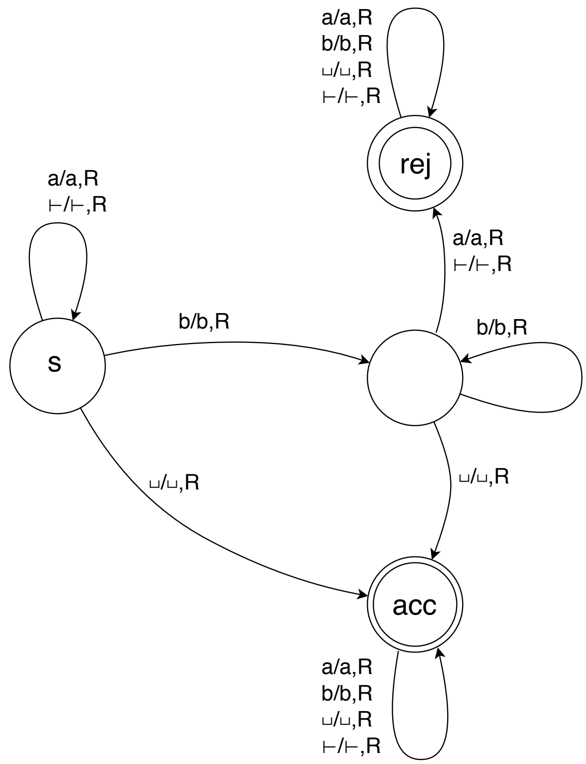

Figure 3.1: Turing machine accepting $\{\mathsf{a}^m\mathsf{b}^n \mid m, n \geq 0\}$

We say Turing machine M *accepts* w if M reaches the accept state starting with $\vdash w$ on its input tape. It *rejects* w if M reaches the reject state starting with $\vdash w$ on its input tape. Turing machine M *halts on input* w if M either accepts w or rejects w. A Turing machine need not halt on an input — it may step forever, never reaching either the accept or reject state.

The *language accepted by a Turing machine* M is the set of strings it accepts, that is,

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

Figure 3.1 gives a simple Turing machine accepting the language described by regular expression $\mathsf{a}^*\mathsf{b}^*$, which is of course a regular language. As usual, we simply draw the diagram of state transitions, where a transition of the form $\delta(p, a) = (q, b, d)$ is drawn as an arrow between states p and q labelled by $a/b, d$. The tape alphabet is given when unclear.

Figure 3.2 gives an already more complex Turing machine accepting $\{\mathtt{a}^n\mathtt{b}^n \mid n \geq 0\}$, which is not regular. The machine works in two phases. First, it scans the input string from left to right to ensure it is of the form $\mathtt{a}^m\mathtt{b}^n$ for some m, n. If not, then it goes to a sink state. If it is of that form, then the tape head is rewound to the left, and finds first an a and then a b, replacing both by a new tape symbol X. The tape head is rewound to the left again, and another a and another b are crossed out, skipping over previous crossed out symbols. If a b is found when an a was looked for, or no b found after an a is found, then the machine transitions to a sink state, otherwise, it accepts. The tape alphabet is $\{\vdash, \sqcup, \mathtt{a}, \mathtt{b}, \mathtt{X}\}$.

One thing to observe about these Turing machines. Since the transition function of a Turing machine is a function, it needs to be defined for every combination of state and tape symbol. This means, in particular, that every state (including the accept and reject states) need to have a transition out of it for every tape symbol. By convention, when the diagram does not describe a transition, then that transition just goes to the reject state. (Interestingly, my samples Turing machines in these notes do in fact describe all transitions.) That reject state always exists, since it is part of the description of a Turing machine, even if there are no transitions going to it.

It is an easy exercise to modify the Turing machine in Figure 3.2 to accept the language $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\}$.

Defining Execution

A configuration is a snapshot of the execution of a Turing machine. To describe the Turing machine at any point in its execution, we need to give: the content of the tape, the position of the tape head, and the current state of the machine.

This information can be represented by a triple (q, u, i), where the current state of the machine is q, u is the content of the tape from its first position until a point where the rest of the tape is filled with only blank symbols, and i is the position of the tape head (where the left-most position on the tape is 0). We require that u has sufficient length so that i represents a valid position in u — clearly, string u can always be padded on the right with blank symbols to make it long enough and still remain a configuration of the machine.

Thus, a configuration is a tuple in $Q \times \Gamma^* \times \mathbb{N}$.

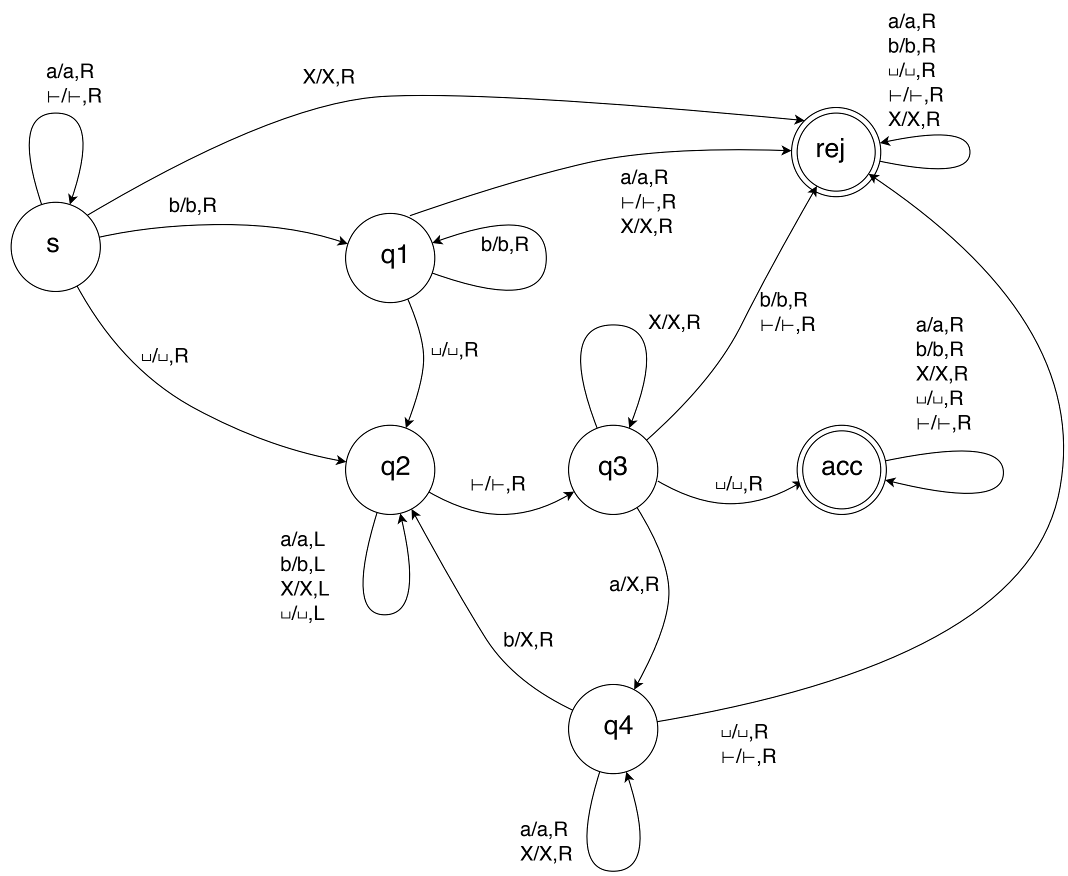

Figure 3.2: Turing machine accepting $\{\mathtt{a}^n\mathtt{b}^n \mid n \geq 0\}$

Fix a Turing machine $M = (Q, \Sigma, \Gamma, \vdash, \sqcup, \delta, s, acc, rej)$.

A *starting configuration* for M is a configuration of the form $(s, \vdash\ w, 0)$ for some input string w.

An *accepting configuration* for M is a configuration of the form (acc, u, i) for some u and i.

A *rejecting configuration* for M is a configuration of the form (rej, u, i) for some u and i.

A *halting configuration* is a configuration that is either accepting or rejecting.

We define a step relation between configurations, written $C \Longrightarrow C'$, that describes how configurations evolve as the Turing machine computes and transitions between states. The step relation is defined by the following rules:

$$\begin{array}{ll} (p, a_0 a_1 \ldots a_k, i) \Longrightarrow (q, a_0 a_1 \ldots a_{i-1} b a_{i+1} \ldots a_k, i+1) & \text{if } \delta(p, a_i) = (q, b, R) \\ (p, a_1 \ldots a_k, i) \Longrightarrow (q, a_1 \ldots a_{i-1} b a_{i+1} \ldots a_k, i-1) & \text{if } \delta(p, a_i) = (q, b, L) \end{array}$$

(The second transition shows why the restriction on δ that the tape head never moves left on $\vdash$ which is always at position 0 on the tape is necessary to keep the position from becoming negative.)

We define $\Longrightarrow^*$ as the reflexive transitive closure of $\Longrightarrow$: $C \Longrightarrow^* C'$ if either $C = C'$ or there exists $k \geq 0$ and $C_1, \ldots, C_k$ such that $C \Longrightarrow C_1 \Longrightarrow \ldots \Longrightarrow C_k \Longrightarrow C'$.

Formally, M accepts w if $(s, \vdash\ w, 0) \Longrightarrow^* C_{acc}$ for some accepting configuration C_{acc}.

Similarly, M rejects w if $(s, \vdash\ w, 0) \Longrightarrow^* C_{rej}$ for some rejecting configuration C_{rej}.

Here is the sequence of configurations showing how the machine in Figure 3.2 accepts aabb:

$$\begin{array}{rcl} (s, \vdash \mathsf{aabb}, 0) & \Longrightarrow & (s, \vdash \mathsf{aabb}, 1) \\ & \Longrightarrow & (s, \vdash \mathsf{aabb}, 2) \\ & \Longrightarrow & (s, \vdash \mathsf{aabb}, 3) \\ & \Longrightarrow & (q_1, \vdash \mathsf{aabb}, 4) \\ & \Longrightarrow & (q_1, \vdash \mathsf{aabb}\sqcup, 5) \\ & \Longrightarrow & (q_2, \vdash \mathsf{aabb}\sqcup\sqcup, 6) \\ & \Longrightarrow & (q_2, \vdash \mathsf{aabb}\sqcup\sqcup, 5) \end{array}$$

$$
\begin{aligned}
&\Longrightarrow (q_2, \vdash \mathtt{aabb}_{\sqcup\sqcup}, 4)\\
&\Longrightarrow (q_2, \vdash \mathtt{aabb}_{\sqcup\sqcup}, 3)\\
&\Longrightarrow (q_2, \vdash \mathtt{aabb}_{\sqcup\sqcup}, 2)\\
&\Longrightarrow (q_2, \vdash \mathtt{aabb}_{\sqcup\sqcup}, 1)\\
&\Longrightarrow (q_2, \vdash \mathtt{aabb}_{\sqcup\sqcup}, 0)\\
&\Longrightarrow (q_3, \vdash \mathtt{aabb}_{\sqcup\sqcup}, 1)\\
&\Longrightarrow (q_4, \vdash \mathtt{Xabb}_{\sqcup\sqcup}, 2)\\
&\Longrightarrow (q_4, \vdash \mathtt{Xabb}_{\sqcup\sqcup}, 3)\\
&\Longrightarrow (q_2, \vdash \mathtt{XaXb}_{\sqcup\sqcup}, 4)\\
&\Longrightarrow (q_2, \vdash \mathtt{XaXb}_{\sqcup\sqcup}, 3)\\
&\Longrightarrow (q_2, \vdash \mathtt{XaXb}_{\sqcup\sqcup}, 2)\\
&\Longrightarrow (q_2, \vdash \mathtt{XaXb}_{\sqcup\sqcup}, 1)\\
&\Longrightarrow (q_2, \vdash \mathtt{XaXb}_{\sqcup\sqcup}, 0)\\
&\Longrightarrow (q_3, \vdash \mathtt{XaXb}_{\sqcup\sqcup}, 1)\\
&\Longrightarrow (q_3, \vdash \mathtt{XaXb}_{\sqcup\sqcup}, 2)\\
&\Longrightarrow (q_4, \vdash \mathtt{XXXb}_{\sqcup\sqcup}, 3)\\
&\Longrightarrow (q_4, \vdash \mathtt{XXXb}_{\sqcup\sqcup}, 4)\\
&\Longrightarrow (q_2, \vdash \mathtt{XXXX}_{\sqcup\sqcup}, 5)\\
&\Longrightarrow (q_2, \vdash \mathtt{XXXX}_{\sqcup\sqcup}, 4)\\
&\Longrightarrow (q_2, \vdash \mathtt{XXXX}_{\sqcup\sqcup}, 3)\\
&\Longrightarrow (q_2, \vdash \mathtt{XXXX}_{\sqcup\sqcup}, 2)\\
&\Longrightarrow (q_2, \vdash \mathtt{XXXX}_{\sqcup\sqcup}, 1)\\
&\Longrightarrow (q_2, \vdash \mathtt{XXXX}_{\sqcup\sqcup}, 0)\\
&\Longrightarrow (q_3, \vdash \mathtt{XXXX}_{\sqcup\sqcup}, 1)\\
&\Longrightarrow (q_3, \vdash \mathtt{XXXX}_{\sqcup\sqcup}, 2)\\
&\Longrightarrow (q_3, \vdash \mathtt{XXXX}_{\sqcup\sqcup}, 3)\\
&\Longrightarrow (q_3, \vdash \mathtt{XXXX}_{\sqcup\sqcup}, 4)\\
&\Longrightarrow (q_3, \vdash \mathtt{XXXX}_{\sqcup\sqcup}, 5)\\
&\Longrightarrow (acc, \vdash \mathtt{XXXX}_{\sqcup\sqcup}, 6)
\end{aligned}
$$

and this last configuration $(acc, \vdash \mathtt{XXXX}_{\sqcup\sqcup}, 6)$ is an accepting configuration, and thus w is accepted.

Enumerable and Computable Languages

Language A is *Turing-enumerable*[1] if there exists a Turing machine M such that $L(M) = A$.

Turing machine M is *total* if it halts on every input string in Σ^*.

Language A is *Turing-computable*[2] if there exists a *total* Turing machine M such that $L(M) = A$.

(We sometimes say Turing machine M *decides* language A when M is total and M accepts A.)

Every computable language is also enumerable, since total Turing machines are just a special class of Turing machines.

Recasting the definition, a language is computable if and only if there a Turing machine M that accepts every string $w \in L(M)$ and that rejects every string $w \notin L(M)$.

It is easy to see that every regular language is computable; it suffices to show that DFAs can be simulated by Turing machines that always halt.

As we shall see, there are languages that are enumerable but not computable. And similarly, languages that are not even enumerable. Given how general and expressive Turing machines are, this is somewhat surprising.

Pseudocode Descriptions of Turing Machines.

Giving complete descriptions of Turing machines becomes painful for anything but the simplest of machines. Therefore, we will often resort to a pseudocode description of the behavior of Turing machines, focusing on tape head movement, and symbol replacement. For example, here is a reasonable description of a total version of the Turing machine accepting $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\}$.

---

[1] also called *semi-decidable* or *recursively enumerable*

[2] also called *decidable* or *recursive*

On input w:

1. Scan tape from left to right, checking that `a`s follow `b`s follow `c`s. Reject if not.
2. Move tape head back to leftmost position.
3. Scan from left to right, replacing the first `a` encountered with `X`, then the first `b` encountered with `X`, then the first `c` encountered with `X`.
4. If no `a`, `b`, or `c` encountered, accept.
5. If any of `a`, `b`, or `c` is not encountered, reject.
6. Go back to step 2.

It must be the case that every step in the pseudocode description of a Turing machine should be easily translatable into a set of transitions between states.

Multitape Turing Machines.

Turing machines are simple. Despite their simplicity, they are powerful. But while their simplicity is useful for proving things about them, it is painful when the time comes to design Turing machines to recognize or decide specific languages.

Variants of Turing machines exist that are simpler to use, but that still lead to the same class of enumerable (and computable) languages.

For instance, while a Turing machine has a single tape, we can imagine working with a Turing machine with multiple tapes. Having multiple tapes (with independent tape heads) means that we can use some of those tapes as temporary storage, or scratch pad to perform calculations, and so on. Having more than one tape is handy.

We can define multitape Turing machines easily enough. A multitape Turing machine (with k tapes) is a tuple $M = (Q, \Sigma, \Gamma, \vdash, \sqcup, \delta, s, acc, rej)$ defined exactly as a one-tape Turing machine, except that the transition relation δ has the form

$$\delta : Q \times \Gamma^k \longrightarrow Q \times \Gamma^k \times \{L, R, S\}^k.$$

(We also allow a tape head to remain in place, indicated by a direction S.) Intuitively, $\delta(p, \langle a_1, \ldots, a_k \rangle) = (q, \langle b_1, \ldots, b_k \rangle, \langle d_1, \ldots, d_k \rangle)$ says that when in state p and when a_i is on tape i under tape i's tape head, then M can transition to state q, writing b_i on tape i under its tape head, and moving each tape head in direction d_i. As for Turing machines, we assume that every tape has a leftmost marker in its leftmost cell, and that the machine cannot move the tape head to the left when on the leftmost marker.

This can be formalized using the notion of configuration, as with standard Turing machines. A *k-tape configuration* is a tuple $(q, u_1, \ldots, u_k, i_1, \ldots, i_k)$ where q is a state to the multitape Turing machine, $u_1, \ldots, u_k$ are the contents of the k tapes, and $i_1, \ldots, i_k$ are the positions of the tape heads on the respective tapes. The starting configuration of the machine with input w is simply $(s, \vdash\ w, \vdash, \ldots, \vdash, 0, \ldots, 0)$, where s is the start state of the machine; a configuration is accepting (resp., rejecting) if the state is the accept (resp., reject) state of the machine. It is an easy exercise to define the step relation $C \Longrightarrow C'$, and from it we can define $C \Longrightarrow^* C'$ as for Turing machines, so that M accepting and rejecting a string w is defined as for Turing machines using the $\Longrightarrow^*$ relation. The language of a multitape Turing machine M is just the set of strings accepted by M. A multitape Turing machine is total if it halts on all inputs.

We say a language is *enumerable by a multitape Turing machine* if there is a multitape Turing machine that accepts it. It is *computable by a multitape Turing machine* if there is a total multitape Turing machine that accepts it.

Multitape Turing machines, while convenient, do not give us more recognizable or computable languages.

Theorem: A language is enumerable (resp., computable) by a multitape Turing machine if and only if it is Turing-enumerable (resp., Turing-computable).

We first prove the reverse direction: if a language is (Turing-)enumerable, it is enumerable by a multitape Turing machine. If a language is enumerable, there is a Turing machine that accepts it. A Turing machine is just a multitape Turing machine with a single tape (you can check the definitions above agree in that case), and so there is a multitape Turing machine (with one tape) that accepts it. Same things if the language is computable.

The forward direction is more interesting. Suppose a language A is enumerable by a multitape Turing machine, say M_{mt}, with k tapes. To show it is Turing-enumerable, we need to show that there is a Turing machine M_A that accepts A. We build this Turing machine M_A by essentially simulating what M_{mt} is doing, but using a single tape. The idea is to put all the tapes that M_{mt} would use on a single tape, separated by a special tape symbol # that marks where each simulated tape ends and a new simulated tape begins. We indicate where each tape head is by using marked tape symbols of the form $\widehat{\mathtt{a}}$, adding those marked symbols to the tape alphabet.

When M_A runs, it simulates every step of Turing machine M_{mt}: whenever

M_{mt} would take a transition in state q, M_A takes a sequence of transitions from a state $\overline{q}$ corresponding to q that first determine what transition M_{mt} would make, updates the tapes accordingly, and then transitions to a new state $\overline{q'}$ that corresponds to the state q' that M_{mt} transitions to.

Given M_{mt}, M_A is defined as follows:

On input w:

1. Rewrite $\vdash\ a_1 \ldots a_n$ on the tape into $\vdash\ \widehat{\#}\, a_1 \ldots a_n\, \widehat{\#}\, \sqcup\, \widehat{\#}\, \sqcup\, \widehat{\#} \ldots \#$ ($k + 1$ #s in total)
2. Simulate a move of M_{mt}:

2a. Scan tape from left to right, noting the marked symbols until the $(k + 1)^{\text{th}}$ #

2b. scan tape from left to right, replacing marked symbols, and moving the mark left or right according to M_{mt} (treat # as $\vdash$)

2c. if the mark moves right onto a #, shift whole tape right from that position and write $\widehat{\sqcup}$ in the cell

3. If M_{mt} accepts, accept; if M_{mt} rejects, reject
4. Go to step 2

Observe that M_A is total exactly when M_{mt} is total; this means that the construction in fact shows that if A is enumerable (resp., computable) by a multitape Turing machine, it is Turing-enumerable (resp., Turing-computable).

Church-Turing Thesis

What the multitape Turing machine example from the notes from last lecture shows is that the definition of computable languages is quite robust: extensions to Turing machines do not add any expressive power to Turing machines. Intuitively, standard Turing machines are powerful enough to be able to simulate any variant of Turing machines.

In fact, Turing machines are powerful enough, as far as we know, to simulate any model of computation, or any programming language. This thesis is known as the *Church-Turing thesis*:

The Church-Turing Thesis: *Any feasible model of computation can be simulated by Turing machines.*

The notion *feasible* is kept vague, partly because it cannot really be defined. But it captures the idea of a model of computation that is, for instance, physically realizable—it does not depend on the ability to perform an infinite amount of computation in a finite amount of time, or requiring an infinite amount of space.

Because *feasible* cannot be defined, the Church-Turing thesis is not a theorem. Nevertheless, we have a lot of evidence that the thesis is true.

While it may seem surprising at first blush that a Turing machine can in fact simulate any model of computation, it really should not be. It should be clear, for starters, that computers can simulate any model of computation. It then only remains to argue that a Turing machine can simulate a computer. But a computer is really just a CPU with a bunch of attendant hardware to make it interact with the world, and a CPU is really just a sort of Turing machines. Putting it the other way around, it is not difficult to imagine simulating a CPU with a Turing machine: the alphabet of the Turing machine includes $\hat{0}$ and $\hat{1}$, which are the basic values that a CPU works with; the registers of the CPU can live on the tape; the finite number of gates of the CPU can be modeled by the finite control of the Turing machine; and the memory that the CPU has access to, which is finite, can be against stored on the tape. While a CPU addresses memory by indexing into it, the Turing machine would have to move the tape head to the appropriate cell corresponding to the memory location, but the difference is merely one of efficiency.[3]

The Church-Turing thesis makes the distinctions between computable languages and languages that are *not* computable relevant for the working programmer. Recall that languages over an alphabet Σ are just functions from Σ^* to the Booleans. A language A is computable if it can be accepted by a total Turing machine; in other words, a function A from Σ^* to the Booleans is computable if you can come up with a total Turing machines that says yes to a string w exactly when the function A maps w to `true`. A language is not computable (we'll call those uncomputable) if there is no total Turing machine that accepts it. Because of the Church-Turing thesis, if a language is uncomputable, then not only is there no total Turing machine that accepts it, there is also no program in any of your favorite programming languages that implements the function corresponding to the language: if there was, then you could also write a Turing machine that simulates that program, and this would contradict undecidability. Thus, if a language is uncomputable, there can be no program that implements the corresponding function, period. The notion of decidability, even though it seems like a technical notion having to do with Turing machines only, in fact impacts

[3]The Church-Turing thesis never claimed that Turing machines can simulate any model of computation *efficiently*, just that they can simulate it.

programming in general. Being able to determine what languages are computable and which aren't determines which problems can be solved using computer programs, and which can't.