Chapter 5

Turing Machines and Computability

We saw hints at the end of the previous chapter that “computation” is a more general concept than we might have thought. General grammars, which at first encounter don’t seem to have much to do with algorithms or computing, turn out to be able to do things that are similar to the tasks carried out by computer programs. In this chapter, we will see that general grammars are precisely equivalent to computer programs in terms of their computational power, and that both are equivalent to a particularly simple model of computation known as a Turing machine. We shall also see that there are limits to what can be done by computing.

5.1 Turing Machines

Historically, the theoretical study of computing began before computers existed. One of the early models of computation was developed in the 1930s by the British mathematician, Alan Turing, who was interested in studying the theoretical abilities and limitations of computation. His model for computation is a very simple abstract computing machine which has come to be known as a Turing machine. While Turing machines are not applicable in the same way that regular expressions, finite-state automata, and grammars are applicable, their use as a fundamental model for computation means that every computer scientist should be familiar with them, at least in a general way.

A Turing machine is really not much more complicated than a finite-
state automaton or a pushdown automaton. Like a FSA, a Turing machine has a finite number of possible states, and it changes from state to state as it computes. However, a Turing machine also has an infinitely long tape that it can use for input and output. The tape extends to infinity in both directions. The tape is divided into cells, which are in one-to-one correspondence with the integers, \( \mathbb{Z} \). Each cell can either be blank or it can hold a symbol from a specified alphabet. The Turing machine can move back and forth along this tape, reading and writing symbols and changing state. It can read only one cell at a time, and possibly write a new value in that cell. After doing this, it can change state and it can move by one cell either to the left or to the right. This is how the Turing machine computes.

To use a Turing machine, you would write some input on its tape, start the machine, and let it compute until it halts. Whatever is written on the tape at that time is the output of the computation.

Although the tape is infinite, only a finite number of cells can be non-blank at any given time. If you don’t like the idea of an infinite tape, you can think of a finite tape that can be extended to an arbitrarily large size as the Turing machine computes: If the Turing machine gets to either end of the tape, it will pause and wait politely until you add a new section of tape. In other words, it’s not important that the Turing machine have an infinite amount of memory, only that it can use as much memory as it needs for a given computation, up to any arbitrarily large size. In this way, a Turing machine is like a computer that can ask you to buy it a new disk drive whenever it needs more storage space to continue a computation.

A given Turing machine has a fixed, finite set of states. One of these states is designated as the start state. This is the state in which the Turing machine begins a computation. Another special state is the halt state. The Turing machine’s computation ends when it enters its halt state. It is possible that a computation might never end because the machine never enters the halt state. This is analogous to an infinite loop in a computer program.

---

1In fact, Turing machines can be shown to be equivalent in their computational power to pushdown automata with two independent stacks.

2The tape of a Turing machine can be used to store arbitrarily large amounts of information in a straightforward way. Although we can imagine using an arbitrary amount of memory with a computer, it’s not so easy. Computers aren’t set up to keep track of unlimited amounts of data. If you think about how it might be done, you probably won’t come with anything better than an infinite tape. (The problem is that computers use integer-valued addresses to keep track of data locations. If a limit is put on the number of bits in an address, then only a fixed, finite amount of data can be addressed. If no limit is put on the number of bits in an address, then we are right back to the problem of storing an arbitrarily large piece of data—just to represent an address!)
At each step in its computation, the Turing machine reads the contents of the tape cell where it is located. Depending on its state and the symbol that it reads, the machine writes a symbol (possibly the same symbol) to the cell, moves one cell either to the left or to the right, and (possibly) changes its state. The output symbol, direction of motion, and new state are determined by the current state and the input symbol. Note that either the input symbol, the output symbol, or both, can be blank. A Turing machine has a fixed set of rules that tell it how to compute. Each rule specifies the output symbol, direction of motion, and new state for some combination of current state and input symbol. The machine has a rule for every possible combination of current state and input symbol, except that there are no rules for what happens if the current state is the halt state. Of course, once the machine enters the halt state, its computation is complete and the machine simply stops.

I will use the character # to represent a blank in a way that makes it visible. I will always use \( h \) to represent the halt state. I will indicate the directions, left and right, with \( L \) and \( R \), so that \( \{L, R\} \) is the set of possible directions of motion. With these conventions, we can give the formal definition of a Turing machine as follows:

**Definition 5.1.** A Turing machine is a 4-tuple \((Q, \Lambda, q_0, \delta)\), where:
1. \( Q \) is a finite set of states, including the halt state, \( h \).
2. \( \Lambda \) is an alphabet which includes the blank symbol, \#.
3. \( q_0 \in Q \) is the start state.
4. \( \delta: (Q \setminus \{h\}) \times \Lambda \to \Lambda \times \{L, R\} \times Q \) is the transition function. The fact that \( \delta(q, \sigma) = (\tau, d, r) \) means that when the Turing machine is in state \( q \) and reads the symbol \( \sigma \), it writes the symbol \( \tau \), moves one cell in the direction \( d \), and enters state \( r \).

Even though this is the formal definition, it’s easier to work with a transition diagram representation of Turing machines. The transition diagram for a Turing machine is similar to the transition diagram for a DFA. However, there are no “accepting” states (only a halt state). Furthermore, there must be a way to specify the output symbol and the direction of motion for each step of the computation. We do this by labeling arrows with notations of the form \((\sigma, \tau, L)\) and \((\sigma, \tau, R)\), where \( \sigma \) and \( \tau \) are symbols in the Turing machine’s alphabet. For example,
indicates that when the machine is in state $q_0$ and reads an $a$, it writes a $b$, moves left, and enters state $h$.

Here, for example, is a transition diagram for a simple Turing machine that moves to the right, changing $a$’s to $b$’s and vice versa, until it finds a $c$. It leaves blanks (’#’s) unchanged. When and if the machine encounters a $c$, it moves to the left and halts:

\[
\begin{array}{ccc}
(a,b,R) & & (c,c,L) \\
(b,a,R) & & \\
(q_0) & & h \\
(\#,\#,R) & & \\
\end{array}
\]

To simplify the diagrams, I will leave out any transitions that are not relevant to the computation that I want the machine to perform. You can assume that the action for any omitted transition is to write the same symbol that was read, move right, and halt.

For example, shown below is a transition diagram for a Turing machine that makes a copy of a string of $a$’s and $b$’s. To use this machine, you would write a string of $a$’s and $b$’s on its tape, place the machine on the first character of the string, and start the machine in its start state, $q_0$. When the machine halts, there will be two copies of the string on the tape, separated by a blank. The machine will be positioned on the first character of the leftmost copy of the string. Note that this machine uses $c$’s and $d$’s in addition to $a$’s and $b$’s. While it is copying the input string, it temporarily changes the $a$’s and $b$’s that it has copied to $c$’s and $d$’s, respectively. In this way it can keep track of which characters it has already copied. After the string has been copied, the machine changes the $c$’s and $d$’s back to $a$’s and $b$’s before halting.
In this machine, state $q_0$ checks whether the next character is an $a$, a $b$, or a $#$ (indicating the end of the string). States $q_1$ and $q_2$ add an $a$ to the end of the new string, and states $q_3$ and $q_4$ do the same thing with a $b$. States $q_5$ and $q_6$ return the machine to the next character in the input string. When the end of the input string is reached, state $q_7$ will move the machine back to the start of the input string, changing $c$’s and $d$’s back to $a$’s and $b$’s as it goes. Finally, when the machine hits the $#$ that precedes the input string, it moves to the right and halts. This leave it back at the first character of the input string. It would be a good idea to work through the execution of this machine for a few sample input strings. You should also check that it works even for an input string of length zero.

Our primary interest in Turing machines is as language processors. Suppose that $w$ is a string over an alphabet $\Sigma$. We will assume that $\Sigma$ does not contain the blank symbol. We can use $w$ as input to a Turing machine $M = (Q, \Lambda, q_0, \delta)$ provided that $\Sigma \subseteq \Lambda$. To use $w$ as input for $M$, we will write $w$ on $M$’s tape and assume that the remainder of the tape is blank. We place the machine on the cell containing the first character of the string, except that if $w = \varepsilon$ then we simply place the machine on a completely blank tape. Then we start the machine in its initial state, $q_0$, and see what computation it performs. We refer to this setup as “running $M$ with input $w$.”
When $M$ is run with input $w$, it is possible that it will just keep running forever without halting. In that case, it doesn’t make sense to ask about the output of the computation. Suppose however that $M$ does halt on input $w$. Suppose, furthermore, that when $M$ halts, its tape is blank except for a string $x$ of non-blank symbols, and that the machine is located on the first character of $x$. In this case, we will say that “$M$ halts with output $x$.” In addition, if $M$ halts with an entirely blank tape, we say that “$M$ halts with output $\varepsilon$.” Note that when we run $M$ with input $w$, one of three things can happen: (1) $M$ might halt with some string as output; (1) $M$ might fail to halt; or (3) $M$ might halt in some configuration that doesn’t count as outputting any string.

The fact that a Turing machine can produce an output value allows us for the first time to deal with computation of functions. A function $f: A \rightarrow B$ takes an input value in the set $A$ and produces an output value in the set $B$. If the sets are sets of strings, we can now ask whether the values of the function can be computed by a Turing machine. That is, is there a Turing machine $M$ such that, given any string $w$ in the domain of $f$ as input, $M$ will compute as its output the string $f(w)$. If this is that case, then we say that $f$ is a Turing-computable function.

**Definition 5.2.** Suppose that $\Sigma$ and $\Gamma$ are alphabets that do not contain # and that $f$ is a function from $\Sigma^*$ to $\Gamma^*$. We say that $f$ is **Turing-computable** if there is a Turing machine $M = (Q, \Lambda, q_0, \delta)$ such that $\Sigma \subseteq \Lambda$ and $\Gamma \subseteq \Lambda$ and for each string $w \in \Sigma^*$, when $M$ is run with input $w$, it halts with output $f(w)$. In this case, we say that $M$ computes the function $f$.

For example, let $\Sigma = \{a\}$ and define $f: \Sigma^* \rightarrow \Sigma^*$ by $f(a^n) = a^{2n}$, for $n \in \mathbb{N}$. Then $f$ is Turing-computable since it is computed by this Turing machine:
We can also use Turing machines to define “computable languages.” There are actually two different notions of Turing-computability for languages. One is based on the idea of Turing-computability for functions. Suppose that \( \Sigma \) is an alphabet and that \( L \subseteq \Sigma^* \). The characteristic function of \( L \) is the function \( \chi_L : \Sigma^* \to \{0, 1\} \) defined by the fact that \( \chi_L(w) = 1 \) if \( w \in L \) and \( \chi_L(w) = 0 \) if \( w \notin L \). Note that given the function \( \chi_L \), \( L \) can be obtained as the set \( L = \{ w \in \Sigma^* | \chi_L(w) = 1 \} \). Given a language \( L \), we can ask whether the corresponding function \( \chi_L \) is Turing-computable. If so, then we can use a Turing machine to decide whether or not a given string \( w \) is in \( L \). Just run the machine with input \( w \). It will halt with output \( \chi_L(w) \). (That is, it will halt and when it does so, the tape will be blank except for a 0 or a 1, and the machine will be positioned on the 0 or 1.) If the machine halts with output 1, then \( w \in L \). If the machine halts with output 0, then \( w \notin L \).

**Definition 5.3.** Let \( \Sigma \) be an alphabet that does not contain \# and let \( L \) be a language over \( \Sigma \). We say that \( L \) is Turing-decidable if there is a Turing machine \( M = (Q, \Lambda, q_0, \delta) \) such that \( \Sigma \subseteq \Lambda \), \( \{0, 1\} \subseteq \Lambda \), and for each \( w \in \Sigma^* \), when \( M \) is run with input \( w \), it halts with output \( \chi_L(w) \). (That is, it halts with output 0 or 1, and the output is 0 if \( w \notin L \) and is 1 if \( w \in L \).) In this case, we say that \( M \) decides the language \( L \).

The second notion of computability for languages is based on the interesting fact that it is possible for a Turing machine to run forever, without ever halting. Whenever we run a Turing machine \( M \) with input \( w \), we can ask the question, will \( M \) ever halt or will it run forever? If \( M \) halts on input \( w \), we will say that \( M \) “accepts” \( w \). We can then look at all the strings over a given alphabet that are accepted by a given Turing machine. This leads to the notion of Turing-acceptable languages.
Definition 5.4. Let $\Sigma$ be an alphabet that does not contain $\#$, and let $L$ be a language over $\Sigma$. We say that $L$ is **Turing-acceptable** if there is a Turing machine $M = (Q, \Lambda, q_0, \delta)$ such that $\Sigma \subseteq \Lambda$, and for each $w \in \Sigma^*$, $M$ halts on input $w$ if and only if $w \in L$. In this case, we say that $M$ accepts the language $L$.

It should be clear that any Turing-decidable language is Turing-acceptable. In fact, if $L$ is a language over an alphabet $\Sigma$, and if $M$ is a Turing machine that decides $L$, then it is easy to modify $M$ to produce a Turing machine that accepts $L$. At the point where $M$ enters the halt state with output 0, the new machine should enter a new state in which it simply moves to the right forever, without ever halting. Given an input $w \in \Sigma^*$, the modified machine will halt if and only if $M$ halts with output 1, that is, if and only if $w \in L$.

**Exercises**

1. Let $\Sigma = \{a\}$. Draw a transition diagram for a Turing machine that computes the function $f: \Sigma^* \to \Sigma^*$ where $f(a^n) = a^{3n}$, for $n \in \mathbb{N}$. Draw a transition diagram for a Turing machine that computes the function $f: \Sigma^* \to \Sigma^*$ where $f(a^n) = a^{3n+1}$, for $n \in \mathbb{N}$.
2. Let $\Sigma = \{a, b\}$. Draw a transition diagram for a Turing machine that computes the function $f: \Sigma^* \to \Sigma^*$ where $f(w) = w^R$.
3. Suppose that $\Sigma$, $\Gamma$, and $\Xi$ are alphabets and that $f: \Sigma^* \to \Gamma^*$ and $g: \Gamma^* \to \Xi^*$ are Turing-computable functions. Show that $g \circ f$ is Turing-computable.
4. We have defined computability for functions $f: \Sigma^* \to \Gamma^*$, where $\Sigma$ and $\Gamma$ are alphabets. How could Turing machines be used to define computable functions from $\mathbb{N}$ to $\mathbb{N}$? (Hint: Consider the alphabet $\Sigma = \{a\}$.)
5. Let $\Sigma$ be an alphabet and let $L$ be a language over $\Sigma$. Show that $L$ is Turing-decidable if and only if its complement, $\overline{L}$, is Turing-decidable.
6. Draw a transition diagram for a Turing machine which decides the language $\{a^n b^n | n \in \mathbb{N}\}$. (Hint: Change the a’s and b’s to $\$$’s in pairs.) Explain in general terms how to make a Turing machine that decides the language $\{a^n b^n c^n | n \in \mathbb{N}\}$.
7. Draw a transition diagram for a Turing machine which decides the language $\{a^n b^m | n > 0 \text{ and } m \text{ is a multiple of } n\}$. (Hint: Erase $n$ b’s at a time.)
8. Based on your answer to the previous problem and the copying machine presented in this section, describe in general terms how you would build a Turing machine to decide the language $\{a^p | p \text{ is a prime number}\}$.
9. Let $g: \{a\}^* \to \{0, 1\}^*$ be the function such that for each $n \in \mathbb{N}$, $g(a^n)$ is the representation of $n$ as a binary number. Draw a transition diagram for a Turing machine that computes $g$. 

5.2 Computability

At this point, it would be useful to look at increasingly complex Turing machines, which compute increasingly complex functions and languages. Although Turing machines are very simple devices, it turns out that they can perform very sophisticated computations. In fact, any computation that can be carried out by a modern digital computer—even one with an unlimited amount of memory—can be carried out by a Turing machine. Although it is not something that can be proved, it is widely believed that anything that can reasonably be called “computation” can be done by a Turing machine. This claim is known as the Church-Turing Thesis.

We do not have time to look at enough examples to convince you that Turing machines are as powerful as computers, but the proof reduces to the fact that computers are actually fairly simple in their basic operation. Everything that a computer does comes down to copying data from one place to another, making simple comparisons between two pieces of data, and performing some basic arithmetic operations. It’s possible for Turing machines to do all these things. In fact, it’s possible to build a Turing machine to simulate the step-by-step operation of a given computer. Doing so proves that the Turing machine can do any computation that the computer could do, although it will, of course, work much, much more slowly.

We can, however, look briefly at some other models of computation and see how they compare with Turing machines. For example, there are various ways in which we might try to increase the power of a Turing machine. For example, consider a two-tape Turing machine that has two tapes, with a read/write head on each tape. In each step of its computation, a two-tape Turing machine reads the symbols under its read/write heads on both tapes. Based on these symbols and on its current state, it can write a new symbol onto each tape, independently move the read/write head on each tape one cell to the left or right, and change state.

It might seem that with two tapes available, two-tape Turing machines might be able to do computations that are impossible for ordinary one-tape machines. In fact, though, this is not the case. The reason, again, is simulation: Given any two-tape Turing machine, it is possible to build a one-tape Turing machine that simulates the step-by-step computation of the two-tape machine. Let $M$ be a two-tape Turing machine. To simulate $M$ with a one-tape machine, $K$, we must store the contents of both of $M$’s tapes on one tape, and we must keep track of the positions of both of $M$’s read/write heads. Let $@$ and $\$ be symbols that are not in the alphabet of $M$. The $@$ will be used to mark the position of a read/write head, and the $\$ will be
used to delimit the parts of \( K \)'s tape that represent the two tapes of \( M \). For example, suppose that one of \( M \)'s tapes contains the symbols “\( \text{abb}##\text{cca} \)” with the read/write head on the first \( b \), and that the other tape contains “\( 01\#111\#001 \)” with the read/write head on the final 1. This configuration would be represented on \( K \)'s tape as “\( $\text{a@bb}##\text{cca}$01\#111\#00$$1$”. To simulate one step of \( M \)'s computation, \( K \) must scan its entire tape, looking for the \( @ \)'s and noting the symbol to the right of each \( @ \). Based on this information, \( K \) can update its tape and its own state to reflect \( M \)'s new configuration after one step of computation. Obviously, \( K \) will take more steps than \( M \) and it will operate much more slowly, but this argument makes it clear that one-tape Turing machines can do anything that can be done by two-tape machines.

We needn’t stop there. We can imagine \( n \)-tape Turing machines, for \( n > 2 \). We might allow a Turing machine to have multiple read/write heads that move independently on each tape. We could even allow two or three-dimensional tapes. None of this makes any difference as far as computational power goes, since each type of Turing machine can simulate any of the other types.\(^3\)

We have used Turing machines to define Turing-acceptable languages and Turing-decidable languages. The definitions seem to depend very much on the peculiarities of Turing machines. But the same classes of languages can be defined in other ways. For example, we could use programs running on an idealized computer, with an unlimited amount of memory, to accept or decide languages. Or we could use \( n \)-tape Turing machines. The resulting classes of languages would be exactly the same as the Turing-acceptable and Turing-decidable languages.

We could look at other ways of specifying languages “computationally.” One of the most natural is to imagine a Turing machine or computer program that runs forever and outputs an infinite list of strings over some alphabet \( \Sigma \). In the case of Turing machines, it’s convenient to think of a two-tape Turing machine that lists the strings on its second tape. The strings in the list form a language over \( \Sigma \). A language that can be listed in this way is said to be recursively enumerable. Note that we make no assumption that the strings must be listed in any particular order, and we

\(^3\)We can also define non-deterministic Turing machines that can have several possible actions at each step. Non-deterministic Turing machines cannot be used to compute functions, since a function can have only one possible output for any given input. However, they can be used to accept languages. We say that a non-deterministic Turing machine accepts a language \( L \) is it is possible for the machine to halt on input \( w \) if and only if \( w \in L \). The class of languages accepted by non-deterministic Turing machines is the same as the class of languages accepted by deterministic Turing machines. So, non-determinism does not add any computational power.
allow the same string to appear in the output any number of times. Clearly, a recursively enumerable language is “computable” in some sense. Perhaps we have found a new type of computable language. But no—it turns out that we have just found another way of describing the Turing-acceptable languages. The following theorem makes this fact official and adds one more way of describing the same class of languages:

**Theorem 5.1.** Let $\Sigma$ be an alphabet and let $L$ be a language over $\Sigma$. Then the following are equivalent:

1. There is a Turing machine that accepts $L$.
2. There is a two-tape Turing machine that runs forever, making a list of strings on its second tape, such that a string $w$ is in the list if and only if $w \in L$.
3. There is a Turing-computable function $f : \{a\}^* \rightarrow \Sigma^*$ such that $L$ is the range of the function $f$.

While I will not give a complete, formal proof of this theorem, it’s not too hard to see why it is true. Consider a language that satisfies property 3. We can use the fact that $L$ is the range of a Turing-computable function, $f$, to build a two-tape Turing machine that lists $L$. The machine will consider each of the strings $a^n$, for $n \in \mathbb{N}$, in turn, and it will compute $f(a^n)$ for each $n$. Once the value of $f(a^n)$ has been computed, it can be copied onto the machine’s second tape, and the machine can move on to do the same with $a^{n+1}$. This machine writes all the elements of $L$ (the range of $f$) onto its second tape, so $L$ satisfies property 2. Conversely, suppose that there is a two-tape Turing machine, $M$, that lists $L$. Define a function $g : \{a\}^* \rightarrow \Sigma^*$ such that for $n \in \mathbb{N}$, $g(a^n)$ is the $(n + 1)^{th}$ item in the list produced by $M$. Then the range of $g$ is $L$, and $g$ is Turing-computable since it can be computed as follows: On input $a^n$, simulate the computation of $M$ until it has produced $n + 1$ strings, then halt, giving the $(n + 1)^{th}$ string as output. This shows that property 2 implies property 3, so these properties are in fact equivalent.

We can also check that property 2 is equivalent to property 1. Suppose that $L$ satisfies property 2. Consider a two-tape Turing machine, $T$, that lists the elements of $L$. We must build a Turing machine, $M$, which accepts $L$. We do this as follows: Given an input $w \in \Sigma^*$, $M$ will simulate the computation of $T$. Every time the simulated $T$ produces a string in the list, $M$ compares that string to $w$. If they are the same, $M$ halts. If $w \in L$, eventually it will be produced by $T$, so $M$ will eventually halt. If $w \notin L$, then it will never turn up in the list produced by $T$, so $M$ will never halt. Thus, $M$ accepts the language $L$. This shows that property 2 implies property 1.
The fact that property 1 implies property 2 is somewhat harder to see. First, we note that it is possible for a Turing machine to generate every possible string in $\Sigma^*$, one-by-one, in some definite order (such as order of increasing length, with something like alphabetical order for strings of the same length). Now, suppose that $L$ is Turing-acceptable and that $M$ is a Turing machine that accepts $L$. We need a two-tape Turing machine, $T$ that makes a list of all the elements of $L$. Unfortunately, the following idea does not work: Generate each of the elements in $\Sigma^*$ in turn, and see whether $M$ accepts it. If so, then add it to the list on the second tape. It looks like we have a machine that lists all the elements of $L$. The problem is that the only way for $T$ to “see whether $M$ accepts” a string is to simulate the computation of $M$. Unfortunately, as soon as we try this for any string $w$ that is not in $L$, the computation never ends! $T$ will get stuck in the simulation and will never even move on to the next string. To avoid this problem, $T$ must simulate multiple computations of $M$ at the same time. $T$ can keep track of these computations in different regions of its first tape (separated by $\$’s). Let the list of all strings in $\Sigma^*$ be $x_1, x_2, x_3, \ldots$. Then $T$ should operate as follows:

1. Set up the simulation of $M$ on input $x_1$, and simulate one step of the computation for $x_1$
2. Set up the simulation of $M$ on input $x_2$, and simulate one step of the computation for $x_1$ and one step of the computation for $x_2$.
3. Set up the simulation of $M$ on input $x_3$, and simulate one step of each of the computations, for $x_1, x_2$, and $x_3$.
   
   ...  

n. Set up the simulation of $M$ on input $x_n$, and simulate one step of each of the computations, for $x_1, x_2, \ldots, x_n$.

and so on. Each time one of the computations halts, $T$ should write the corresponding $x_i$ onto its second tape. Over the course of time, $T$ simulates the computation of $M$ for each input $w \in \Sigma^*$ for an arbitrary number of steps. If $w \in L$, the simulated computation will eventually end and $w$ will appear on $T$’s second tape. On the other hand, if $w \not\in L$, then the simulated computation will never end, so $w$ will not appear in the list. So we see that $T$ does in fact make a list of all the elements, and only the elements of $L$. This completes an outline of the proof of the theorem.

Next, we compare Turing machines to a completely different method of specifying languages: general grammars. Suppose $G = (V, \Sigma, P, S)$ is a general grammar and that $L$ is the language generated by $G$. Then there is a Turing machine, $M$, that accepts the same language, $L$. The alphabet
for $M$ will be $V \cup \Sigma \cup \{\$,\#\}$, where $\$ is a symbol that is not in $V \cup \Sigma$. (We also assume that $\#$ is not in $V \cup \Sigma$.) Suppose that $M$ is started with input $w$, where $w \in \Sigma^*$. We have to design $M$ so that it will halt if and only if $w \in L$. The idea is to have $M$ find each string that can be derived from the start symbol $S$. The strings will be written to $M$'s tape and separated by $\$’s. $M$ can begin by writing the start symbol, $S$, on its tape, separated from $w$ by a $\$. Then it repeats the following process indefinitely: For each string on the tape and for each production rule, $x \rightarrow y$, of $G$, search the string for occurrences of $x$. When one is found, add a $\$ to the end of the tape and copy the string to the end of the tape, replacing the occurrence of $x$ by $y$. The new string represents the results of applying the production rule $x \rightarrow y$ to the string. Each time $M$ produces a new string, it compares that string to $w$. If they are equal, then $M$ halts. If $w$ is in fact in $L$, then eventually $M$ will produce the string $w$ and will halt. Conversely, if $w$ is not in $L$, then $M$ will go on producing strings forever without ever finding $w$, so $M$ will never halt. This shows that, in fact, the language $L$ is accepted by $M$.

Conversely, suppose that $L$ is a language over an alphabet $\Sigma$, and that $L$ is Turing-acceptable. Then it is possible to find a grammar $G$ that generates $L$. To do this, it’s convenient to use the fact that, as discussed above, there is a Turing-computable function $f: \{a\}^* \rightarrow \Sigma$ such that $L$ is the range of $f$. Let $M = (Q, \Lambda, q_0, \delta)$ be a Turing machine that computes the function $f$. We can build a grammar, $G$, that imitates the computations performed by $M$. The idea is that most of the production rules of $G$ will imitate steps in the computation of $M$. Some additional rules are added to get things started, to clean up, and to otherwise bridge the conceptual gap between grammars and Turing machines.

The terminal symbols of $G$ will be the symbols from the alphabet, $\Sigma$. For the non-terminal symbols, we use: the states of $M$, every member of $\Lambda$ that is not in $\Sigma$, two special symbols $<$ and $>$, and two additional symbols $S$ and $A$. (We can assume that all these symbols are distinct.) $S$ will be the start symbol of $G$. As for production rules, we begin with the following three rules:

\begin{align*}
S &\rightarrow <q_0A> \\
A &\rightarrow aA \\
A &\rightarrow \varepsilon
\end{align*}

These rules make it possible to produce any string of the form $<q_0a^n>$. This is the only role that $S$ and $A$ play in the grammar. Once we’ve gotten rid of $S$ and $A$, strings of the remaining terminal and non-terminal symbols
represent configurations of the Turing machine $M$. The string will contain exactly one of the states of $M$ (which is, remember, one of the non-terminal symbols of $G$). This tells us which state $M$ is in. The position of the state-symbol tells us where $M$ is positioned on the tape: the state-symbol is located in the string to the left of the symbol on which $M$ is positioned. And the special symbols $<$ and $>$ just represent the beginning and the end of a portion of the tape of $M$. So, the initial string $<q_0a^n>$ represents a configuration in which $M$ is in its start state, and is positioned on the first $a$ in a string of $n$ $a$’s. This is the starting configuration of $M$ when it is run with input $a^n$.

Now, we need some production rules that will allow the grammar to simulate the computations performed by $M$. For each state $q_i$ and each symbol $\sigma \in \Lambda$, we need a production rule that imitates the transition rule $\delta(q_i, \sigma) = (\tau, d, q_j)$. If $d = R$, that is if the machine moves to the right, then all we need is the rule

$$q_i \sigma \rightarrow \tau q_j$$

This represents that fact that $M$ converts the $\sigma$ to a $\tau$, moves to the right, and changes to state $q_j$. If $d = L$, that is if the machine moves to the left, then we will need several rules—one rule for each $\lambda \in \Lambda$, namely

$$\lambda q_i \sigma \rightarrow q_j \lambda \tau$$

This rule says that $M$ changes the $\sigma$ to a $\tau$, moves left, and changes to state $q_j$. The $\lambda$ doesn’t affect the application of the rule, but is necessary to represent the fact that $M$ moves left.

Each application of one of these rules represents one step in the computation of $M$. There is one remaining requirement for correctly simulating $M$. Since $M$’s tape contains an infinite number of cells and we are only representing a finite portion of that tape, we need a way to add and remove #’s at the ends of the string. We can use the following four rules to do this:

$$< \rightarrow <\#$$
$$<\# \rightarrow <$$
$$> \rightarrow \#>$$
$$\#> \rightarrow >$$

These rules allow blank symbols to appear at the ends of the string when they are needed to continue the computation, and to disappear from the ends of the string whenever we like.
Now, suppose that \( w \) is some element of \( L \). Then \( w = f(a^n) \) for some \( n \in \mathbb{N} \). We know that on input \( a^n \), \( M \) halts with output \( w \). If we translate the computation of \( M \) into the corresponding sequence of production rules in \( G \), we see that for the grammar \( G \), \( <q_0a^n> \rightarrow^* <hw> \), where \( h \) is the halt state of \( M \). Since we already know that \( S \rightarrow^* <q_0a^n> \), for every \( n \in \mathbb{N} \), we see that in fact \( S \rightarrow^* <hw> \) for each \( w \in L \). We almost have it! We want to show that \( S \rightarrow^* w \). If we can just get rid of the \( < \), the \( h \), and the \( > \), we will have that \( <hw> \rightarrow^* w \) and we can then deduce that \( S \rightarrow^* w \) for each \( w \in L \), as desired. We can do this by adding just a few more rules to \( G \). We want to let the \( h \) eliminate the \( < \), move through the \( w \), and then eliminate the \( > \) along with itself. We need the rules
\[
<h \rightarrow h \\
h \rightarrow \varepsilon
\]
and, for each \( \sigma \in \Sigma \),
\[
h\sigma \rightarrow \sigma h
\]
We have constructed \( G \) so that it generates every string in \( L \). It is not difficult to see that the strings in \( L \) are in fact the only strings that are generated by \( G \). That is, \( L \) is precisely \( L(G) \).

We have now shown, somewhat informally, that a language \( L \) is Turing-acceptable if and only if there is a grammar \( G \) that generates \( L \). Even though Turing machines and grammars are very different things, they are equivalent in terms of their ability to describe languages. We state this as a theorem:

**Theorem 5.2.** A language \( L \) is Turing acceptable (equivalently, recursively enumerable) if and only if there is a general grammar that generates \( L \).

In this section, we have been talking mostly about recursively enumerable languages (also known as the Turing-acceptable languages). What about the Turing-decidable languages? We already know that if a language \( L \) is Turing-decidable, then it is Turing-acceptable. The converse is not true (although we won’t be able to prove this until the next section). However, suppose that \( L \) is a language over the alphabet \( \Sigma \) and that both \( L \) and its complement, \( \overline{L} = \Sigma^* \setminus L \), are Turing-acceptable. Then \( L \) is Turing-decidable.

For suppose that \( M \) is a Turing machine that accepts the language \( L \) and that \( M' \) is a Turing machine that accepts \( \overline{L} \). We must show that \( L \) is Turing-decidable. That is, we have to build a Turing machine \( T \) that
decides $L$. For each $w \in \Sigma^*$, when $T$ is run with input $w$, it should halt with output 1 if $w \in L$ and with output 0 if $w \notin L$. To do this, $T$ will simulate the computation of both $M$ and $M'$ on input $w$. (It will simulate one step in the computation of $M$, then one step in the computation of $M'$, then one step of $M$, then one step of $M'$, and so on.) If and when the simulated computation of $M$ halts, then $T$ will halt with output 1; since $M$ accepts $L$, this will happen if and only if $w \in L$. If and when the simulated computation of $M'$ halts, then $T$ will halt with output 0; since $M$ accepts $L$, this will happen if and only if $w \notin L$. So, for any $w \in \Sigma^*$, $T$ halts with the desired output. This means that $T$ does in fact decide the language $L$.

It is easy to prove the converse, and the proof is left as an exercise.

So we see that a language is Turing-decidable if and only if both it and its complement are Turing-acceptable. Since Turing-acceptability can be defined using other forms of computation besides Turing machines, so can Turing-decidability. For example, a language is Turing-decidable if and only if both it and its complement can be generated by general grammars. We introduced the term “recursively enumerable” as a synonym for Turing-acceptable, to get away from the association with a particular form of computation. Similarly, we define the term “recursive” as a synonym for Turing-decidable. That is, a language $L$ is said to be recursive if and only if it is Turing-decidable. We then have the theorem:

**Theorem 5.3.** Let $\Sigma$ be an alphabet and let $L$ be a language over $\Sigma$. Then $L$ is recursive if and only if both $L$ and its complement, $\Sigma^* \smallsetminus L$, are recursively enumerable.

**Exercises**

1. The language $L = \{a^m \mid m > 0\}$ is the range of the function $f(a^n) = a^{n+1}$. Design a Turing machine that computes this function, and find the grammar that generates the language $L$ by imitating the computation of that machine.

2. Complete the proof of Theorem 5.3 by proving the following: If $L$ is a recursive language over an alphabet $\Sigma$, then both $L$ and $\Sigma^* \smallsetminus L$ are recursively enumerable.

3. Show that a language $L$ over an alphabet $\Sigma$ is recursive if and only if there are grammars $G$ and $H$ such that the language generated by $G$ is $L$ and the language generated by $H$ is $\Sigma^* \smallsetminus L$.

4. This section discusses recursive languages and recursively enumerable languages. How could one define recursive subsets of $\mathbb{N}$ and recursively enumerable subsets of $\mathbb{N}$?

5. Give an informal argument to show that a subset $X \subseteq \mathbb{N}$ is recursive if and only if there is a computer program that prints out the elements of $X$ in increasing order.
5.3 The Limits of Computation

Recursively enumerable languages are languages that can be defined by computation. We have seen that there are many different models of computation—Turing machines, two-tape Turing machines, grammars, computer programs—but they all lead to the same class of languages. In fact, every computational method for specifying languages that has ever been developed produces only recursively enumerable languages. There is something about these languages—some pattern or property—that makes them “computable,” and it is some intrinsic property of the languages themselves, not some peculiarity of any given model of computation.

This is especially interesting since most languages are not recursively enumerable. Given an alphabet \( \Sigma \), there are uncountably many languages over \( \Sigma \), but only countably many of them are recursively enumerable. The rest—the vast majority—are not recursively enumerable. What can we say about all these non-recursively-enumerable languages? If the language \( L \) is not recursively enumerable, then there is no algorithm for listing the members of \( L \). It might be possible to define \( L \) by specifying some property that all its members satisfy, but that property can’t be computable. That is, there can be no computer program or Turing machine that tests whether a given string \( w \) has the property, since if there were, then we could write a program that lists the members of \( L \).

So, even though almost every language is non-recursively-enumerable, it’s difficult to find a particular language that is not recursively enumerable. Nevertheless, in this section we will find one such language. At that same time, we will find an example of a language that is recursively enumerable but not recursive. And we will discover some interesting limitations to the power of computation.

The examples that we will look at in this section involve Turing machines that work with other Turing machines as data. For this to work, we need a symbolic representation of Turing machines—a representation that can be written on the tape of another Turing machine. This will let us create two machines: First, a Turing machine that can generate Turing machines on demand by writing their symbolic representations on its tape. We will design a Turing machine \( G \) to do this. And second, a Turing machine that can simulate the computation of other Turing machines whose descriptions are written on its tape.

In order to do all this, we must put some limitations on the states and alphabetic symbols that can be used in the Turing machines that we consider. Clearly, given any Turing machine, we can change the names of
the states without changing the behavior of the machine. So, without any loss of generality, we can assume that all states have names chosen from the list: \( h, q, q', q'', q''', \ldots \). We assume that \( h \) is the halt state and \( q \) is the start state. Note that there is an infinite number of possible states, but any given Turing machine will only use finitely many states from this list.

As for the alphabets of the Turing machines, I want to look at Turing machines whose alphabets include the symbols 0, 1, \( a \), and of course \#. These are the symbols that the machines will use for input and output. The alphabets can also include other symbols. We will assume that these auxiliary symbols are chosen from the list: \( a', a'', a''', \ldots \). Given a Turing machine whose alphabet includes the symbols 0, 1, \( a \), and \#, we can rename any other symbols in its alphabet using names from this list. This renaming will not affect any of the behavior that we are interested in.

Now suppose we have one of these standard Turing machines—one whose states are chosen from the list \( h, q, q', q'', q''', \ldots \), whose start state is \( q \), and whose symbols are chosen from the list \#, 0, 1, \( a \), \( a' \), \( a'' \), \( a''' \), \ldots \). Such a machine can be completely encoded as a string of symbols over the alphabet \{\( h, q, L, R, \#, 0, 1, a, a' \}\}. A transition rule such as \( \delta(q'', 0) = (a''', L, q) \) can be encoded as a string \( q''0a''''Lq \). To encode a complete machine, simply encode each of its transition rules in this way and join them together in a string, separated by \$'s. We now have the symbolic representation for Turing machines that we need.

Note that a string over the alphabet \{\( h, q, L, R, \#, 0, 1, a, a' \}\} might or might not encode a Turing machine. However, it is a simple matter to check whether such a string is the code for a Turing machine. We can imagine the following process: Generate all the strings over the alphabet \{\( h, q, L, R, \#, 0, 1, a, a' \}\}. Check each string to see whether it encodes a Turing machine. If so, add the string to an output list. In this way, we can generate a list of all strings that encode standard Turing machines. In effect, the standard Turing machines, or at least their symbolic representations, form a recursively enumerable set. Let \( T_0 \) be the machine encoded by the first string in this list of standard Turing machines; let \( T_1 \) be the machine encoded by the second string; let \( T_2 \) be the machine encoded by the third string; and so on. The list \( T_0, T_1, T_2, \ldots \), includes every standard Turing machine. Furthermore, given \( n \in \mathbb{N} \), we can find the symbolic representation for \( T_n \) by generating strings in the list until we have \( n + 1 \) strings. Furthermore—and this is the essential point—we can use a Turing machine to do all these calculations. In fact, there is a Turing machine that, when run with input \( a^n \), will halt with the string representation of \( T_n \) written on its tape as output. The Turing machine that does this is \( G \),
5.3. THE LIMITS OF COMPUTATION

the first of the two machines that we need.

The second machine that we need will be called $U$. It is a so-called \textit{Universal Turing Machine}. The single Turing machine $U$ can simulate the computation of any standard Turing machine, $T$, on any input. Both the symbolic representation of $T$ and that of the input string are written to $U$’s tape, separated by a space. As $U$ simulates the computation of $T$, it will need some way to keep track of what state $T$ is in and of the position of $T$ on its (simulated) tape. It does this by writing the current state of $T$ on its tape, following $T$’s input string, and by adding a special symbol, such as $\@$, to the input string to mark $T$’s position. When $U$ is first started, it begins by adding the $\@$ to the beginning of the input string and writing a $q$ after the string to represent the start state of $T$. It is then relatively straightforward for $U$ to simulate the computation of $T$. For each step in the computation of $T$, it can determine the current state of $T$ (which is recorded on $U$’s tape) and the symbol which $T$ is currently reading (which is on $U$’s tape, after the $\@$). $U$ searches the symbolic representation of $T$ for the rule that tells $T$ what to do in this situation. Using this rule, $U$ can update its representation of $T$’s state, position, and tape to reflect the result of applying the rule. If the new state of $T$ is the halt state, then $U$ also halts. Otherwise, it goes on to simulate the next step in $T$’s computation. Note that when $U$ is given $T$ and an input string $w$ as input, $U$ will halt if and only if $T$ halts on input $w$. (Obviously, this is a very inefficient simulation, but we are not concerned with efficiency here.)

So, we have our two machines, $G$ and $U$. After all this setup, we are finally in a position to look at the major theorem that we have been working towards.

\textbf{Theorem 5.4.} Let $T_0$, $T_1$, $T_2$, \ldots, be the standard Turing machines, as described above. Let $K$ be the language over the alphabet \{a\} defined by

$$K = \{a^n \mid T_n \text{ halts when run with input } a^n\}.$$  

Then $K$ is a recursively enumerable language, but $K$ is not recursive. The complement

$$\overline{K} = \{a^n \mid T_n \text{ does not halt when run with input } a^n\}.$$  

is a language that is not recursively enumerable.

First note that if both $K$ and $\overline{K}$ were recursively enumerable, then $K$ would be recursive, by Theorem 5.3. So, once we show that $K$ is recursively enumerable but not recursive, it follows immediately that $\overline{K}$ cannot be
recursively enumerable. That is, the second part of the theorem follows from the first.

To show that \( K \) is recursively enumerable, it suffices to find a Turing machine, \( M \), that accepts \( K \). That is, when run with input \( a^n \), for \( n \in \mathbb{N} \), \( M \) should halt if and only if \( a^n \in K \). We can build \( M \) from the Turing machines \( G \) and \( U \) which were introduced above. When started with input \( a^n \), \( M \) should proceed as follows: First copy the input. Run \( G \) on the first copy of \( a^n \). This will produce a symbolic description of the Turing machine \( T_n \). Now run \( U \) to simulate the computation of \( T_n \) on input \( a^n \). This simulation will end if and only if \( T_n \) halts when run with input \( a^n \), that is, if and only if \( a^n \in K \). The Turing machine \( M \) that performs the computation we have described accepts the language \( K \). This proves that \( K \) is recursively enumerable.

To show that \( K \) is not recursive, we need to show that there is no Turing machine that decides \( K \). Let \( H \) be any Turing machine. We must show that no matter what \( H \) does, it does not decide the language \( K \). We must do this without knowing anything more about \( H \) that the fact that it is a Turing machine. To say that \( H \) decides \( K \) would mean that for any \( n \in \mathbb{N} \), when \( H \) is run with input \( a^n \), \( H \) will halt with output 1 if \( a^n \in K \) and will halt with output 0 if \( a^n \notin K \). To show that \( H \) does not decide \( K \) we need to show that there is some \( n \in \mathbb{N} \) such that when \( H \) is run with input \( a^n \), \( H \) either fails to halt or else halts but gives the wrong output. Note in particular that we only need to find one \( n \) for which \( H \) does not give the correct result. As we try to find \( n \), we have nothing much to work with but \( H \) itself.

To find \( n \), we construct a Turing machine \( M \) that is a simple variation on \( H \). When \( M \) is run on any input, it duplicates the behavior of \( H \) on that input until \( H \) halts (if it ever does). At that point, \( M \) should check \( H \)'s output. If \( H \) has halted with output 1, then \( M \) should go into an infinite loop, so that \( M \) never halts in this case. Otherwise, if the output of \( H \) is not 1, then \( M \) should halt. Now, we can assume that \( M \) is one of the standard Turing machines, say \( M = T_n \). (If \( M \) is not already one of these machines, it is because it uses different names for its states and symbols. Renaming the states and symbols will produce an equivalent machine with the same behavior as \( M \), and we can replace \( M \) with this standard machine.)

We now have a Turing machine \( T_n = M \) which has the following behavior when it is run with input \( a^n \) (note that the \( n \) here is the same \( n \) as in \( T_n \)): If \( H \) halts with output 1 on input \( a^n \), then \( T_n \) will fail to halt on input \( a^n \). If \( H \) halts with output 0 on input \( a^n \), then \( T_n \) fails to halt on input \( a^n \). (What \( T_n \) might do in other cases is not relevant here.)

Remember that we are trying to show that \( H \) does not decide the language \( K \). I claim that, in fact, \( H \) does not give the correct answer for
When \( H \) is run with input \( a^n \), it is supposed to halt with output 1 if \( a^n \in K \), and it is supposed to halt with output 0 if \( a^n \not\in K \). Recall that \( a^n \in K \) if and only if \( T_n \) halts when run with input \( a^n \).

Suppose that we run \( H \) with input \( a^n \). If \( H \) does not halt with output 0 or 1, then it has certainly not given the correct answer for \( a^n \). Now, suppose that \( H \) halts with output 1 on input \( a^n \). In this case, by the properties of \( T_n \) given above, we know that \( T_n \) does not halt on input \( a^n \). But that means, by definition of \( K \), that \( a^n \notin K \). By halting with output 1 in this case, \( H \) has given the wrong answer for \( a^n \). Finally, suppose that \( H \) halts with output 0 on input \( a^n \). We then know that \( T_n \) halts on input \( a^n \). But that means that \( a^n \in K \). Again, by halting with output 0 in this case, \( H \) has given the wrong answer for \( a^n \). So, in no case will \( H \) give the correct answer for \( a^n \). This means that \( H \) does not decide the language \( K \), because \( H \) gives an incorrect answer when it is run with the particular input \( a^n \). \( H \) does not decide \( K \), and since \( H \) was an arbitrary Turing machine, we see that there is no Turing machine at all that decides the language \( K \). Thus, \( K \) is not a recursive language, as the theorem claims.

To decide the language \( K \) would be to solve the following problem: Given a Turing machine \( T_n \), decide whether or not \( T_n \) will halt when it is run with input \( a^n \). This problem is called the **Halting Problem**. We have shown that there is no Turing machine that solves this problem. Given the equivalence of Turing machines and computer programs, we can also say that there is no computer program that solves the halting problem. We say that the halting problem is **computationally unsolvable**.

The halting problem is just one of many problems that cannot be solved by Turing machines or computer programs. In fact, almost any interesting yes/no question that can be asked about Turing machines or programs is in this class: Does this Turing machine halt for all possible inputs in \( \Sigma^* \)? Given this input, will this program ever halt? Do these two programs (or Turing machines) have the same output for each possible input? Will this Turing machine ever halt if it is started on a blank tape? All these problems are computationally unsolvable in the sense that there is no Turing machine or computer program that will answer them correctly in all cases. The existence of such problems is a real limitation on the power of computation.