Lecture 15

# Encodings; examples of decidable languages

Now that we have studied basic aspects of the Turing machine model, including variants of Turing machines and the computationally equivalent stack machine model, it is time to discuss some examples of decidable languages. In this lecture we will focus on examples based on finite automata and context-free grammars. These languages will have a somewhat different character from most of the languages we discussed previously in the course; their definitions are centered on fundamental mathematical concepts, as opposed to simple syntactic patters.

Before doing this, however, we will discuss *encodings*, which allow us to represent complicated mathematical objects using strings over a given alphabet. For example, we may wish to consider a DTM that takes as input a number, a graph, a DFA, a CFG, another DTM (maybe even a description of itself), or a list of objects of multiple types. We will make use of the notion of an encoding for the remainder of the course.

## 15.1 Encodings of interesting mathematical objects

The idea that we can encode different sorts of objects as strings will be familiar to students of computer science, and for this reason we will not belabor this issue—but it will nevertheless be helpful to establish a few conventions and introduce useful ideas concerning the encoding of different objects of interest.

### Encoding multiple strings into one

Let us begin by considering the following task that concerns two hypothetical individuals: Alice and Bob. Alice has two binary strings $x \in \{0,1\}^*$ and $y \in \{0,1\}^*$,

and she would like to communicate these two strings to Bob. However, for some hypothetical reason, Alice is only allowed to transmit a single binary string to Bob, so somehow $x$ and $y$ must be packed into a single string $z \in \{0,1\}^*$ from which Bob can recover both $x$ and $y$. The two of them may agree ahead of time on a method through which this will be done, but naturally the method must be agreed upon prior to Alice knowing which strings $x$ and $y$ are to be communicated. That is, the method must work for an arbitrary choice of binary strings $x$ and $y$. There are different methods through which this task may be accomplished, but let us describe just one method.

The first step is to introduce a new symbol, which we will call #. We then choose to encode the pair $(x, y)$ into a single string $x\#y \in \{0,1,\#\}^*$. Obviously, if Alice were to send this string to Bob, he could recover $x$ and $y$ without difficulty, so it is a good method in that sense—but unfortunately it does not solve the original problem because it makes use of the alphabet $\{0,1,\#\}$ rather than $\{0,1\}$.

The second step of the method will take us back to the binary alphabet: we can encode the string $x\#y$ as a binary string by substituting the individual symbols according to this pattern:[1]

$$\begin{aligned} 0 &\to 00 \\ 1 &\to 01 \\ \# &\to 1. \end{aligned} \tag{15.1}$$

The resulting binary string will be the *encoding* of the two strings $x$ and $y$ that Alice sends to Bob.

For example, if the two strings are $x = 0110$ and $y = 01111$, we first consider the string 0110#01111, and then perform the substitution suggested above to obtain

$$\langle 0110, 01111 \rangle = 0001010010001010101. \tag{15.2}$$

Here we have used a notation that we will use frequently throughout much of the remainder of the course: whenever we have some object $X$, along with an encoding scheme that encodes a class of objects that includes $X$ as strings, we write $\langle X \rangle$ to denote the string that encodes $X$. In the equation above, the notation $\langle 0110, 01111 \rangle$ therefore refers to the encoding of the two strings 0110 and 01111, viewed as an ordered pair.

Let us make a few observations about the encoding scheme just described:

1. It is easy to recover the strings $x$ and $y$ from the encoding $\langle x, y \rangle$. Specifically, so long as we find the symbol 0 in each odd-numbered position, the symbols

[1] There are other patterns that would work equally well. The one we have selected is an example of a *prefix-free code*; because none of the strings appearing on the right-hand side of (15.1) is a prefix of any of the other strings, we are guaranteed that by concatenating together a sequence of these strings we can recover the original string without ambiguity.

in the even-numbered positions that follow belong to $x$; and once we find a 1 in an odd-numbered position, we know that $x$ is determined and it is time to recover $y$ through a similar process.

2. The scheme works not only for two strings $x$ and $y$, but for any finite number of binary strings $x_1, \ldots, x_n$; such a list of strings may be encoded by first forming the string $x_1 \# x_2 \# \cdots \# x_n \in \{0, 1, \#\}^*$, and then performing the substitutions described above to obtain $\langle x_1, \ldots, x_n \rangle \in \{0, 1\}^*$.

3. Every $n$-tuple $(x_1, \ldots, x_n)$ of binary strings has a unique encoding $\langle x_1, \ldots, x_n \rangle$, but it is not the case that every binary string encodes an $n$-tuple of binary strings. In other words, the encoding is one-to-one but not onto. For instance, the string 10 does not decode to any string over the alphabet $\{0, 1, \#\}$, and therefore does not encode an $n$-tuple of binary strings. This is not a problem; most of the encoding schemes we will consider in this course have the same property that not every string is a valid encoding of some object of interest.

4. You could easily generalize this scheme to larger alphabets by adding a new symbol to mark the division between strings over the original alphabet, and then choosing a suitable encoding in place of (15.1).

This one method of encoding multiple strings into one turns out to be incredibly useful, and by using it repeatedly we can devise encoding schemes for highly complex mathematical objects.

## Encoding strings over arbitrary alphabets using a fixed alphabet

Next, let us consider the task of encoding a string over an alphabet $\Gamma$ whose size we do not know ahead of time by a string over a fixed alphabet $\Sigma$. In the interest of simplicity, let us take $\Sigma = \{0, 1\}$ to be the binary alphabet.

Before we discuss a particular scheme through which this task can be performed, let us take a moment to clarify the task at hand. In particular, it should be made clear that we are not looking for a way to encode strings over any possible alphabet $\Gamma$ that you could ever imagine. For instance, consider the alphabet

$$\Gamma = \{\text{}, \text{}, \text{}, \text{}\}, \tag{15.3}$$

from the very first lecture of the course. Some might consider this to be an interesting alphabet, but in some sense there is nothing special about it—all that is really relevant from the viewpoint of the theory of computation is that it has four sym-

bols, so there is little point in differentiating it from the alphabet $\Gamma = \{0, 1, 2, 3\}$.[2] That is, when we think about models of computation, all that really matters is the number of symbols in our alphabet, and sometimes the order we choose to put them in, but not the size, shape, or color of the symbols.

With this understanding in place, we will make the assumption that our encoding task is to be performed for an alphabet of the form

$$\Gamma = \{0, 1, \ldots, n-1\} \tag{15.4}$$

for some positive integer $n$, where we are imagining that each integer between 0 and $n-1$ is a single symbol.

The method from the previous subsection provides a simple means through which the task at hand can be accomplished. First, for every nonnegative integer $k \in \mathbb{N}$, let us decide that the encoding $\langle k \rangle$ of this number is given by its representation using binary notation:

$$\langle 0 \rangle = 0, \quad \langle 1 \rangle = 1, \quad \langle 2 \rangle = 10, \quad \langle 3 \rangle = 11, \quad \langle 4 \rangle = 100, \quad \text{etc.} \tag{15.5}$$

Then, to encode a given string $k_1 k_2 \cdots k_m$, we simply encode the $m$ binary strings $\langle k_1 \rangle, \langle k_2 \rangle, \ldots, \langle k_m \rangle$ into a single binary string

$$\big\langle \langle k_1 \rangle, \langle k_2 \rangle, \ldots, \langle k_m \rangle \big\rangle \tag{15.6}$$

using the method from the previous subsection.

For example, let us consider the string 001217429, which we might assume is over the alphabet $\{0, \ldots, 9\}$ (although this assumption will not influence the encoding that is obtained). The method from the previous subsection suggests that we first form the string

$$0\#0\#1\#10\#1\#111\#100\#10\#1001 \tag{15.7}$$

and then encode this string using the substitutions (15.1). The binary string we obtain is

$$\langle 001217429 \rangle = 00100101101001011010101101000010100101000001. \tag{15.8}$$

Finally, let us briefly discuss the possibility that the alphabet $\Sigma$ is the unary alphabet $\Sigma = \{0\}$ rather than the binary alphabet. You can still encode strings over any alphabet $\Gamma = \{0, \ldots, n-1\}$ using this alphabet, although (not surprisingly) it

[2] You could of course consider encoding schemes that represent the shapes and sizes of different alphabet symbols—the symbols appearing in (15.3), for example, are in fact the result of a binary string encoding obtained from an image compression algorithm—but this is not what we are talking about.

will be extremely inefficient. One way to do this is to first encode strings over $\Gamma$ as strings over the binary alphabet, exactly as discussed above, and then to encode binary strings as unary strings with respect to the lexicographic ordering:

$$\begin{aligned} \varepsilon &\to \varepsilon \\ 0 &\to 0, \\ 1 &\to 00, \\ 10 &\to 000, \\ 11 &\to 0000, \\ 100 &\to 00000, \end{aligned} \tag{15.9}$$

and so on.

This means that you could, in principle, encode an entire book in unary. Think of an ordinary book as a string over the alphabet that includes upper- and lowercase letters, spaces, and punctuation marks, and imagine encoding this string over the unary alphabet as just described. You open the unary-encoded book and see that every page is filled with 0s, and as you are reading the book you have absolutely no idea what it is about. All you can do is to eliminate the possibility that the book corresponds to a shorter string of 0s than the number you have seen so far, just like when you rule out the possibility that it is 3 o'clock when the bells at City Hall have (thus far) rung four times. Finally you finish the book and in an instant it all becomes clear, and you say "Wow, what a great book!"

## Numbers, vectors, and matrices

We already used the elementary fact that nonnegative integers can be encoded as binary strings using binary notation in the previous subsection. One can also encode arbitrary integers using binary notation by interpreting the first bit of the encoding to be a sign bit. Rational numbers can be encoded as pairs of integers (representing the numerator and denominator), by first expressing the individual integers in binary, and then encoding the two strings into one using the method from earlier in the lecture. One could also consider floating point representations, which are of course very common in practice, but also have the disadvantage that they only represent rational numbers for which the denominator is a power of two.

With a method for encoding numbers as binary strings in mind, one can represent vectors by simply encoding the entries as strings, and then encoding these multiple strings into a single string using the method described at the start of the lecture. Matrices can be represented as lists of vectors. Indeed, once you know how to encode lists of strings as strings, you can very easily devise encoding schemes for highly complex mathematical objects.

## An encoding scheme for DFAs and NFAs

Now let us devise an encoding scheme for DFAs and NFAs. We will start with DFAs, and once we are finished we will observe how the scheme can be easily modified to obtain an encoding scheme for NFAs.

What we are aiming for is a way to encode every possible DFA

$$M = (Q, \Gamma, \delta, q_0, F) \tag{15.10}$$

as a binary string $\langle M \rangle$.[3] Intuitively speaking, given the binary string $\langle M \rangle \in \Sigma^*$, it should be possible to recover a description of exactly how $M$ operates without difficulty. There are, of course, many possible encoding schemes that one could devise—we are just choosing one that works but otherwise is not particularly special.

Along similar lines to the discussion above concerning the encoding of strings over arbitrary alphabets, we will make the assumption that the alphabet $\Gamma$ of $M$ takes the form

$$\Gamma = \{0, \ldots, n-1\} \tag{15.11}$$

for some positive integer $n$. For the same reasons, we will assume that the state set of $M$ takes the form

$$Q = \{q_0, \ldots, q_{m-1}\} \tag{15.12}$$

for some positive integer $m$.

There will be three parts of the encoding:

1. A positive integer $n$ representing $|\Gamma|$. This number will be represented using binary notation.
2. A specification of the set $F$ together with the number of states $m$. These two things together can be described by a binary string $s$ of length $m$. Specifically, the string
$$s = b_0 b_1 \cdots b_{m-1} \tag{15.13}$$
specifies that
$$F = \{q_k \,:\, k \in \{0, \ldots, m-1\},\ b_k = 1\}, \tag{15.14}$$
and of course the number of states $m$ is given by the length of the string $s$. Hereafter we will write $\langle F \rangle$ to refer to the encoding of the subset $F$ that is obtained in this way.

[3] Notice that we are taking the alphabet of $M$ to be $\Gamma$ rather than $\Sigma$ to be consistent with the conventions used in the previous subsections: $\Gamma$ is an alphabet having an arbitrary size, and we cannot assume it is fixed as we devise our encoding scheme, while $\Sigma = \{0, 1\}$ is the alphabet we are using for the encoding.

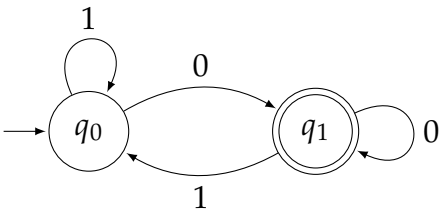


Figure 15.1: A simple example of a DFA.

3. The transition function $\delta$ will be described by listing all of the inputs and outputs of this function, in the following way. First, for $j, k \in \{0, \ldots, m-1\}$ and $a \in \Gamma = \{0, \ldots, n-1\}$, the string

$$\big\langle \langle j \rangle, \langle a \rangle, \langle k \rangle \big\rangle \tag{15.15}$$

specifies that

$$\delta(q_j, a) = q_k. \tag{15.16}$$

Here, $\langle j \rangle$, $\langle k \rangle$, and $\langle a \rangle$ refer to the strings obtained from binary notation, which makes sense because $j$, $k$, and $a$ are all nonnegative integers. We then encode the list of all of these strings, in the natural ordering that comes from iterating over all pairs $(j, a)$, into a single string $\langle \delta \rangle$.

For example, the DFA depicted in Figure 15.1 has a transition function $\delta$ whose encoding is

$$\langle \delta \rangle = \big\langle \langle 0,0,1 \rangle, \langle 0,1,0 \rangle, \langle 1,0,1 \rangle, \langle 1,1,0 \rangle \big\rangle. \tag{15.17}$$

(We will leave it in this form rather than expanding it out as a binary string in the interest of clarity.)

Finally, the encoding of a given DFA $M$ is just a list of the three parts just described:

$$\langle M \rangle = \big\langle \langle n \rangle, \langle F \rangle, \langle \delta \rangle \big\rangle. \tag{15.18}$$

This encoding scheme can easily be modified to obtain an encoding scheme for NFAs. This time, the values the transition function takes are subsets of $Q$ rather than elements of $Q$, and we must also account for the possibility of $\varepsilon$-transitions. Fortunately, we already know how to encode subsets of $Q$; we did this for the set $F$, and exactly the same method can be used to encode any one of the subsets $\delta(q_j, a)$ as a binary string $\langle \delta(q_j, a) \rangle$ having length equal to the total number of states in $Q$. That is, the string

$$\big\langle \langle j \rangle, \langle a \rangle, \langle \delta(q_j, a) \rangle \big\rangle \tag{15.19}$$

describes the value of the transition function for the pair $(q_j, a)$. To specify the $\varepsilon$-transitions of $M$, we may use the string

$$\big\langle \langle j \rangle, \varepsilon, \langle \delta(q_j, \varepsilon) \rangle \big\rangle, \tag{15.20}$$

which takes advantage of the fact that we never have $\langle a \rangle = \varepsilon$ for any symbol $a \in \Gamma$. As before, we simply list all of the strings corresponding to the different inputs of $\delta$ in order to encode $\delta$.

### Encoding schemes for regular expressions, CFGs, PDAs, etc.

We could continue on and devise encoding schemes through which regular expressions, CFGs, PDAs, DTMs, and DSMs can be specified. Because of its importance, we will in fact return to the case of DTMs in the next lecture, but for the others I will leave it to you to think about how you might design encoding schemes. There are countless specific ways to do this, but it turns out that the specifics are not really all that important—the reason why we did this carefully DFAs and NFAs is to illustrate how it can be done for those models, with the principal aim being to clarify the concept rather than to create an encoding scheme whose specific aspects are conceptually relevant.

## 15.2 Decidability of formal language problems

Now let us turn our attention toward languages that concern the models of computation we have studied previously in the course.

### Languages based on DFAs, NFAs, and regular expressions

The first language we will consider is this one:

$$\mathrm{A}_{\mathrm{DFA}} = \big\{ \langle \langle D \rangle, \langle w \rangle \rangle \,:\, D \text{ is a DFA and } w \in \mathrm{L}(D) \big\}. \tag{15.21}$$

Here we assume that $\langle D \rangle$ is the encoding of a given DFA $D$, $\langle w \rangle$ is the encoding of a given string $w$, and $\langle \langle D \rangle, \langle w \rangle \rangle$ is the encoding of the two strings $\langle D \rangle$ and $\langle w \rangle$, all as described earlier in the lecture. Thus, $\langle D \rangle$, $\langle w \rangle$, and $\langle \langle D \rangle, \langle w \rangle \rangle$ are all binary strings. It could be the case, however, that the alphabet of $D$ is any alphabet of the form $\Gamma = \{0, \ldots, n-1\}$, and likewise for the string $w$.

It is a natural question to ask whether or not the language $\mathrm{A}_{\mathrm{DFA}}$ is decidable. It certainly is. For a given input string $x \in \{0,1\}^*$, one can easily check that it takes the form $x = \langle \langle D \rangle, \langle w \rangle \rangle$ for a DFA $D$ and a string $w$, and then check whether or not $D$ accepts $w$ by simply *simulating*, just like you would do with a piece of paper

The DTM $M$ operates as follows on input $x \in \{0,1\}^*$:

1. If it is not the case that $x = \langle\langle D \rangle, \langle w \rangle\rangle$ for $D$ being a DFA and $w$ being a string over the alphabet of $D$, then reject.
2. Simulate $D$ on input $w$; accept if $D$ accepts $w$ and reject if $D$ rejects $w$.

Figure 15.2: A high-level description of a DTM $M$ that decides the language $\mathrm{A}_{\mathrm{DFA}}$.

and a pencil if you were asked to make this determination for yourself. Figure 15.2 gives a high-level description of a DTM $M$ that decides the language $\mathrm{A}_{\mathrm{DFA}}$ along these lines.

Now, you might object to the claim that Figure 15.2 describes a DTM that decides $\mathrm{A}_{\mathrm{DFA}}$. It does describe the main idea of how $M$ operates, which is that it simulates $D$ on input $w$, but it offers hardly any detail at all. It seems more like a suggestion for how to design a DTM than an actual description of a DTM.

This is a fair criticism, but as we move forward with the course, we will need to make a transition along these lines. The computations we will consider will become more and more complicated, and in the interest of both time and clarity we must abandon the practice of describing the DTMs that perform these computations explicitly. Hopefully our discussions and development of the DTM model have convinced you that the process of taking a high-level description of a DTM, such as the one in Figure 15.2, and producing an actual DTM that performs the computation described would be a more or less routine task.

Because the description of the DTM $M$ suggested by Figure 15.2 is our first example of such a high-level DTM description, let us take a moment to consider in greater detail how it could be turned into a formal specification of a DTM. Because we know that any deterministic stack machine can be simulated by a DTM, it suffices to describe a DSM $M$ that operates as described in Figure 15.2.

1. The input $x$ to the DSM $M$ is initially stored in stack number 1, which we might instead choose to name X for clarity. The first step in Figure 15.2 is to check that the input takes the form $x = \langle\langle D \rangle, \langle w \rangle\rangle$. Assuming that the input does take this form, it is convenient for the sake of the second step of $M$ (meaning the simulation of $D$ on input $w$) that the input is split into two parts, with the string $\langle D \rangle$ being stored in a stack called D and $\langle w \rangle$ being stored in a stack called W. This splitting could easily be done as a part of the check that the input does take the form $x = \langle\langle D \rangle, \langle w \rangle\rangle$.

2. To simulate $D$ on input $w$, the DSM $M$ will need to keep track of the current state of $D$, so it is natural to introduce a new stack Q for this purpose. At the start of the simulation, Q is initialized it so that it stores 0 (the encoding of the state $q_0$).
3. The actual simulation proceeds in the natural way, which is to examine the encodings of the symbols of $w$ stored in W, one at a time, updating the state contained in Q accordingly. While an explicit description of the DSM states and transitions needed to do this would probably look rather complex, it could be done in a conceptually simple manner. In particular, each step of the simulation would presumably involve $M$ searching through the transitions of $D$ stored in D to find a match with the current state encoding stored in Q and the next input symbol encoding stored in W, after which Q is updated. Naturally, $M$ can make use of additional stacks and make copies of strings as needed so that the encoding $\langle D \rangle$ is always available at the start of each simulation step.
4. Once the simulation is complete, an examination of the state stored in Q and the encoding $\langle F \rangle$ of the accepting states of $D$ leads to acceptance or rejection appropriately.

All in all, it would be a tedious task to write down the description of a DSM $M$ that behaves in the manner just described—but I hope you will agree that with a bit of time, patience, and planning, it would be feasible to do this. An explicit description of such a DSM $M$ would surely be made more clear if a thoughtful use of subroutines was devised (not unlike the analogous task of writing a computer program to perform such a simulation). Once the specification of this DSM is complete, it can then be simulated by a DTM as described in the previous lecture.

Next let us consider a variant of the language $\mathrm{A}_{\mathrm{DFA}}$ for NFAs in place of DFAs:

$$\mathrm{A}_{\mathrm{NFA}} = \bigl\{ \langle \langle N \rangle, \langle w \rangle \rangle \,:\, N \text{ is an NFA and } w \in \mathrm{L}(N) \bigr\}. \tag{15.22}$$

Again, it is our assumption that the encodings with respect to which this language is defined are as discussed earlier in the lecture. The language $\mathrm{A}_{\mathrm{NFA}}$ is also decidable. This time, however, it would not be reasonable to simply describe a DSM that "simulates $N$ on input $w$," because it is not at all clear how a deterministic computation can simulate a nondeterministic finite automaton computation.

What we can do instead is to make use of the process through which NFAs are converted to DFAs that we discussed in Lecture 3; this is a well-defined deterministic procedure, and it can certainly be performed by a DTM. Figure 15.3 gives a high-level description of a DTM $M$ that decides $\mathrm{A}_{\mathrm{NFA}}$. Once again, although it would be a time-consuming task to explicitly describe a DTM that performs this

The DTM $M$ operates as follows on input $x \in \{0,1\}^*$:

1. If it is not the case that $x = \langle\langle N \rangle, \langle w \rangle\rangle$ for $N$ being an NFA and $w$ being a string over the alphabet of $N$, then reject.
2. Convert $N$ into an equivalent DFA $D$ using the subset construction described in Lecture 3.
3. Simulate $D$ on input $w$; accept if $D$ accepts $w$ and reject if $D$ rejects $w$.

Figure 15.3: A high-level description of a DTM $M$ that decides the language $\mathrm{A}_{\mathrm{NFA}}$.

computation, it is reasonable to view this as a straightforward task in a conceptual sense.

One can also define a language similar to $\mathrm{A}_{\mathrm{DFA}}$ and $\mathrm{A}_{\mathrm{NFA}}$, but for regular expressions in place of DFAs and NFAs:

$$\mathrm{A}_{\mathrm{REX}} = \bigl\{ \langle\langle R \rangle, \langle w \rangle\rangle \;:\; R \text{ is a regular expression and } w \in \mathrm{L}(R) \bigr\}. \tag{15.23}$$

We did not actually discuss an encoding scheme for regular expressions, so it will be left to you to devise or imagine your own encoding scheme—but as long as you picked a reasonable one, the language $\mathrm{A}_{\mathrm{REX}}$ would be decidable. In particular, given a reasonable encoding scheme for regular expressions, a DTM could first convert this regular expression into an equivalent DFA, and then simulate this DFA on the string $w$.

Here is a different example of a language, which we will argue is also decidable:

$$\mathrm{E}_{\mathrm{DFA}} = \bigl\{ \langle D \rangle \;:\; D \text{ is a DFA and } \mathrm{L}(D) = \varnothing \bigr\}. \tag{15.24}$$

In this case, one cannot decide this language simply by "simulating the DFA $D$," because a priori there is no particular string on which to simulate it; we care about every possible string that could be given as input to $D$ and whether or not $D$ accepts any of them. Deciding the language $\mathrm{E}_{\mathrm{DFA}}$ is therefore not necessarily a straightforward simulation task.

What we can do instead is to treat the decision problem associated with this language as a *graph reachability* problem. The DTM $M$ suggested by Figure 15.4 takes this approach and decides $\mathrm{E}_{\mathrm{DFA}}$. By combining this DTM with ideas from the previous examples, one can prove that analogously defined languages $\mathrm{E}_{\mathrm{NFA}}$ and $\mathrm{E}_{\mathrm{REX}}$ are also decidable:

$$\begin{aligned} \mathrm{E}_{\mathrm{NFA}} &= \bigl\{ \langle N \rangle \;:\; N \text{ is an NFA and } \mathrm{L}(N) = \varnothing \bigr\}, \\ \mathrm{E}_{\mathrm{REX}} &= \bigl\{ \langle R \rangle \;:\; R \text{ is a regular expression and } \mathrm{L}(R) = \varnothing \bigr\}. \end{aligned} \tag{15.25}$$

The DTM $M$ operates as follows on input $x \in \{0,1\}^*$:

1. If it is not the case that $x = \langle D \rangle$ for $D$ being a DFA, then reject.
2. Set $S \leftarrow \{0\}$.
3. Set $a \leftarrow 1$.
4. For every pair of integers $j, k \in \{0, \ldots, m-1\}$, where $m$ is the number of states of $D$, do the following:
   4.1 If $j \in S$ and $k \notin S$, and $D$ includes a transition from $q_j$ to $q_k$, then set $S \leftarrow S \cup \{k\}$ and $a \leftarrow 0$.
5. If $a = 0$ then goto step 3.
6. *Reject* if there exists $k \in S$ such that $q_k \in F$ (i.e., $q_k$ is an accept state of $D$), otherwise *accept*.

Figure 15.4: A high-level description of a DTM $M$ that decides the language $\mathrm{E}_{\mathrm{DFA}}$.

One more example of a decidable language concerning DFAs is this language:

$$\mathrm{EQ}_{\mathrm{DFA}} = \big\{ \langle \langle A \rangle, \langle B \rangle \rangle \;:\; A \text{ and } B \text{ are DFAs and } \mathrm{L}(A) = \mathrm{L}(B) \big\}. \tag{15.26}$$

Figure 15.5 gives a high-level description of a DTM that decides this language. One natural way to perform the construction in step 2 is to use the Cartesian product construction described in Lecture 4.

## Languages based on CFGs

Next let us turn to a couple of examples of decidable languages concerning context-free grammars. Following along the same lines as the examples discussed above, we may consider these languages:

$$\begin{aligned} \mathrm{A}_{\mathrm{CFG}} &= \big\{ \langle \langle G \rangle, \langle w \rangle \rangle \;:\; G \text{ is a CFG and } w \in \mathrm{L}(G) \big\}, \\ \mathrm{E}_{\mathrm{CFG}} &= \big\{ \langle G \rangle \;:\; G \text{ is a CFG and } \mathrm{L}(G) = \varnothing \big\}. \end{aligned} \tag{15.27}$$

Once again, although we have not explicitly described an encoding scheme for context-free grammars, it is not difficult to come up with such a scheme (or to just imagine that such a scheme has been defined). A DTM that decides the first language is described in Figure 15.6. It is worth noting that this is a ridiculously inefficient way to decide the language $\mathrm{A}_{\mathrm{CFG}}$, but right now we do not care! We are

The DTM $M$ operates as follows on input $x \in \{0,1\}^*$:

1. If it is not the case that $x = \langle\langle A\rangle, \langle B\rangle\rangle$ for $A$ and $B$ being DFAs, then reject.
2. Construct a DFA $C$ for which $\mathrm{L}(C) = \mathrm{L}(A) \triangle \mathrm{L}(B)$.
3. Accept if $\langle C\rangle \in \mathrm{E}_{\mathrm{DFA}}$ and otherwise reject.

Figure 15.5: A high-level description of a DTM $M$ that decides the language $\mathrm{EQ}_{\mathrm{DFA}}$.

The DTM $M$ operates as follows on input $x \in \{0,1\}^*$:

1. If it is not the case that $x = \langle\langle G\rangle, \langle w\rangle\rangle$ for $G$ a CFG and $w$ a string, then reject.
2. Convert $G$ into an equivalent CFG $H$ in Chomsky normal form.
3. If $w = \varepsilon$ then *accept* if $S \to \varepsilon$ is a rule in $H$ and *reject* otherwise.
4. Search over all possible derivations by $H$ having $2|w| - 1$ steps (of which there are finitely many). *Accept* if a valid derivation of $w$ is found, and *reject* otherwise.

Figure 15.6: A high-level description of a DTM $M$ that decides the language $\mathrm{A}_{\mathrm{CFG}}$. This DTM is ridiculously inefficient, but there are more efficient ways to decide this language.

just trying to prove that this language is decidable. There are, in fact, much more efficient ways to decide this language, but we will not discuss them now.

Finally, the language $\mathrm{E}_{\mathrm{CFG}}$ can be decided using a variation on the reachability technique. In essence, we keep track of a set containing variables that generate at least one string, and then test to see if the start variable is contained in this set. A DTM that decides this language is described in Figure 15.7.

Now, you may be wondering about this next language, as it is analogous to one concerning DFAs from above:

$$\mathrm{EQ}_{\mathrm{CFG}} = \big\{\langle\langle G\rangle, \langle H\rangle\rangle \,:\, G \text{ and } H \text{ are CFGs and } \mathrm{L}(G) = \mathrm{L}(H)\big\}. \tag{15.28}$$

As it turns out, this language is *undecidable*. We will not go through the proof because it would take us a bit too far off the path of the rest of the course—but some

The DTM $M$ operates as follows on input $x \in \{0,1\}^*$:

1. If it is not the case that $x = \langle G \rangle$ for $G$ a CFG, then *reject*.
2. Set $T \leftarrow \Sigma$ (for $\Sigma$ being the alphabet of $G$).
3. Set $a \leftarrow 1$.
4. For each rule $X \to w$ of $G$ do the following:
   4.1 If $X$ is not contained in $T$, and every variable and every symbol of $w$ is contained in $T$, then set $T \leftarrow T \cup \{X\}$ and $a \leftarrow 0$.
5. If $a = 0$ then goto step 3.
6. *Reject* if the start variable of $G$ is contained in $T$, otherwise *accept*.

Figure 15.7: A high-level description of a DTM $M$ that decides the language $\mathrm{E}_{\mathrm{CFG}}$.

of the facts we will prove in the next lecture may shed some light on why this language is undecidable. Some other examples of undecidable languages concerning context-free grammars are these:

$$\begin{aligned}
&\big\{\langle G \rangle \,:\, G \text{ is a CFG that generates all strings over its alphabet}\big\},\\
&\big\{\langle G \rangle \,:\, G \text{ is an ambiguous CFG}\big\},\\
&\big\{\langle G \rangle \,:\, G \text{ is a CFG and } \mathrm{L}(G) \text{ is inherently ambiguous}\big\}.
\end{aligned} \tag{15.29}$$