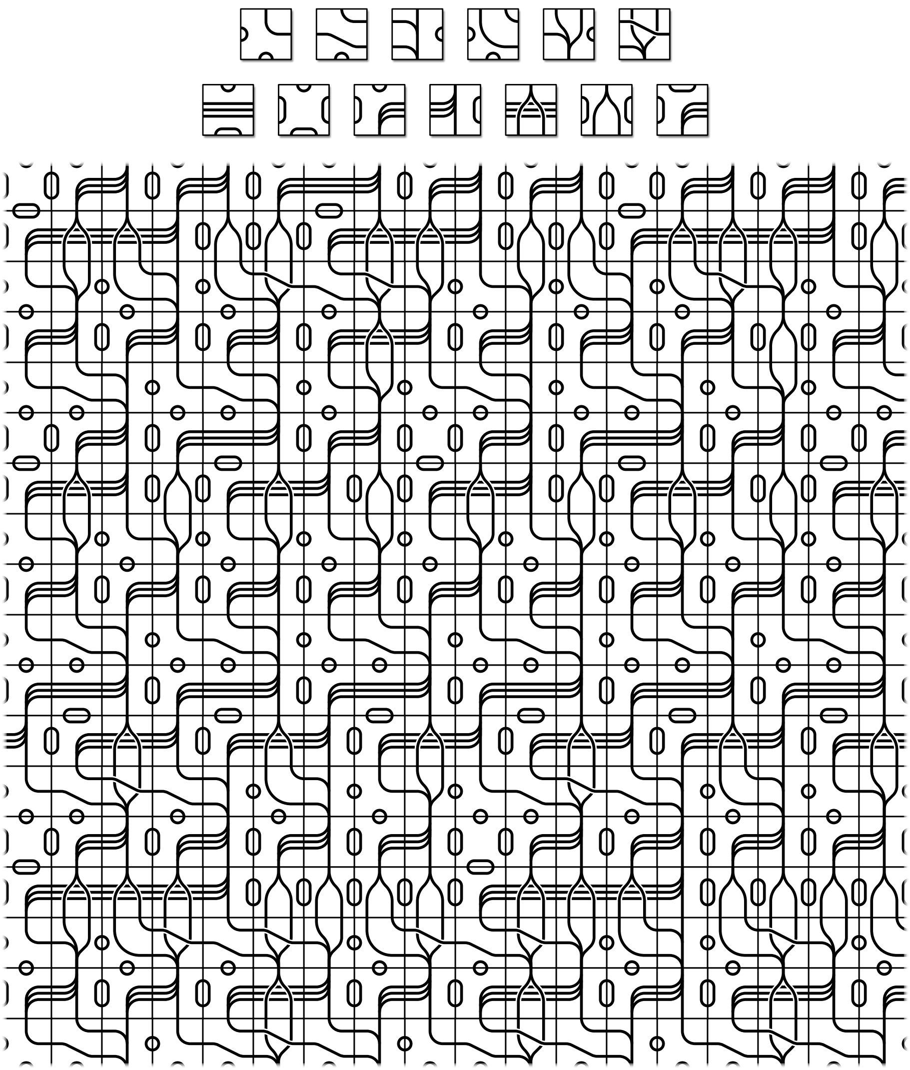
Models of Computation

Jeff Erickson



January 4, 2015

I'm writing a book.
I've got the page numbers done,
so now I just have to fill in the rest.

— Stephen Wright

About These Notes

These are lecture notes that I wrote for the course "Algorithms and Models of Computation" at the University of Illinois, Urbana-Champaign for the first time in Fall 2014. This course is a broad introduction to theoretical computer science, aimed at third-year computer science and computer engineering majors, that covers both fundamental topics in algorithms, for which I already have copious notes, and fundamental topics on formal languages and automata, for which I wrote the notes you are reading now.

The most recent revision of these notes (or nearly so) is available online at http://www.cs.illinois.edu/~jeffe/teaching/algorithms/, along with my algorithms notes and a near-complete archive of past homeworks and exams from all my theoretical computer science classes. I plan to revise and reorganize these whenever I teach this material, so you may find more recent versions on the web page of whatever course I am currently teaching.

About the Exercises

Each note ends with several exercises, many of which I used in homeworks, discussion sections, or exams. *Stars indicate more challenging problems (which I have *not* used in homeworks, discussion sections, or exams). Many of these exercises were contributed by my amazing teaching assistants:

> Alex Steiger, Chao Xu, Connor Clark, Gail Steitz, Grant Czajkowski, Hsien-Chih Chang, Junqing Deng, Nick Bachmair, and Tana Wattanawaroon

Please do not ask me for solutions to the exercises. If you are a student, seeing the solution will rob you of the experience of solving the problem yourself, which is the only way to learn the material. If you are an instructor, you shouldn't ask your students to solve problems that you can't solve yourself. (I don't always follow my own advice, so I'm sure some of the problems are buggy.)

Caveat Lector!

These notes are best viewed as an unfinished first draft. You should assume the notes contain several major errors, in addition to the usual unending supply of typos, fencepost errors, off-by-one errors, and brain farts. Before Fall 2014, I had not taught this material in more than two decades. Moreover, the course itself is still very new—Lenny Pitt and I developed the course and offered the first pilot in Spring 2014 (with Lenny presenting the formal language material)—so even the choice of which material to emphasize, sketch, or exclude is still very much in flux.

I would sincerely appreciate feedback of any kind, especially bug reports.

Thanks, and enjoy!

— *Jeff*

Contents

THOMAS GODFREY, a self-taught mathematician, great in his way, and afterward inventor of what is now called Hadley's Quadrant. But he knew little out of his way, and was not a pleasing companion; as, like most great mathematicians I have met with, he expected universal precision in everything said, or was forever denying or distinguishing upon trifles, to the disturbance of all conversation. He soon left us.

— Benjamin Franklin, *Memoirs, Part 1* (1771)
describing one of the founding members of the Junto

I hope the reader sees that the alphabet can be understood by any intelligent being who has any one of the five senses left him,—by all rational men, that is, excepting the few eyeless deaf persons who have lost both taste and smell in some complete paralysis. . . . Whales in the sea can telegraph as well as senators on land, if they will only note the difference between long spoutings and short ones. . . . A tired listener at church, by properly varying his long yawns and his short ones, may express his opinion of the sermon to the opposite gallery before the sermon is done.

— Edward Everett Hale, "The Dot and Line Alphabet", *Altlantic Monthy* (October 1858)

If indeed, as Hilbert asserted, mathematics is a meaningless game played with meaningless marks on paper, the only mathematical experience to which we can refer is the making of marks on paper.

— Eric Temple Bell, *The Queen of the Sciences* (1931)

1 Strings

Throughout this course, we will discuss dozens of algorithms and computational models that manipulate sequences: one-dimensional arrays, linked lists, blocks of text, walks in graphs, sequences of executed instructions, and so on. Ultimately the input and output of any algorithm must be representable as a finite string of symbols—the raw contents of some contiguous portion of the computer's memory. Reasoning about computation requires reasoning about strings.

This note lists several formal definitions and formal induction proofs related to strings. These definitions and proofs are *intentionally* much more detailed than normally used in practice—most people's intuition about strings is fairly accurate—but the extra precision is necessary for any sort of formal proof. It may be helpful to think of this material as part of the "assembly language" of theoretical computer science. We normally think about computation at a *much* higher level of abstraction, but ultimately every argument must "compile" down to these (and similar) definitions.

1.1 Definitions

Fix an arbitrary finite set $\boldsymbol{\Sigma}$ called the ***alphabet***; the elements of Σ are called ***symbols*** or ***characters***. As a notational convention, I will always use lower-case letters near the start of the English alphabet ($a, b, c, \ldots$) as symbol variables, and *never* as explicit symbols. For explicit symbols, I will always use fixed-width upper-case letters (`A`, `B`, `C`, . . .), digits (`0`, `1`, `2`, . . .), or other symbols (`◇`, `$`, `#`, `•`, . . .) that are clearly distinguishable from variables.

A ***string*** (or ***word***) over Σ is a finite sequence of zero or more symbols from Σ. Formally, a string w over Σ is defined recursively as either

- the empty string, denoted by the Greek letter $\boldsymbol{\varepsilon}$ (epsilon),or
- an ordered pair $\boldsymbol{(a, x)}$, where a is a symbol in Σ and x is a string over Σ.

We normally write either $\boldsymbol{a \cdot x}$ or simply $\boldsymbol{ax}$ to denote the ordered pair (a, x). Similarly, we normally write explicit strings as sequences of symbols instead of nested ordered pairs; for example, `STRING` is convenient shorthand for the formal expression $(\mathtt{S},(\mathtt{T},(\mathtt{R},(\mathtt{I},(\mathtt{N},(\mathtt{G},\varepsilon))))))$. As a notational convention, I will always use lower-case letters near the end of the alphabet $(\ldots, w, x, y, z)$ to represent unknown strings, and `SHOUTY◇MONOSPACED◇TEXT` to represent explicit symbols and (non-empty) strings.

The set of all strings over Σ is denoted $\boldsymbol{\Sigma^*}$ (pronounced "sigma star"). It is very important to remember that every element of Σ^* is a *finite* string, although Σ^* itself is an infinite set containing strings of every possible *finite* length.

The ***length*** $\boldsymbol{|w|}$ of a string w is the number of symbols in w, defined formally as follows:

$$|w| := \begin{cases} 0 & \text{if } w = \varepsilon, \\ 1 + |x| & \text{if } w = ax. \end{cases}$$

For example, the string `SEVEN` has length 5. Although they are formally different objects, we do not normally distinguish between symbols and strings of length 1.

The ***concatenation*** of two strings x and y, denoted either $\boldsymbol{x \bullet y}$ or simply $\boldsymbol{xy}$, is the unique string containing the characters of x in order, followed by the characters in y in order. For example, the string `NOWHERE` is the concatenation of the strings `NOW` and `HERE`; that is, `NOW` • `HERE` = `NOWHERE`. (On the other hand, `HERE` • `NOW` = `HERENOW`.) Formally, concatenation is defined recusively as follows:

$$w \bullet z := \begin{cases} z & \text{if } w = \varepsilon \\ a \cdot (x \bullet z) & \text{if } w = ax \end{cases}$$

(Here I'm using a larger dot • to formally distinguish the operator that concatenates two arbitrary strings from from the operator $\cdot$ that builds a string from a single character and a string.)

When we describe the concatenation of more than two strings, we normally omit all dots and parentheses, writing $wxyz$ instead of $(w \bullet (x \bullet y)) \bullet z$, for example. This simplification is justified by the fact (which we will prove shortly) that • is associative.

1.2 Induction on Strings

Induction is *the* standard technique for proving statements about recursively defined objects. Hopefully you are already comfortable proving statements about *natural numbers* via induction, but induction actually a far more general technique. Several different variants of induction can be used to prove statements about more general structures; here I describe the variant that I recommend (and actually use in practice). This variant follows two primary design considerations:

- ***The case structure of the proof should mirror the case structure of the recursive definition.*** For example, if you are proving something about all *strings*, your proof should have two cases: Either $w = \varepsilon$, or $w = ax$ for some symbol a and string x.

- ***The inductive hypothesis should be as strong as possible.*** The (strong) inductive hypothesis for statements about natural numbers is *always* "Assume there is no counterexample k such that $k < n$." I recommend adopting a similar inductive hypothesis for strings: "Assume there is no counterexample x such that $|x| < |w|$." Then for the case $w = ax$, we have $|x| = |w| - 1 < |w|$ by definition of $|w|$, so the inductive hypothesis applies to x.

Thus, string-induction proofs have the following boilerplate structure. Suppose we want to prove that every string is perfectly cromulent, whatever that means. The white boxes hide additional proof details that, among other things, depend on the precise definition of "perfectly cromulent".

> **Proof:** Let w be an arbitrary string.
> Assume, for every string x such that $|x| < |w|$, that x is perfectly cromulent.
> There are two cases to consider.
>
> - Suppose $w = \varepsilon$.
>
>   Therefore, w is perfectly cromulent.
> - Suppose $w = ax$ for some symbol a and string x.
>   The induction hypothesis implies that x is perfectly cromulent.
>
>   Therefore, w is perfectly cromulent.
>
> In both cases, we conclude that w is perfectly cromulent. □

Here are three canonical examples of this proof structure. When developing proofs in this style, I strongly recommend first ***mindlessly*** writing the green text (the boilerplate) with lots of space for each case, then filling in the red text (the actual theorem and the induction hypothesis), and only then starting to actually think.

Lemma 1.1. *For every string w, we have $w \bullet \varepsilon = w$.*

Proof: Let w be an arbitrary string. Assume that $x \bullet \varepsilon = x$ for every string x such that $|x| < |w|$. There are two cases to consider:

- Suppose $w = \varepsilon$.

$$\begin{aligned} w \bullet \varepsilon &= \varepsilon \bullet \varepsilon && \text{because } w = \varepsilon, \\ &= \varepsilon && \text{by definition of concatenation,} \\ &= w && \text{because } w = \varepsilon. \end{aligned}$$

- Suppose $w = ax$ for some symbol a and string x.

$$\begin{aligned} w \bullet \varepsilon &= (a \cdot x) \bullet \varepsilon && \text{because } w = ax, \\ &= a \cdot (x \bullet \varepsilon) && \text{by definition of concatenation,} \\ &= a \cdot x && \text{by the inductive hypothesis,} \\ &= w && \text{because } w = ax. \end{aligned}$$

In both cases, we conclude that $w \bullet \varepsilon = w$. □

Lemma 1.2. *Concatenation adds length: $|w \bullet x| = |w| + |x|$ for all strings w and x.*

Proof: Let w and x be arbitrary strings. Assume that $|y \bullet x| = |y| + |x|$ for every string y such that $|y| < |w|$. (Notice that we are using induction only on w, not on x.) There are two cases to consider:

- Suppose $w = \varepsilon$.

$$\begin{aligned}
|w \bullet x| &= |\varepsilon \bullet x| && \text{because } w = \varepsilon\\
&= |x| && \text{by definition of } |\,|\\
&= |\varepsilon| + |x| && |e| = 0 \text{ by definition of } |\,|\\
&= |w| + |x| && \text{because } w = \varepsilon
\end{aligned}$$

- Suppose $w = ay$ for some symbol a and string y.

$$\begin{aligned}
|w \bullet x| &= |ay \bullet x| && \text{because } w = ay\\
&= |a \cdot (y \bullet x)| && \text{by definition of } \bullet\\
&= 1 + |y \bullet x| && \text{by definition of } |\,|\\
&= 1 + |y| + |x| && \text{by the inductive hypothesis}\\
&= |ay| + |x| && \text{by definition of } |\,|\\
&= |w| + |x| && \text{because } w = ay
\end{aligned}$$

In both cases, we conclude that $|w \bullet x| = |w| + |x|$. $\square$

Lemma 1.3. *Concatenation is* associative*:* $(w \bullet x) \bullet y = w \bullet (x \bullet y)$ *for all strings* w*,* x*, and* y*.*

Proof: Let w, x, and y be arbitrary strings. Assume that $(z \bullet x) \bullet y = w \bullet (x \bullet y)$ for every string z such that $|z| < |w|$. (Again, we are using induction only on w.) There are two cases to consider.

- Suppose $w = \varepsilon$.

$$\begin{aligned}
(w \bullet x) \bullet y &= (\varepsilon \bullet x) \bullet y && \text{because } w = \varepsilon\\
&= x \bullet y && \text{by definition of } \bullet\\
&= \varepsilon \bullet (x \bullet y) && \text{by definition of } \bullet\\
&= w \bullet (x \bullet y) && \text{because } w = \varepsilon
\end{aligned}$$

- Suppose $w = az$ for some symbol a and some string z.

$$\begin{aligned}
(w \bullet x) \bullet y &= (az \bullet x) \bullet y && \text{because } w = az\\
&= (a \cdot (z \bullet x)) \bullet y && \text{by definition of } \bullet\\
&= a \cdot ((z \bullet x) \bullet y) && \text{by definition of } \bullet\\
&= a \cdot (z \bullet (x \bullet y)) && \text{by the inductive hypothesis}\\
&= az \bullet (x \bullet y) && \text{by definition of } \bullet\\
&= w \bullet (x \bullet y) && \text{because } w = az
\end{aligned}$$

In both cases, we conclude that $(w \bullet x) \bullet y = w \bullet (x \bullet y)$. $\square$

This is not the only boilerplate that one can use for induction proofs on strings. For example, we can modify the inductive case analysis using the following observation: A *non-empty* string w is either a single symbol or the concatenation of two non-empty strings, which (by Lemma 1.2) must be shorter than w. Here is a proof of Lemma 1.3 that uses this alternative recursive structure:

Proof: Let w, x, and y be arbitrary strings. Assume that $(z \bullet x') \bullet y' = z \bullet (x' \bullet y')$ for all strings x', y', and z such that $|z| < |w|$. (We need a stronger induction hypothesis here than in the previous proofs!) There are ***three*** cases to consider.

- Suppose $w = \varepsilon$.

$$\begin{aligned}
(w \bullet x) \bullet y &= (\varepsilon \bullet x) \bullet y && \text{because } w = \varepsilon \\
&= x \bullet y && \text{by definition of } \bullet \\
&= \varepsilon \bullet (x \bullet y) && \text{by definition of } \bullet \\
&= w \bullet (x \bullet y) && \text{because } w = \varepsilon
\end{aligned}$$

- Suppose w is equal to some symbol a.

$$\begin{aligned}
(w \bullet x) \bullet y &= (a \bullet x) \bullet y && \text{because } w = a \\
&= (a \cdot x) \bullet y && \text{because } a \bullet z = a \cdot z \text{ by definition of } \bullet \\
&= a \cdot (x \bullet y) && \text{by definition of } \bullet \\
&= a \bullet (x \bullet y) && \text{because } a \bullet z = a \cdot z \text{ by definition of } \bullet \\
&= w \bullet (x \bullet y) && \text{because } w = a
\end{aligned}$$

- Suppose $w = uv$ for some nonempty strings u and v.

$$\begin{aligned}
(w \bullet x) \bullet y &= ((u \bullet v) \bullet x) \bullet y && \text{because } w = uv \\
&= (u \bullet (v \bullet x)) \bullet y && \text{by the inductive hypothesis, because } |u| < |w| \\
&= u \bullet ((v \bullet x) \bullet y) && \text{by the inductive hypothesis, because } |u| < |w| \\
&= u \bullet (v \bullet (x \bullet y)) && \text{by the inductive hypothesis, because } |v| < |w| \\
&= (u \bullet v) \bullet (x \bullet y) && \text{by the inductive hypothesis, because } |u| < |w| \\
&= w \bullet (x \bullet y) && \text{because } w = uv
\end{aligned}$$

In both cases, we conclude that $(w \bullet x) \bullet y = w \bullet (x \bullet y)$. □

1.3 Indices, Substrings, and Subsequences

For any string w and any integer $1 \le i \le |w|$, the expression $\boldsymbol{w_i}$ denotes the ith symbol in w, counting from left to right. More formally, w_i is recursively defined as follows:

$$w_i := \begin{cases} a & \text{if } w = ax \text{ and } i = 1 \\ x_{i-1} & \text{if } w = ax \text{ and } i > 1 \end{cases}$$

As one might reasonably expect, w_i is formally undefined if $i < 1$ or $w = \varepsilon$, and therefore (by induction) if $i > |w|$. The integer i is called the ***index*** of w_i.

We sometimes write strings as a concatenation of their constituent symbols using this subscript notation: $w = w_1 w_2 \cdots w_{|w|}$. While standard, this notation is slightly misleading, since it *incorrectly* suggests that the string w contains at least three symbols, when in fact w could be a single symbol or even the empty string.

In actual code, subscripts are usually expressed using the bracket notation $\boldsymbol{w[i]}$. Brackets were introduced as a typographical convention over a hundred years ago because subscripts and superscripts[1] were difficult or impossible to type.[2] We sometimes write strings as explicit arrays $w[1\,..\,n]$, with the understanding that $n = |w|$. Again, this notation is potentially misleading; always remember that n might be zero; the string/array could be empty.

A ***substring*** of a string w is another string obtained from w by deleting zero or more symbols from the beginning and from the end. Formally, a string y is a substring of w if and only if there are strings x and z such that $w = xyz$. Extending the array notation for strings, we write $\boldsymbol{w[i\,..\,j]}$ to denote the substring of w starting at w_i and ending at w_j. More formally, we define

$$w[i\,..\,j] := \begin{cases} \varepsilon & \text{if } j < i, \\ w_i \cdot w[i+1\,..\,j] & \text{otherwise.} \end{cases}$$

A ***proper substring*** of w is any substring other than w itself. For example, `LAUGH` is a proper substring of `SLAUGHTER`. Whenever y is a (proper) substring of w, we also call w a (proper) ***superstring*** of y.

A ***prefix*** of $w[1\,..\,n]$ is any substring of the form $w[1\,..\,j]$. Equivalently, a string p is a ***prefix*** of another string w if and only if there is a string x such that $px = w$. A ***proper prefix*** of w is any prefix except w itself. For example, `DIE` is a proper prefix of `DIET`.

Similarly, a suffix of $w[1\,..\,n]$ is any substring of the form $w[i\,..\,n]$. Equivalently, a string s is a ***suffix*** of a string w if and only if there is a string x such that $xs = w$. A ***proper suffix*** of w is any suffix except w itself. For example, `YES` is a proper suffix of `EYES`, and `HE` is both a proper prefix and a proper suffix of `HEADACHE`.

A ***subsequence*** of a string w is a strong obtained by deleting zero or more symbols from *anywhere* in w. More formally, z is a subsequence of w if and only if

- $z = \varepsilon$, or
- $w = ax$ for some symbol a and some string x such that z is a subsequence of x.
- $w = ax$ and $z = ay$ for some symbol a and some strings x and y, and y is a subsequence of x.

A ***proper subsequence*** of w is any subsequence of w other than w itself. Whenever z is a (proper) subsequence of w, we also call w a (proper) ***supersequence*** of z.

---

[1]The same bracket notation is also used for bibliographic references, instead of the traditional footnote/endnote superscripts, for exactly the same reasons.

[2]A ***typewriter*** is an obsolete mechanical device loosely resembling a computer keyboard. Pressing a key on a typewriter moves a lever (called a "typebar") that strikes a cloth ribbon full of ink against a piece of paper, leaving the image of a single character. Many historians believe that the ordering of letters on modern keyboards (`QWERTYUIOP`) evolved in the late 1800s, reaching its modern form on the 1874 Sholes & Glidden Type-Writer™, in part to separate many common letter pairs, to prevent typebars from jamming against each other; this is also why the keys on most modern keyboards are arranged in a slanted grid. (The common folk theory that the ordering was deliberately intended to slow down typists doesn't withstand careful scrutiny.) A more recent theory suggests that the ordering was influenced by telegraph[3] operators, who found older alphabetic arrangements confusing, in part because of ambiguities in American Morse Code.

[3]A ***telegraph*** is an obsolete electromechanical communication device consisting of an electrical circuit with a switch at one end and an electromagnet at the other. The sending operator would press and release a key, closing and opening the circuit, originally causing the electromagnet to push a stylus onto a moving paper tape, leaving marks that could be decoded by the receiving operator. (Operators quickly discovered that they could directly decode the clicking sounds made by the electromagnet, and so the paper tape became obsolete almost immediately.) The most common scheme within the US to encode symbols, developed by Alfred Vail and Samuel Morse in 1837, used (mostly) short (·) and long (—) marks—now called "dots" and "dashes", or "dits" and "dahs"—separated by gaps of various lengths. American Morse code (as it became known) was ambiguous; for example, the letter `Z` and the string `SE` were both encoded by the sequence $\cdots\ \cdot$ ("di-di-dit, dit"). This ambiguity has been blamed for the `S` key's position on the

Substrings and subsequences are not the same objects; don't confuse them! Every substring of w is also a subsequence of w, but not every subsequence is a substring. For example, `METAL` is a subsequence, but not a substring, of `MEATBALL`. To emphasize the distinction, we sometimes redundantly refer to substrings of w as ***contiguous*** substrings, meaning all their symbols appear together in w.

Exercises

Most of the following exercises ask for proofs of various claims about strings. For each claim, give a complete, self-contained, formal proof by inductive definition-chasing, using the boilerplate structure recommended in Section 1.2. You can use Lemmas 1.1, 1.2, and 1.3, but don't assume any other facts about strings that you have not actually proved. Do not use the words "obvious" or "clearly" or "just". Most of these claims **are** in fact obvious; the real exercise is understanding **why** they're obvious.

1. For any symbol a and any string w, let $\boldsymbol{\#(a, w)}$ denote the number of occurrences of a in w. For example, $\#(\texttt{A}, \texttt{BANANA}) = 3$ and $\#(\texttt{X}, \texttt{FLIBBERTIGIBBET}) = 0$.

   (a) Give a formal recursive definition of the function $\# \colon \Sigma \times \Sigma^* \to \mathbb{N}$.

   (b) Prove that $\#(a, xy) = \#(a, x) + \#(a, y)$ for every symbol a and all strings x and y. Your proof must rely on both your answer to part (a) and the formal recursive definition of string concatenation.

2. Recursively define a set L of strings over the alphabet $\{\texttt{0}, \texttt{1}\}$ as follows:

   - The empty string ε is in L.
   - For any two strings x and y in L, the string $\texttt{0}x\texttt{1}y\texttt{0}$ is also in L.
   - These are the only strings in L.

   (a) Prove that the string `000010101010010100` is in L.

   (b) Prove by induction that every string in L has exactly twice as many `0`s as `1`s. (You may assume the identity $\#(a, xy) = \#(a, x) + \#(a, y)$ for any symbol a and any strings x and y; see Exercise 1(b).)

   (c) Give an example of a string with exactly twice as many `0`s as `1`s that is *not* in L.

3. For any string w and any non-negative integer n, let $\boldsymbol{w^n}$ denote the string obtained by concatenating n copies of w; more formally, we define

$$w^n := \begin{cases} \varepsilon & \text{if } n = 0 \\ w \bullet w^{n-1} & \text{otherwise} \end{cases}$$

   For example, $(\texttt{BLAH})^5 = \texttt{BLAHBLAHBLAHBLAHBLAH}$ and $\varepsilon^{374} = \varepsilon$.

   Prove that $w^m \bullet w^n = w^{m+n}$ for every string w and all integers non-negative integer n and m.

typewriter keyboard near `E` and `Z`.

Vail and Morse were of course not the first people to propose encoding symbols as strings of bits. That honor apparently falls to Francis Bacon, who devised a five-bit binary encoding of the alphabet (except for the letters `J` and `U`) in 1605 as the basis for a steganographic code—a method or hiding secret message in otherwise normal text.

4. Let w be an arbitrary string, and let $n = |w|$. Prove each of the following statements.

   (a) w has exactly $n+1$ prefixes.

   (b) w has exactly n proper suffixes.

   (c) w has at most $n(n+1)/2$ distinct substrings.

   (d) w has at most $2^n - 1$ proper subsequences.

5. The ***reversal*** $\boldsymbol{w^R}$ of a string w is defined recursively as follows:

$$w^R := \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ x^R \bullet a & \text{if } w = a \cdot x \end{cases}$$

   (a) Prove that $|w^R| = |w|$ for every string w.

   (b) Prove that $(wx)^R = x^R w^R$ for all strings w and x.

   (c) Prove that $(w^R)^n = (w^n)^R$ for every string w and every integer $n \geq 0$. (See Exercise 1.)

   (d) Prove that $(w^R)^R = w$ for every string w.

6. Let w be an arbitrary string, and let $n = |w|$. Prove the following statements for all indices $1 \leq i \leq j \leq k \leq n$.

   (a) $|w[i \mathinner{..} j]| = j - i + 1$

   (b) $w[i \mathinner{..} j] \bullet w[j+1 \mathinner{..} k] = w[i \mathinner{..} k]$

   (c) $w^R[i \mathinner{..} j] = (w[i' \mathinner{..} j'])^R$ where $i' = |w| + 1 - j$ and $j' = |w| + 1 - i$.

7. A ***palindrome*** is a string that is equal to its reversal.

   (a) Give a recursive definition of a palindrome over the alphabet Σ.

   (b) Prove that any string p meets your recursive definition of a palindrome if and only if $p = p^R$.

8. A string $w \in \Sigma^*$ is called a ***shuffle*** of two strings $x, y \in \Sigma^*$ if at least one of the following recursive conditions is satisfied:

   - $w = x = y = \varepsilon$.
   - $w = aw'$ and $x = ax'$ and w' is a shuffle of x' and y, for some $a \in \Sigma$ and some $w', x' \in \Sigma^*$.
   - $w = aw'$ and $y = ay'$ and w' is a shuffle of x and y', for some $a \in \Sigma$ and some $w', y' \in \Sigma^*$.

   For example, the string BANANANANASA is a shuffle of the strings BANANA and ANANAS.

   (a) Prove that if w is a shuffle of x and y, then $|w| = |x| + |y|$.

   (b) Prove that if w is a shuffle of x and y, then w^R is a shuffle of x^R and y^R.

9. Consider the following pair of mutually recursive functions on strings:

$$evens(w) := \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ odds(x) & \text{if } w = ax \end{cases} \qquad odds(w) := \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ a \cdot evens(x) & \text{if } w = ax \end{cases}$$

   (a) Prove the following identity for all strings w and x:

$$evens(w \bullet x) = \begin{cases} evens(w) \bullet evens(x) & \text{if } |w| \text{ is even,} \\ evens(w) \bullet odds(x) & \text{if } |w| \text{ is odd.} \end{cases}$$

   (b) State and prove a similar identity for $odds(w \bullet x)$.

10. For any positive integer n, the ***Fibonacci string*** $\boldsymbol{F_n}$ is defined recursively as follows:

$$F_n = \begin{cases} \texttt{0} & \text{if } n = 1, \\ \texttt{1} & \text{if } n = 2, \\ F_{n-2} \bullet F_{n-1} & \text{otherwise.} \end{cases}$$

For example, $F_6 = \texttt{10101101}$ and $F_7 = \texttt{0110110101101}$.

   (a) Prove that for every integer $n \geq 2$, the string F_n can also be obtained from F_{n-1} by replacing every occurrence of `0` with `1` and replacing every occurrence of `1` with `01`. More formally, prove that $F_n = Finc(F_{n-1})$, where

$$Finc(w) = \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ \texttt{1} \cdot Finc(x) & \text{if } w = \texttt{0}x \\ \texttt{01} \bullet Finc(x) & \text{if } w = \texttt{1}x \end{cases}$$

   [Hint: First prove that $\mathrm{Finc}(x \bullet y) = \mathrm{Finc}(x) \bullet \mathrm{Finc}(y)$.*]*

   (b) Prove that `00` and `111` are not substrings of any Fibonacci string F_n.

11. Prove that the following three properties of strings are in fact identical.

   - A string $w \in \{\texttt{0}, \texttt{1}\}^*$ is ***balanced*** if it satisfies one of the following conditions:
     - $w = \varepsilon$,
     - $w = \texttt{0}x\texttt{1}$ for some balanced string x, or
     - $w = xy$ for some balanced strings x and y.
   - A string $w \in \{\texttt{0}, \texttt{1}\}^*$ is ***erasable*** if it satisfies one of the following conditions:
     - $w = \varepsilon$, or
     - $w = x\texttt{01}y$ for some strings x and y such that xy is erasable. (The strings x and y are not necessarily erasable.)
   - A string $w \in \{\texttt{0}, \texttt{1}\}^*$ is ***conservative*** if it satisfies ***both*** of the following conditions:
     - w has an equal number of `0`s and `1`s, and
     - no prefix of w has more `0`s than `1`s.

   (a) Prove that every balanced string is erasable.

(b) Prove that every erasable string is conservative.

(c) Prove that every conservative string is balanced.

[Hint: To develop intuition, it may be helpful to think of `0`*s as left brackets and* `1`*s as right brackets, but* ***don't*** *invoke this intuition in your proofs.]*

12. A string $w \in \{\mathtt{0}, \mathtt{1}\}^*$ ***equitable*** if it has an equal number of `0`s and `1`s.

    (a) Prove that a string w is equitable if and only if it satisfies one of the following conditions:

    - $w = \varepsilon$,
    - $w = \mathtt{0}x\mathtt{1}$ for some equitable string x,
    - $w = \mathtt{1}x\mathtt{0}$ for some equitable string x, or
    - $w = xy$ for some equitable strings x and y.

    (b) Prove that a string w is equitable if and only if it satisfies one of the following conditions:

    - $w = \varepsilon$,
    - $w = x\mathtt{01}y$ for some strings x and y such that xy is equitable, or
    - $w = x\mathtt{10}y$ for some strings x and y such that xy is equitable.

    In the last two cases, the individual strings x and y are not necessarily equitable.

    (c) Prove that a string w is equitable if and only if it satisfies one of the following conditions:

    - $w = \varepsilon$,
    - $w = xy$ for some balanced string x and some equitable string y, or
    - $w = x^R y$ for some for some balanced string x and some equitable string y.

    (See the previous exercise for the definition of "balanced".)

> **Caveat lector:** This is the first edition of this lecture note. Please send bug reports and suggestions to jeffe@illinois.edu.

But the Lord came down to see the city and the tower the people were building. The Lord said, "If as one people speaking the same language they have begun to do this, then nothing they plan to do will be impossible for them. Come, let us go down and confuse their language so they will not understand each other."

— Genesis 11:6–7 (New International Version)

Soyez réglé dans votre vie et ordinaire comme un bourgeois,
afin d'être violent et original dans vos œuvres.

[Be regular and orderly in your life like a bourgeois,
so that you may be violent and original in your work.]

— Gustave Flaubert, in a letter to Gertrude Tennant (December 25, 1876)

Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems.

— Jamie Zawinski, alt.religion.emacs (August 12, 1997)

I define UNIX as 30 definitions of regular expressions living under one roof.

— Donald Knuth, *Digital Typography* (1999)

2 Regular Languages

2.1 Languages

A ***formal language*** (or just a ***language***) is a set of strings over some finite alphabet Σ, or equivalently, an arbitrary subset of Σ^*. For example, each of the following sets is a language:

- The empty set $\varnothing$.[1]
- The set $\{\varepsilon\}$.
- The set $\{\mathtt{0}, \mathtt{1}\}^*$.
- The set $\{\mathtt{THE}, \mathtt{OXFORD}, \mathtt{ENGLISH}, \mathtt{DICTIONARY}\}$.
- The set of all subsequences of $\mathtt{THE}\diamond\mathtt{OXFORD}\diamond\mathtt{ENGLISH}\diamond\mathtt{DICTIONARY}$.
- The set of all words in *The Oxford English Dictionary*.
- The set of all strings in $\{\mathtt{0}, \mathtt{1}\}^*$ with an odd number of $\mathtt{1}$s.
- The set of all strings in $\{\mathtt{0}, \mathtt{1}\}^*$ that represent a prime number in base 13.
- The set of all sequences of turns that solve the Rubik's cube (starting in some fixed configuration)
- The set of all python programs that print "Hello World!"

As a notational convention, I will always use italic upper-case letters (usually L, but also A, B, C, and so on) to represent languages.

[1]The empty set symbol $\varnothing$ derives from the Norwegian letter Ø, pronounced like a sound of disgust or a German ö, and *not* from the Greek letter ϕ. Calling the empty set "fie" or "fee" makes the baby Jesus cry.

Formal languages are not "languages" in the same sense that English, Klingon, and Python are "languages". Strings in a formal language do not necessarily carry any "meaning", nor are they necessarily assembled into larger units ("sentences" or "paragraphs" or "packages") according to some "grammar".

It is ***very*** important to distinguish between three "empty" objects. Many beginning students have trouble keeping these straight.

- $\varnothing$ is the empty *language*, which is a set containing zero strings. $\varnothing$ is not a string.
- $\boldsymbol{\{\varepsilon\}}$ is a language containing exactly one string, which has length zero. $\{\varepsilon\}$ is not empty, and it is not a string.
- $\boldsymbol{\varepsilon}$ is the empty *string*, which is a sequence of length zero. ε is not a language.

2.2 Building Languages

Languages can be combined and manipulated just like any other sets. Thus, if A and B are languages over Σ, then their union $A \cup B$, intersection $A \cap B$, difference $A \setminus B$, and symmetric difference $A \oplus B$ are also languages over Σ, as is the complement $\overline{A} := \Sigma^* \setminus A$. However, there are two more useful operators that are specific to sets of *strings*.

The ***concatenation*** of two languages A and B, again denoted $\boldsymbol{A \bullet B}$ or just $\boldsymbol{AB}$, is the set of all strings obtained by concatenating an arbitrary string in A with an arbitrary string in B:

$$A \bullet B := \{xy \mid x \in A \text{ and } y \in B\}.$$

For example, if $A = \{\texttt{HOCUS}, \texttt{ABRACA}\}$ and $B = \{\texttt{POCUS}, \texttt{DABRA}\}$, then

$$A \bullet B = \{\texttt{HOCUSPOCUS}, \texttt{ABRACAPOCUS}, \texttt{HOCUSDABRA}, \texttt{ABRACADABRA}\}.$$

In particular, for every language A, we have

$$\varnothing \bullet A = A \bullet \varnothing = \varnothing \qquad \text{and} \qquad \{\varepsilon\} \bullet A = A \bullet \{\varepsilon\} = A.$$

The ***Kleene closure*** or ***Kleene star***[2] of a language L, denoted $\boldsymbol{L^*}$, is the set of all strings obtained by concatenating a sequence of zero or more strings from L. For example, $\{\texttt{0}, \texttt{11}\}^* = \{\varepsilon, \texttt{0}, \texttt{00}, \texttt{11}, \texttt{000}, \texttt{011}, \texttt{110}, \texttt{0000}, \texttt{0011}, \texttt{0110}, \texttt{1100}, \texttt{1111}, \texttt{00000}, \texttt{00011}, \texttt{00110}, \dots, \texttt{011110011011}, \dots\}$. More formally, L^* is defined recursively as the set of all strings w such that either

- $w = \varepsilon$, or
- $w = xy$, for some strings $x \in L$ and $y \in L^*$.

This definition immediately implies that

$$\varnothing^* = \{\varepsilon\}^* = \{\varepsilon\}.$$

For any other language L, the Kleene closure L^* is infinite and contains arbitrarily long (but *finite!*) strings. Equivalently, L^* can also be defined as the smallest superset of L that contains the empty string ε and is closed under concatenation (hence "closure"). The set of all strings Σ^* is, just as the notation suggests, the Kleene closure of the alphabet Σ (where each symbol is viewed as a string of length 1).

[2]after Stephen Kleene, who pronounced his last name "***clay***-knee", not "clean" or "cleanie" or "claynuh" or "dimaggio".

A useful variant of the Kleene closure operator is the ***Kleene plus***, defined as $L^+ := L \bullet L^*$. Thus, L^+ is the set of all strings obtained by concatenating a sequence of ***one*** or more strings from L.

The following identities, which we state here without (easy) proofs, are useful for designing, simplifying, and understanding languages.

Lemma 2.1. *The following identities hold for all languages A, B, and C:*
(a) $\varnothing A = A\varnothing = \varnothing$.
(b) $\varepsilon A = A\varepsilon = A$.
(c) $A + B = B + A$.
(d) $(A + B) + C = A + (B + C)$.
(e) $(AB)C = A(BC)$.
(f) $A(B + C) = AB + AC$.

Lemma 2.2. *The following identities hold for every language L:*
(a) $L^* = \varepsilon + L^+ = L^*L^* = (L + \varepsilon)^* = (L \setminus \varepsilon)^* = \varepsilon + L + L^+L^+$.
(b) $L^+ = L^* \setminus \varepsilon = LL^* = L^*L = L^+L^* = L^*L^+ = L + L^+L^+$.
(c) $L^+ = L^*$ *if and only if* $\varepsilon \in L$.

Lemma 2.3 (Arden's Rule). *For any languages A, B, and L such that $L = AL + B$, we have $A^*B \subseteq L$. Moreover, if A does not contain the empty string, then $L = AL + B$ if and only if $L = A^*B$.*

2.3 Regular Languages and Regular Expressions

A language L is ***regular*** if and only if it satisfies one of the following (recursive) conditions:

- L is empty;
- L contains a single string (which could be the empty string ε);
- L is the union of two regular languages;
- L is the concatenation of two regular languages; or
- L is the Kleene closure of a regular language.

Regular languages are normally described using more compact notation, which omits braces around one-string sets, uses $+$ to represent union instead of $\cup$, and juxtaposes subexpressions to represent concatenation instead of using an explicit operator $\bullet$; the resulting string of symbols is called a ***regular expression***. By convention, in the absence of parentheses, the * operator has highest precedence, followed by the (implicit) concatenation operator, followed by $+$. Thus, for example, the regular expression $\mathtt{10}^*$ is shorthand for the language $\{\mathtt{1}\} \bullet \{\mathtt{0}\}^*$ (containing all strings consisting of a `1` followed by zero or more `0`s), and *not* the language $\{\mathtt{10}\}^*$ (containing all strings of even length that start with `1` and alternate between `1` and `0`). As a larger example, the regular expression

$$\mathtt{0} + \mathtt{0}^*\mathtt{1}(\mathtt{10}^*\mathtt{1} + \mathtt{01}^*\mathtt{0})^*\mathtt{10}^*$$

represents the language

$$\{\mathtt{0}\} \cup \Big(\{\mathtt{0}\}^* \bullet \{\mathtt{1}\} \bullet \Big((\{\mathtt{1}\} \bullet \{\mathtt{0}\}^* \bullet \{\mathtt{1}\}) \cup (\{\mathtt{0}\} \bullet \{\mathtt{1}\}^* \bullet \{\mathtt{0}\})\Big)^* \bullet \{\mathtt{1}\} \bullet \{\mathtt{0}\}^*\Big).$$

Here are a few more examples of regular expressions and the languages they represent.

- $\mathtt{0}^*$ — the set of all strings of `0`s, including the empty string.

- $\mathtt{00000}^*$ — the set of all strings consisting of at least four 0s.
- $(\mathtt{00000})^*$ — the set of all strings of 0s whose length is a multiple of 5.
- $(\varepsilon + \mathtt{1})(\mathtt{01})^*(\varepsilon + \mathtt{0})$ — the set of all strings of alternating 0s and 1s, or equivalently, the set of all binary strings that do not contain the substrings 00 or 11.
- $((\varepsilon + \mathtt{0} + \mathtt{00} + \mathtt{000})\mathtt{1})^*(\varepsilon + \mathtt{0} + \mathtt{00} + \mathtt{000})$ — the set of all binary strings that do not contain the substring 0000.
- $((\mathtt{0} + \mathtt{1})(\mathtt{0} + \mathtt{1}))^*$ — the set of all binary strings whose length is even.
- $\mathtt{1}^*(\mathtt{01}^*\mathtt{01}^*)^*$ — the set of all binary strings with an even number of 0s.
- $\mathtt{0} + \mathtt{1}(\mathtt{0} + \mathtt{1})^*\mathtt{00}$ — the set of all non-negative binary numerals divisible by 4 and with no redundant leading 0s.
- $\mathtt{0} + \mathtt{0}^*\mathtt{1}(\mathtt{10}^*\mathtt{1} + \mathtt{01}^*\mathtt{0})^*\mathtt{10}^*$ — the set of all non-negative binary numerals divisible by 3, possibly with redundant leading 0s.

The last example should ***not*** be obvious. It is straightforward, but rather tedious, to prove by induction that every string in $\mathtt{0} + \mathtt{0}^*\mathtt{1}(\mathtt{10}^*\mathtt{1} + \mathtt{01}^*\mathtt{0})^*\mathtt{10}^*$ is the binary representation of a non-negative multiple of 3. It is similarly straightforward, and similarly tedious, to prove that the binary representation of *every* non-negative multiple of 3 matches this regular expression. In a later note, we will see a systematic method for deriving regular expressions for some languages that avoids (or more accurately, automates) this tedium.

Most of the time we do not distinguish between regular expressions and the languages they represent, for the same reason that we do not normally distinguish between the arithmetic expression "2+2" and the integer 4, or the symbol π and the area of the unit circle. However, we sometimes need to refer to regular expressions themselves *as strings*. In those circumstances, we write $\boldsymbol{L(R)}$ to denote the language represented by the regular expression R. String w ***matches*** regular expression R if and only if $w \in L(R)$. Two regular expressions R and R' are ***equivalent*** if they describe the same language; for example, the regular expressions $(\mathtt{0} + \mathtt{1})^*$ and $(\mathtt{1} + \mathtt{0})^*$ are equivalent, because the union operator is commutative.

Almost every regular language can be represented by infinitely many distinct but equivalent regular expressions, even if we ignore ultimately trivial equivalences like $L = (L\varnothing)^*L\varepsilon + \varnothing$.

2.4 Things What Ain't Regular Expressions

Many computing environments and programming languages support patterns called ***regexen*** (singular ***regex***, pluralized like *ox*) that are considerably more general and powerful than regular expressions. Regexen include special symbols representing negation, character classes (for example, upper-case letters, or digits), contiguous ranges of characters, line and word boundaries, limited repetition (as opposed to the unlimited repetition allowed by *), back-references to earlier subexpressions, and even local variables. Despite its obvious etymology, a regex is ***not*** necessarily a regular expression, and it does ***not*** necessarily describe a regular language![3]

Another type of pattern that is often confused with regular expression are ***globs***, which are patterns used in most Unix shells and some scripting languages to represent sets file names. Globs include symbols for arbitrary single characters (?), single characters from a

[3]However, regexen are not all-powerful, either; see http://stackoverflow.com/a/1732454/775369.

specified range (`[a-z]`), arbitrary substrings (`*`), and substrings from a specified finite set (`{foo,ba{r,z}}`). Globs are significantly *less* powerful than regular expressions.

2.5 Not Every Language is Regular

You may be tempted to conjecture that *all* languages are regular, but in fact, the following cardinality argument *almost all* languages are *not* regular. To make the argument concrete, let's consider languages over the single-symbol alphabet $\{\diamond\}$.

- Every regular expression over the one-symbol alphabet $\{\diamond\}$ is itself a string over the 7-symbol alphabet $\{\diamond, +, (,), *, \varepsilon, \emptyset\}$. By interpreting these symbols as the digits 1 through 7, we can interpret any string over this larger alphabet as the base-8 representation of some unique integer. Thus, the set of all regular expressions over $\{\diamond\}$ is *at most* as large as the set of integers, and is therefore countably infinite. It follows that the set of all regular *languages* over $\{\diamond\}$ is also countably infinite.

- On the other hand, for any real number $0 \le \alpha < 1$, we can define a corresponding language

$$L_\alpha = \left\{ \diamond^n \;\middle|\; \alpha 2^n \bmod 1 \ge 1/2 \right\}.$$

  In other words, L_α contains the string $\diamond^n$ if and only if the $(n+1)$th bit in the binary representation of α is equal to 1. For any distinct real numbers $\alpha \ne \beta$, the binary representations of α and β must differ in some bit, so $L_\alpha \ne L_\beta$. We conclude that the set of ***all*** languages over $\{\diamond\}$ is *at least* as large as the set of real numbers between 0 and 1, and is therefore uncountably infinite.

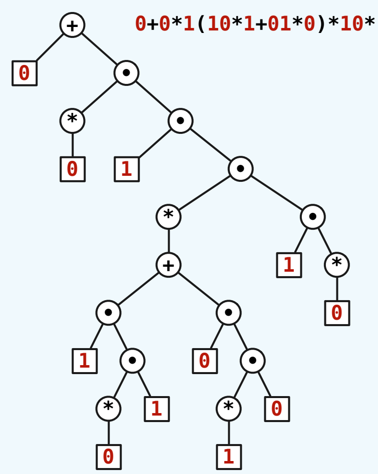
We will see several explicit examples of non-regular languages in future lectures. For example, the set of all regular expressions over $\{0, 1\}$ is not itself a regular language!

2.6 Parsing Regular Expressions

Most algorithms for regular expressions require them in the form of *regular expression trees*, rather than as raw strings. A regular expression tree is one of the following:

- A leaf node labeled $\varnothing$.
- A leaf node labeled with a string in Σ^*.
- A node labeled $+$ with two children, each the root of an expression tree.
- A node labeled $*$ with one child, which is the root of an expression tree.
- A node labeled $\bullet$ with two children, each the root of an expression tree.

In other words, a regular expression tree directly encodes a sequence of alternation, concatenation and Kleene closure operations that defines a regular language. Similarly, when we want to prove things about regular expressions or regular languages, it is more natural to think of subexpressions *as subtrees* rather than *as substrings*.



Given any regular expression of length n, we can ***parse*** it into an equivalent regular expression tree in $O(n)$ time. Thus, when we see an algorithmic problem that starts "Given a regular expression...", we can assume without loss of generality that we're actually given a regular expression *tree*.

We'll see more on this topic later.

Exercises

1. (a) Prove that $\{\varepsilon\} \bullet L = L \bullet \{\varepsilon\} = L$, for any language L.
   (b) Prove that $\varnothing \bullet L = L \bullet \varnothing = \varnothing$, for any language L.
   (c) Prove that $(A \bullet B) \bullet C = A \bullet (B \bullet C)$, for all languages A, B, and C.
   (d) Prove that $|A \bullet B| = |A| \cdot |B|$, for all languages A and B. (The second $\cdot$ is multiplication!)
   (e) Prove that L^* is finite if and only if $L = \varnothing$ or $L = \{\varepsilon\}$.
   (f) Prove that $AB = BC$ implies $A^*B = BC^* = A^*BC^*$, for all languages A, B, and C.

2. Recall that the reversal w^R of a string w is defined recursively as follows:

$$w^R := \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ x^R \bullet a & \text{if } w = a \cdot x \end{cases}$$

The reversal L^R of any language L is the set of reversals of all strings in L:

$$L^R := \{w^R \mid w \in L\}.$$

(a) Prove that $(AB)^R = B^R A^R$ for all languages A and B.

(b) Prove that $(L^R)^R = L$ for every language L.

(c) Prove that $(L^*)^R = (L^R)^*$ for every language L.

3. Prove that each of the following regular expressions is equivalent to $(\texttt{0} + \texttt{1})^*$.

(a) $\varepsilon + \texttt{0}(\texttt{0} + \texttt{1})^* + \texttt{1}(\texttt{1} + \texttt{0})^*$

(b) $\texttt{0}^* + \texttt{0}^*\texttt{1}(\texttt{0} + \texttt{1})^*$

(c) $((\varepsilon + \texttt{0})(\varepsilon + \texttt{1}))^*$

(d) $\texttt{0}^*(\texttt{1}\texttt{0}^*)^*$

(e) $(\texttt{1}^*\texttt{0})^*(\texttt{0}^*\texttt{1})^*$

4. For each of the following languages in $\{\texttt{0}, \texttt{1}\}^*$, describe an equivalent regular expression. There are infinitely many correct answers for each language. (This problem will become significantly simpler after we've seen finite-state machines, in the next lecture note.)

(a) Strings that end with the suffix $\texttt{0}^9 = \texttt{000000000}$.

(b) All strings except 010.

(c) Strings that contain the substring 010.

(d) Strings that contain the subsequence 010.

(e) Strings that do not contain the substring 010.

(f) Strings that do not contain the subsequence 010.

(g) Strings that contain an even number of occurrences of the substring 010.

⋆(h) Strings that contain an even number of occurrences of the substring 000.

(i) Strings in which every occurrence of the substring 00 appears before every occurrence of the substring 11.

(j) Strings w such that *in every prefix of w*, the number of 0s and the number of 1s differ by at most 1.

⋆(k) Strings w such that *in every prefix of w*, the number of 0s and the number of 1s differ by at most 2.

⋆(l) Strings in which the number of 0s and the number of 1s differ by a multiple of 3.

⋆(m) Strings that contain an even number of 1s and an odd number of 0s.

★(n) Strings that represent a number divisible by 5 in binary.

5. Prove that for any regular expression R such that $L(R)$ is nonempty, there is a regular expression equivalent to R that does not use the empty-set symbol $\varnothing$.

6. Prove that if L is a regular language, then L^R is also a regular language. *[Hint: How do you reverse a regular expression?]*

7. (a) Describe and analyze an efficient algorithm to determine, given a regular expression R, whether $L(R)$ is empty.

   (b) Describe and analyze an efficient algorithm to determine, given a regular expression R, whether $L(R)$ is infinite.

   In each problem, assume you are given R as a regular expression tree, not just a raw string.

Caveat lector! This is the first edition of this lecture note. A few topics are missing, and there are almost certainly a few serious errors. Please send bug reports and suggestions to jeffe@illinois.edu.

Life only avails, not the having lived. Power ceases in the instant of repose;
it resides in the moment of transition from a past to a new state,
in the shooting of the gulf, in the darting to an aim.

— Ralph Waldo Emerson, "Self Reliance", *Essays, First Series* (1841)

O Marvelous! what new configuration will come next?
I am bewildered with multiplicity.

— William Carlos Williams, "At Dawn" (1914)

3 Finite-State Machines

3.1 Intuition

Suppose we want to determine whether a given string $w[1..n]$ of bits represents a multiple of 5 in binary. After a bit of thought, you might realize that you can read the bits in w one at a time, from left to right, keeping track of the value modulo 5 of the prefix you have read so far.

```
MultipleOf5(w[1..n]):
  rem ← 0
  for i ← 1 to n
      rem ← (2 · rem + w[i]) mod 5
  if rem = 0
      return True
  else
      return False
```

Aside from the loop index i, which we need just to read the entire input string, this algorithm has a single local variable *rem*, which has only four different values (0, 1, 2, 3, or 4).

This algorithm already runs in $O(n)$ time, which is the best we can hope for—after all, we have to read every bit in the input—but we can speed up the algorithm *in practice*. Let's define a ***change*** or ***transition*** function $\delta\colon \{0,1,2,3,4\} \times \{\texttt{0},\texttt{1}\} \to \{0,1,2,3,4\}$ as follows:

$$\delta(q,a) = (2q + a) \bmod 5.$$

(Here I'm implicitly converting the symbols `0` and `1` to the corresponding integers 0 and 1.) Since we already know all values of the transition function, we can store them in a precomputed table, and then replace the computation in the main loop of MultipleOf5 with a simple array lookup.

We can also modify the return condition to check for different values modulo 5. To be completely general, we replace the final if-then-else lines with another array lookup, using an array $A[0..4]$ of booleans describing which final mod-5 values are "acceptable".

After both of these modifications, our algorithm can be rewritten as follows, either iteratively or recursively (with $q = 0$ in the initial call):

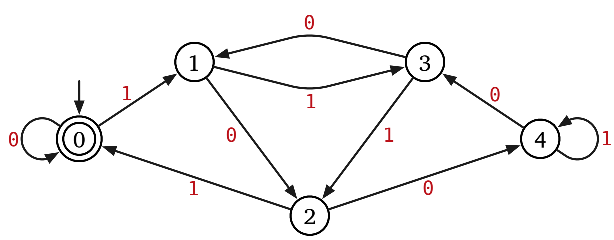
```
DoSomethingCool(w[1..n]):
  q ← 0
  for i ← 1 to n
      q ← δ[q, w[i]]
  return A[q]
```

```
DoSomethingCool(q, w):
  if w = ε
      return A[q]
  else
      decompose w = a · x
      return DoSomethingCool(δ(q, a), x)
```

If we want to use our new DoSomethingCool algorithm to implement MultipleOf5, we simply give the arrays δ and A the following hard-coded values:

q	$\delta[q, 0]$	$\delta[q, 1]$	$A[q]$
0	0	1	True
1	2	3	False
2	4	0	False
3	1	2	False
4	3	4	False

We can also visualize the behavior of DoSomethingCool by drawing a directed graph, whose vertices represent possible values of the variable q—the possible ***states*** of the algorithm—and whose edges are labeled with input symbols to represent transitions between states. Specifically, the graph includes the labeled directed edge $p \xrightarrow{a} q$ if and only if $\delta(p, a) = q$. To indicate the proper return value, we draw the "acceptable" final states using doubled circles. Here is the resulting graph for MultipleOf5:



State-transition graph for MultipleOf5

If we run the MultipleOf5 algorithm on the string 00101110110 (representing the number 374 in binary), the algorithm performs the following sequence of transitions:

$$0 \xrightarrow{0} 0 \xrightarrow{0} 0 \xrightarrow{1} 1 \xrightarrow{0} 2 \xrightarrow{1} 0 \xrightarrow{1} 1 \xrightarrow{1} 3 \xrightarrow{0} 1 \xrightarrow{1} 3 \xrightarrow{1} 2 \xrightarrow{0} 4$$

Because the final state is not the "acceptable" state 0, the algorithm correctly returns False. We can also think of this sequence of transitions as a walk in the graph, which is completely determined by the start state 0 and the sequence of edge labels; the algorithm returns True if and only if this walk ends at an "acceptable" state.

3.2 Formal Definitions

The object we have just described is an example of a ***finite-state machine***. A finite-state machine is a formal model of any system/machine/algorithm that can exist in a finite number of ***states*** and that transitions among those states based on sequence of ***input*** symbols.

Finite-state machines are also commonly called ***deterministic finite-state automata***, abbreviated ***DFAs***. The word "deterministic" means that the behavior of the machine is completely *determined* by the input string; we'll discuss nondeterministic automata in the next lecture. The word "automaton" (plural "automata") comes from ancient Greek $\alpha\upsilon\tau o\mu\alpha\tau o\varsigma$ meaning "self-acting", from the roots $\alpha\upsilon\tau o$- ("self") and -$\mu\alpha\tau o\varsigma$ ("thinking, willing", the root of Latin *mentus*).

Formally, every finite-state machine consists of five components:

- An arbitrary finite set $\mathbf{\Sigma}$, called the ***input alphabet***.
- Another arbitrary finite set $\boldsymbol{Q}$, whose elements are called ***states***.
- An arbitrary ***transition*** function $\boldsymbol{\delta}: Q \times \Sigma \to Q$.
- A ***start state*** $\boldsymbol{s} \in Q$.
- A subset $\boldsymbol{A} \subseteq Q$ of ***accepting states***.

The behavior of a finite-state machine is governed by an ***input string*** w, which is a finite sequence of symbols from the input alphabet Σ. The machine ***reads*** the symbols in w one at a time in order (from left to right). At all times, the machine has a *current state* q; initially q is the machine's start state s. Each time the machine reads a symbol a from the input string, its current state ***transitions*** from q to $\delta(q,a)$. After all the characters have been read, the machine ***accepts*** w if the current state is in A and ***rejects*** w otherwise. In other words, every finite state machine runs the algorithm DoSomethingCool! The ***language*** of a finite state machine M, denoted $\boldsymbol{L(M)}$ is the set of all strings that M accepts.

More formally, we extend the transition function $\delta: Q \times \Sigma^* \to Q$ of any finite-state machine to a function $\delta^*: Q \times \Sigma^* \to Q$ that transitions on *strings* as follows:

$$\delta^*(q,w) := \begin{cases} q & \text{if } w = \varepsilon, \\ \delta^*(\delta(q,a),x) & \text{if } w = ax. \end{cases}$$

Finally, a finite-state machine ***accepts*** a string w if and only if $\delta^*(s,w) \in A$, and ***rejects*** w otherwise. (Compare this definition with the recursive formulation of DoSomethingCool!)

For example, our final MultipleOf5 algorithm is a DFA with the following components:

- input alphabet: $\Sigma = \{\mathtt{0}, \mathtt{1}\}$
- state set: $Q = \{0,1,2,3,4\}$
- transition function: $\delta(q,a) = (2q + a) \bmod 5$
- start state: $s = 0$
- accepting states: $A = \{0\}$

This machine rejects the string `00101110110`, because

$$\begin{aligned}
\delta^*(0, \mathtt{00101110110}) &= \delta^*(\delta(0,\mathtt{0}), \mathtt{0101110110}) \\
&= \delta^*(0, \mathtt{0101110110}) = \delta^*(\delta(0,\mathtt{0}), \mathtt{101110110}) \\
&= \delta^*(0, \mathtt{101110110}) = \delta^*(\delta(0,\mathtt{1}), \mathtt{01110110}) = \cdots \\
&\vdots \\
\cdots &= \delta^*(1, \mathtt{110}) = \delta^*(\delta(1,\mathtt{1}), \mathtt{10}) \\
&= \delta^*(3, \mathtt{10}) = \delta^*(\delta(3,\mathtt{1}), \mathtt{0}) \\
&= \delta^*(2, \mathtt{0}) = \delta^*(\delta(3,\mathtt{0}), \varepsilon) \\
&= \delta^*(4, \varepsilon) = 4 \notin A.
\end{aligned}$$

We have already seen a more graphical representation of this entire sequence of transitions:

$$0 \xrightarrow{\texttt{0}} 0 \xrightarrow{\texttt{0}} 0 \xrightarrow{\texttt{1}} 1 \xrightarrow{\texttt{0}} 2 \xrightarrow{\texttt{1}} 0 \xrightarrow{\texttt{1}} 1 \xrightarrow{\texttt{1}} 3 \xrightarrow{\texttt{0}} 1 \xrightarrow{\texttt{1}} 3 \xrightarrow{\texttt{1}} 2 \xrightarrow{\texttt{0}} 4$$

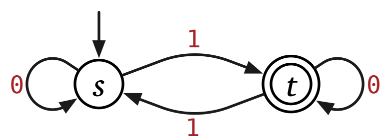
The arrow notation is easier to read and write for specific examples, but surprisingly, most people actually find the more formal functional notation easier to use in formal proofs. Try them both!

We can equivalently define a DFA as a directed graph whose vertices are the states Q, whose edges are labeled with symbols from Σ, such that every vertex has exactly one outgoing edge with each label. In our drawings of finite state machines, the start state s is always indicated by an incoming arrow, and the accepting states A are always indicted by doubled circles. By induction, for any string $w \in \Sigma^*$, this graph contains a unique walk that starts at s and whose edges are labeled with the symbols in w in order. The machine accepts w if this walk ends at an accepting state. This graphical formulation of DFAs is incredibly useful for developing intuition and even designing DFAs. For proofs, it's largely a matter of taste whether to write in terms of extended transition functions or labeled graphs, but (as much as I wish otherwise) I actually find it easier to write ***correct*** proofs using the functional formulation.

3.3 Another Example

The following drawing shows a finite-state machine with input alphabet $\Sigma = \{\texttt{0}, \texttt{1}\}$, state set $Q = \{s, t\}$, start state s, a single accepting state t, and the transition function

$$\delta(s, \texttt{0}) = s, \quad \delta(s, \texttt{1}) = t, \quad \delta(t, \texttt{0}) = t, \quad \delta(t, \texttt{1}) = s.$$



A simple finite-state machine.

For example, the two-state machine M at the top of this page accepts the string `00101110100` after the following sequence of transitions:

$$s \xrightarrow{\texttt{0}} s \xrightarrow{\texttt{0}} s \xrightarrow{\texttt{1}} t \xrightarrow{\texttt{0}} t \xrightarrow{\texttt{1}} s \xrightarrow{\texttt{1}} t \xrightarrow{\texttt{1}} s \xrightarrow{\texttt{0}} s \xrightarrow{\texttt{1}} t \xrightarrow{\texttt{0}} t \xrightarrow{\texttt{0}} t.$$

The same machine M rejects the string `11100101` after the following sequence of transitions:

$$s \xrightarrow{\texttt{1}} t \xrightarrow{\texttt{1}} s \xrightarrow{\texttt{1}} t \xrightarrow{\texttt{0}} t \xrightarrow{\texttt{0}} s \xrightarrow{\texttt{1}} t \xrightarrow{\texttt{0}} t \xrightarrow{\texttt{1}} s.$$

Finally, M rejects the empty string, because the start state s is not an accepting state.

From these examples and others, it is easy to conjecture that the language of M is the set of all strings of `0`s and `1`s with an odd number of `1`s. So let's prove it!

Proof (tedious case analysis): Let $\#(a, w)$ denote the number of times symbol a appears in string w. We will prove the following stronger claims, for any string w.

$$\delta^*(s, w) = \begin{cases} s & \text{if } \#(\texttt{1}, w) \text{ is even} \\ t & \text{if } \#(\texttt{1}, w) \text{ is odd} \end{cases} \qquad \text{and} \qquad \delta^*(t, w) = \begin{cases} t & \text{if } \#(\texttt{1}, w) \text{ is even} \\ s & \text{if } \#(\texttt{1}, w) \text{ is odd} \end{cases}$$

Let w be an arbitrary string. Assume that for any string x that is shorter than w, we have $\delta^*(s, x) = s$ and $\delta^*(t, x) = t$ if x has an even number of `1`s, and $\delta^*(s, x) = t$ and $\delta^*(t, x) = s$ if x has an odd number of `1`s. There are five cases to consider.

- If $w = \varepsilon$, then w contains an even number of 1s and $\delta^*(s, w) = s$ and $\delta^*(t, w) = t$ by definition.

- Suppose $w = 1x$ and $\#(1, w)$ is even. Then $\#(1, x)$ is odd, which implies

$$\begin{aligned}
\delta^*(s,w) &= \delta^*(\delta(s,1),x) && \text{by definition of } \delta^* \\
&= \delta^*(t,x) && \text{by definition of } \delta \\
&= s && \text{by the inductive hypothesis}
\end{aligned}$$

$$\begin{aligned}
\delta^*(t,w) &= \delta^*(\delta(t,1),x) && \text{by definition of } \delta^* \\
&= \delta^*(s,x) && \text{by definition of } \delta \\
&= T && \text{by the inductive hypothesis}
\end{aligned}$$

  Since the remaining cases are similar, I'll omit the line-by-line justification.

- If $w = 1x$ and $\#(1, w)$ is odd, then $\#(1, x)$ is even, so the inductive hypothesis implies

$$\begin{aligned}
\delta^*(s,w) &= \delta^*(\delta(s,1),x) = \delta^*(t,x) = t \\
\delta^*(t,w) &= \delta^*(\delta(t,1),x) = \delta^*(s,x) = s
\end{aligned}$$

- If $w = 0x$ and $\#(1, w)$ is even, then $\#(1, x)$ is even, so the inductive hypothesis implies

$$\begin{aligned}
\delta^*(s,w) &= \delta^*(\delta(s,0),x) = \delta^*(s,x) = s \\
\delta^*(t,w) &= \delta^*(\delta(t,0),x) = \delta^*(t,x) = t
\end{aligned}$$

- Finally, if $w = 0x$ and $\#(1, w)$ is odd, then $\#(1, x)$ is odd, so the inductive hypothesis implies

$$\begin{aligned}
\delta^*(s,w) &= \delta^*(\delta(s,0),x) = \delta^*(s,x) = t \\
\delta^*(t,w) &= \delta^*(\delta(t,0),x) = \delta^*(t,x) = s
\end{aligned}$$ □

Notice that this proof contains $|Q|^2 \cdot |\Sigma| + |Q|$ separate inductive arguments. For every pair of states p and q, we must argue about the language so strings w such that $\delta^*(p, w) = q$, and we must consider each first symbol in w. We must also argue about $\delta(p, \varepsilon)$ for every state p. Each of those arguments is typically straightforward, but it's easy to get lost in the deluge of cases.

For this particular proof, however, we can reduce the number of cases by switching from tail recursion to *head* recursion. The following identity holds for all strings $x \in \Sigma^*$ and symbols $a \in \Sigma$:

$$\boxed{\delta^*(q, xa) = \delta(\delta^*(q, x), a)}$$

We leave the inductive proof of this identity as a straightforward exercise (hint, hint).

Proof (clever renaming, head induction): Let's rename the states 0 and 1 instead of s and t. Then the transition function can be described concisely as $\boldsymbol{\delta(q, a) = (q + a) \bmod 2}$.

Now we claim that for every string w, we have $\delta^*(0, w) = \#(1, w) \bmod 2$. So let w be an arbitrary string, and assume that for any string x that is shorter than w that $\delta^*(0, x) = \#(1, x) \bmod 2$. There are only two cases to consider: either w is empty or it isn't.

- If $w = \varepsilon$, then $\delta^*(0, w) = 0 = \#(1, w) \bmod 2$ by definition.

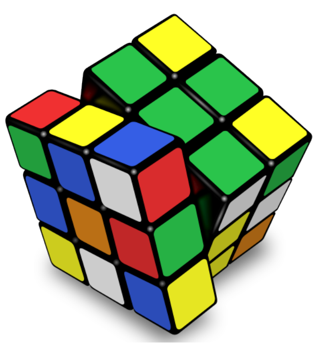
- Otherwise, $w = xa$ for some string x and some symbol a, and we have

$$\begin{aligned}
\delta^*(0,w) &= \delta(\delta^*(0,x),a) \\
&= \delta(\#(\mathtt{1},x) \bmod 2, a) && \text{by the inductive hypothesis} \\
&= (\#(\mathtt{1},x) \bmod 2 + a) \bmod 2 && \text{by definition of } \delta \\
&= (\#(\mathtt{1},x) + a) \bmod 2 && \text{by definition of } \bmod 2 \\
&= (\#(\mathtt{1},x) + \#(\mathtt{1},a)) \bmod 2 && \text{because } \#(\mathtt{1},\mathtt{0}) = 0 \text{ and } \#(\mathtt{1},\mathtt{1}) = 1 \\
&= (\#(\mathtt{1},xa)) \bmod 2 && \text{by definition of } \# \\
&= (\#(\mathtt{1},w)) \bmod 2 && \text{because } w = xa \qquad \square
\end{aligned}$$

Hmmm. This "clever" proof is certainly shorter than the earlier brute-force proof, but is it really "better"? "Simpler"? More intuitive? Easier to understand? I'm skeptical. Sometimes brute force really is more effective.

3.4 Yet Another Example

As a more complex example, consider the ***Rubik's cube***, a well-known mechanical puzzle invented independently by Ern Rubik in Hungary and Terutoshi Ishigi in Japan in the mid-1970s. This puzzle has precisely 519,024,039,293,878,272,000 distinct configurations. In the unique *solved* configuration, each of the six faces of the cube shows exactly one color. We can change the configuration of the cube by rotating one of the six faces of the cube by 90 degrees, either clockwise or counterclockwise. The cube has six faces (front, back, left, right, up, and down), so there are exactly twelve possible turns, typically represented by the symbols $\mathtt{R}, \mathtt{L}, \mathtt{F}, \mathtt{B}, \mathtt{U}, \mathtt{D}, \bar{\mathtt{R}}, \bar{\mathtt{L}}, \bar{\mathtt{F}}, \bar{\mathtt{B}}, \bar{\mathtt{U}}, \bar{\mathtt{D}}$, where the letter indicates which face to turn and the presence or absence of a bar over the letter indicates turning counterclockwise or clockwise, respectively. Thus, we can represent a Rubik's cube as a finite-state machine with 519,024,039,293,878,272,000 states and an input alphabet with 12 symbols; or equivalently, as a directed graph with 519,024,039,293,878,272,000 vertices, each with 12 outgoing edges. In practice, the number of states is *far* too large for us to actually draw the machine or explicitly specify its transition function; nevertheless, the number of states is still finite. If we let the start state s and the sole accepting state be the solved state, then the language of this finite state machine is the set of all move sequences that leave the cube unchanged.



A complicated finite-state machine.

3.5 Building DFAs

This section describes a few examples of building DFAs that accept particular languages, thereby proving that those languages are automatic. As usual in algorithm design, there is no purely mechanical recipe—no *automatic* method—no *algorithm*—for building DFAs in general. However, the following examples show several useful design strategies.

3.5.1 Superstrings

Perhaps the simplest rule of thumb is to try to construct an algorithm that looks like MULTIPLEOF5: A simple for-loop through the symbols, using a *constant* number of variables, where each variable (except the loop index) has only a *constant* number of possible values. Here, "constant" means an actual number that is not a function of the input size n. You should be able to compute the number of possible values for each variable *at compile time*.

For example, the following algorithm determines whether a given string in $\Sigma = \{0, 1\}$ contains the substring 11.

```
CONTAINS11(w[1..n]):
  found ← FALSE
  for i ← 1 to n
      if i = 1
          last2 ← w[1]
      else
          last2 ← w[1] · w[2]
      if last = 11
          found ← TRUE
  return found
```

Aside from the loop index, this algorithm has exactly two variables.

- A boolean flag *found* indicating whether we have seen the substring 11. This variable has exactly two possible values: TRUE and FALSE.
- A string *last2* containing the last (up to) three symbols we have read so far. This variable has exactly 7 possible values: ε, 0, 1, 00, 01, 10, and 11.

Thus, altogether, the algorithm can be in at most $2 \times 7 = 14$ possible states, one for each possible pair (*found*, *last2*). Thus, we can encode the behavior of CONTAINS11 as a DFA with fourteen states, where the start state is (FALSE, ε) and the accepting states are all seven states of the form (TRUE, $*$). The transition function is described in the following table (split into two parts to save space):

q	$\delta[q, 0]$	$\delta[q, 1]$
(FALSE, ε)	(FALSE, 0)	(FALSE, 1)
(FALSE, 0)	(FALSE, 00)	(FALSE, 01)
(FALSE, 1)	(FALSE, 10)	(**TRUE**, 11)
(FALSE, 00)	(FALSE, 00)	(FALSE, 01)
(FALSE, 01)	(FALSE, 10)	(**TRUE**, 11)
(FALSE, 10)	(FALSE, 00)	(FALSE, 01)
(FALSE, 11)	(FALSE, 10)	(**TRUE**, 11)

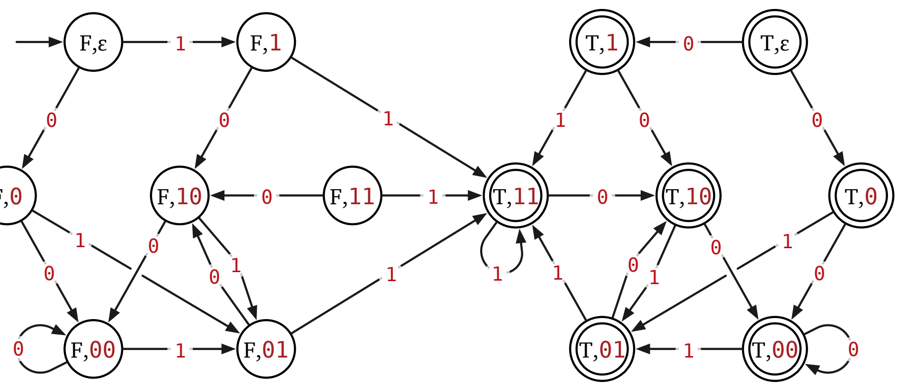
q	$\delta[q, 0]$	$\delta[q, 1]$
(TRUE, ε)	(TRUE, 0)	(TRUE, 1)
(TRUE, 0)	(TRUE, 00)	(TRUE, 01)
(TRUE, 1)	(TRUE, 10)	(TRUE, 11)
(TRUE, 00)	(TRUE, 00)	(TRUE, 01)
(TRUE, 01)	(TRUE, 10)	(TRUE, 11)
(TRUE, 10)	(TRUE, 00)	(TRUE, 01)
(TRUE, 11)	(TRUE, 10)	(TRUE, 11)

For example, given the input string 1001011100, this DFA performs the following sequence of transitions and then accepts.

$$(\textsc{False}, \varepsilon) \xrightarrow{1} (\textsc{False}, 1) \xrightarrow{0} (\textsc{False}, 10) \xrightarrow{0} (\textsc{False}, 00) \xrightarrow{1}$$
$$(\textsc{False}, 01) \xrightarrow{0} (\textsc{False}, 10) \xrightarrow{1} (\textsc{False}, 01) \xrightarrow{1}$$
$$(\textsc{True}, 11) \xrightarrow{1} (\textsc{True}, 11) \xrightarrow{0} (\textsc{True}, 10) \xrightarrow{0} (\textsc{True}, 00)$$

3.5.2 Reducing states

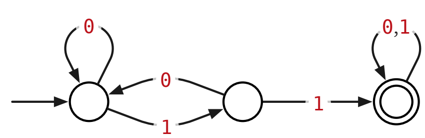
You can probably guess that the brute-force DFA we just constructed has considerably more states than necessary, especially after seeing its transition graph:



Our brute-force DFA for strings containing the substring 11

For example, we don't need actually to remember both of the last two symbols, but only the penultimate symbol, because the last symbol is the one we're currently reading. This observation allows us to reduce the number of states from fourteen to only six. Once the flag part of the state is set to TRUE, we know the machine will eventually accept, so we might as well merge the two accepting states together. Finally, and more subtly, because all transitions out of $(\textsc{False}, \varepsilon)$ and $(\textsc{False}, 0)$ are identical, we can merge those two states together as well. In the end, we obtain the following DFA with just three states:

- The start state, which indicates that the machine has not read the substring 11 an did not just read the symbol 1.
- An intermediate state, which indicates that the machine has not read the substring 11 but just read the symbol 1.
- A unique accept state, which indicates that the machine has read the substring 11.



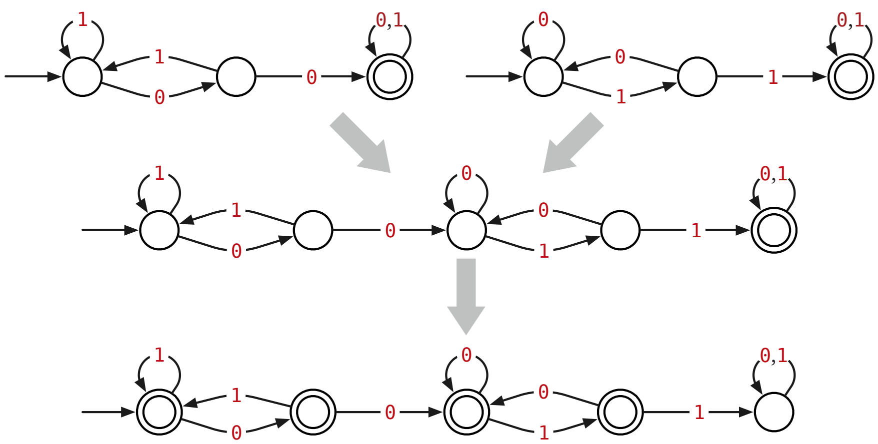
A minimal DFA for superstrings of 11

At the end of this note, I'll describe an efficient algorithm to transform any given DFA into an equivalent DFA with the fewest possible states. Given that this minimization algorithm exists, there is very little incentive to optimize DFAs *by hand*. Clarity is infinitely more important than brevity, especially in this class.

3.5.3 Every this after that

Suppose we want to accept the set of strings in which every occurrence of the substring `00` occurs after every occurrence of the substring `11`. Equivalently, we want to *reject* every string in which some `00` occurs before `11`. Often the easiest way to design a DFA to check whether a string is *not* in some set is first to build a DFA that *is* in that set and then invert which states in that machine are accepting.

From the previous example, we know that there is a three-state DFA M_{11} that accepts the set of strings with the substring `11` and a nearly identical DFA M_{00} that accepts the set of strings containing the substring `00`. By identifying the accept state of M_{00} with the start state of M_{11}, we obtain a five-state DFA that accepts the set of strings with `00` before `11`. Finally, by inverting which states are accepting, we obtain the DFA we want.



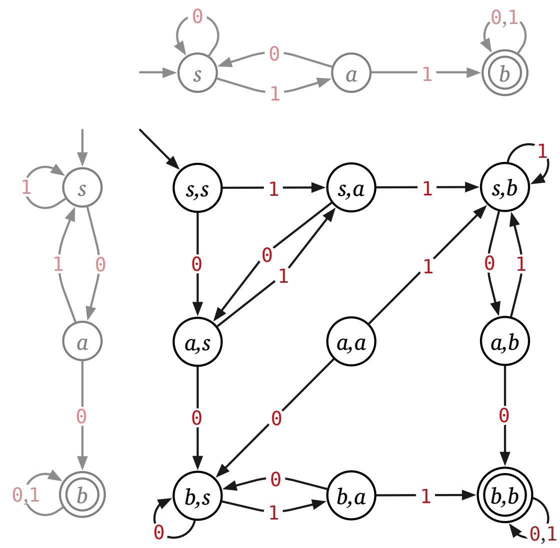
Building a DFA for the language of strings in which every `00` is after every `11`.

3.5.4 Both This and That: The Product Construction

Now suppose we want to accept all strings that contain both `00` and `11` as substrings, in either order. Intuitively, we'd like to run two of our earlier DFAs in parallel—the DFA M_{00} to detect superstrings of `00` and the DFA M_{11} to detect superstrings of `11`—and then accept the input string if and only if both of these DFAs accept. In fact, we can encode precisely this "parallel computation" into a single DFA, whose states are all ordered pairs (p,q), where p is a state in M_{00} and q is a state in M_{11}. The new "parallel" DFA includes the transition $(p,q) \xrightarrow{a} (p',q')$ if and only if M_{00} contains the transition $p \xrightarrow{a} p'$ and M_{11} contains the transition $q \xrightarrow{a} q'$. Finally, the state (p,q) is accepting if and only if p and q are accepting states in their respective machines. The resulting nine-state DFA is shown on the next page.

More generally, let $M_1 = (\Sigma, Q_1, \delta_1, s_1, A_1)$ be an arbitrary DFA that accepts some language L_1, and let $M_2 = (\Sigma, Q_2, \delta_2, s_2, A_2)$ be an arbitrary DFA that accepts some language L_2 (over the same alphabet Σ). We can construct a third DFA $M = (\Sigma, Q, \delta, s, A)$ that accepts the intersection language $L_1 \cap L_2$ as follows.

$$
\begin{aligned}
Q &:= Q_1 \times Q_2 = \big\{(p,q) \bigm| p \in Q_1 \text{ and } q \in Q_2\big\} \\
s &:= (s_1, s_2) \\
A &:= A_1 \times A_2 = \big\{(p,q) \bigm| p \in A_1 \text{ and } q \in A_2\big\} \\
\delta((p,q),a) &:= \big(\delta_1(p,a),\ \delta_2(q,a)\big)
\end{aligned}
$$



Building a DFA for the language of strings in which every `00` is after every `11`.

To convince yourself that this product construction is actually correct, consider the extended transition function $\delta^*\colon (Q \times Q') \times \Sigma^* \to (Q \times Q')$, which acts on strings instead of individual symbols. Recall that this function is defined recursively as follows:

$$\delta^*\big((p,q),w\big) := \begin{cases} q & \text{if } w = \varepsilon, \\ \delta^*\big(\delta((p,q),a),\ x\big) & \text{if } w = ax. \end{cases}$$

Inductive definition-chasing gives us the identity $\delta^*((p,q),w) = \big(\delta_1^*(p,w),\ \delta_2^*(q,w)\big)$ for any string w:

$$\begin{aligned} \delta^*\big((p,q),\varepsilon\big) &= (p,q) && \text{by the definition of } \delta^* \\ &= \big(\delta_1^*(p,\varepsilon),\ \delta_2^*(q,\varepsilon)\big) && \text{by the definitions of } \delta_1^* \text{ and } \delta_2^*; \\ \delta^*\big((p,q),ax\big) &= \delta^*\big(\delta((p,q),a),\ x\big) && \text{by the definition of } \delta^* \\ &= \delta^*\big((\delta_1(p,a),\ \delta_2(q,a)),\ x\big) && \text{by the definition of } \delta \\ &= \big(\delta_1^*((\delta_1(p,a),\ x),\ \delta_2^*(\delta_2(q,a),x)\big) && \text{by the induction hypothesis} \\ &= \big(\delta_1^*(p,ax),\ \delta_2^*(q,ax)\big) && \text{by the definitions of } \delta_1^* \text{ and } \delta_2^*. \end{aligned}$$

It now follows from this seemingly impenetrable wall of notation that for any string w, we have $\delta^*(s,w) \in A$ if and only if both $\delta_1^*(s_1,w) \in A_1$ and $\delta_2^*(s_2,w) \in A_2$. In other words, M accepts w if and only if both M_1 and M_2 accept w, as required.

As usual, this construction technique does not necessarily yield *minimal* DFAs. For example, in our first example of a product DFA, illustrated above, the central state (a,a) cannot be reached by any other state and is therefore redundant. Whatever.

Similar product constructions can be used to build DFAs that accept any other boolean combination of languages; in fact, the only part of the construction that needs to be changed is the choice of accepting states. For example:

- To accept the union $L_1 \cup L_2$, define $A = \big\{(p,q) \bigm| p \in A_1 \textbf{ or } q \in A_2\big\}$.

- To accept the difference $L_1 \setminus L_2$, define $A = \{(p,q) \mid p \in A_1 \textbf{ but not } q \notin A_2\}$.
- To accept the symmetric difference $L_1 \oplus L_2$, define $A = \{(p,q) \mid p \in A_1 \textbf{ xor } q \in A_2\}$.

Moreover, by cascading this product construction, we can construct DFAs that accept arbitrary boolean combinations of arbitrary finite collections of regular languages.

3.6 Decision Algorithms

★★★

It's unclear how much we can say here, since we haven't yet talked about graph algorithms, or even really about graphs. Perhaps this discussion should simply be moved to the graph-traversal notes.

- ***Is $w \in L(M)$?*** Follow the unique path from q_0 with label w. By definition, $w \in L(M)$ if and only if this path leads to an accepting state.
- ***Is $L(M)$ empty?*** The language $L(M)$ is empty if and only if no accepting state is reachable from q_0. This condition can be checked in $O(n)$ time via whatever-first search, where n is the number of states. Alternatively, but less usefully, $L(M) = \varnothing$ if and only if $L(M)$ contains no string w such that $|w| < n$.
- ***Is $L(M)$ finite?*** Remove all states unreachable from q_0 (via whatever first search). Then $L(M)$ is finite if and only if the reduced DFA is a dag; this condition can be checked by depth-first search. Alternatively, but less usefully, $L(M)$ is finite if and only if $L(M)$ contains no string w such that $n \le |w| < 2n$.
- ***Is $L(M) = \Sigma^*$?*** Remove all states unreachable from q_0 (via whatever first search). Then $L(M) = \Sigma^*$ if and only if every state in M is an accepting state.
- ***Is $L(M) = L(M')$?*** Build a DFA N such that $L(N) = L(M) \setminus L(M')$ using a standard product construction, and then check whether $L(N) = \varnothing$.

3.7 Closure Properties

★★★

We haven't yet proved that automatic languages are regular yet, so formally, for now, some of these are closure properties of ***automatic*** languages.

- Complement (easy for DFAs, hard for regular expressions.)
- Concatenation (trivial for regular expressions, hard for DFAs)
- Union (trivial for regular expressions, easy for DFAs via product)
- Intersection (hard for regular expressions, easy for DFAs via product)
- Difference (hard for regular expressions, easy for DFAs via product)
- Kleene star: wait for NFAs (trivial for regular expression, hard for DFAs)
- Homomorphism: only mention in passing
- Inverse homomorphism: only mention in passing

3.8 Fooling Sets

Fix an arbitrary language L over an arbitrary alphabet Σ. For any strings $x, y, z \in \Sigma^*$, we say that ***z distinguishes x from y*** if exactly one of the strings xz and yz is in L. If no string distinguishes x and y, we say that x and y are ***L-equivalent*** and write $\boldsymbol{x \equiv_L y}$. Thus,

$$x \equiv_L y \iff \text{For every string } z \in \Sigma^*, \text{ we have } xz \in L \text{ if and only if } yz \in L.$$

For example, let L_{eo} denote the language of strings over $\{0,1\}$ with an even number of 0s and an odd number of 1s. Then the strings $x = 01$ and $y = 0011$ are distinguished by the string $z = 100$, because

$$xz = 01 \bullet 100 = 01100 \in L_{eo}$$
$$yz = 0011 \bullet 100 = 0011100 \notin L_{eo}.$$

On the other hand, it is quite easy to prove (hint, hint) that the strings 0001 and 1011 are L_{eo}-equivalent.

Let M be an arbitrary DFA for an arbitrary language L, and let x be y be arbitrary strings. If x and y lead to the same state in M—that is, if $\delta^*(s,x) = \delta^*(s,y)$—then we have

$$\delta^*(s,xz) = \delta^*(\delta^*(s,x),z) = \delta^*(\delta^*(s,y),z) = \delta^*(s,yz)$$

for any string z. In particular, either M accepts both x and y, or M rejects both x and y, and therefore $x \equiv_L y$. It follows that if x and y are not L-equivalent, then ***any*** DFA that accepts L has at least two distinct states $\delta^*(s,x) \neq \delta^*(s,y)$.

Finally, a ***fooling set*** for L is a set F of strings such that *every* pair of strings in F has a distinguishing suffix. For example, $F = \{01, 101, 010, 1010\}$ is a fooling set for the language L_{eo} of strings with an even number of 0s and an odd number of 1s, because each pair of strings in F has a distinguishing suffix:

- 0 distinguishes 01 and 101;
- 0 distinguishes 01 and 010;
- 0 distinguishes 01 and 1010;
- 10 distinguishes 101 and 010;
- 1 distinguishes 101 and 1010;
- 1 distinguishes 010 and 1010.

The pigeonhole principle now implies that for any integer k, if language L is accepted by a DFA with k states, then *every* fooling set for L contains at most k strings. This simple observation has two immediate corollaries.

First, for any integer k, if L has a fooling set of size k, then *every* DFA that accepts L has at least k states. For example, the fooling set $\{01, 101, 010, 1010\}$ proves that any DFA for L_{eo} has at least four states. Thus, we can use fooling sets to prove that certain DFAs are as small as possible.

Second, and more interestingly, if a language L is accepted by *any* DFA, then *every* fooling set for L must be finite. Equivalently:

If L has an infinite fooling set, then L is not accepted by any DFA.

This is arguably both the simplest and most powerful method for proving that a language is non-regular. Here are a few canonical examples of the fooling-set technique in action.

Lemma 3.1. *The language $L = \{0^n 1^n \mid n \geq 0\}$ is not regular.*

Proof: Consider the set $F = \{0^n \mid n \geq 0\}$, or more simply $F = 0^*$. Let x and y be arbitrary distinct strings in F. Then we must have $x = 0^i$ and $y = 0^j$ for some integers $i \neq j$. The suffix $z = 1^i$ distinguishes x and y, because $xz = 0^i 1^i \in L$, but $yz = 0^i 1^j \notin L$. We conclude that F is a fooling set for L. Because F is infinite, L cannot be regular. □

Lemma 3.2. *The language $L = \{ww^R \mid w \in \Sigma^*\}$ of even-length palindromes is not regular.*

Proof: Let x and y be arbitrary distinct strings in $\mathtt{0}^*\mathtt{1}$. Then we must have $x = \mathtt{0}^i\mathtt{1}$ and $y = \mathtt{0}^j\mathtt{1}$ for some integers $i \neq j$. The suffix $z = \mathtt{10}^i$ distinguishes x and y, because $xz = \mathtt{0}^i\mathtt{110}^i \in L$, but $yz = \mathtt{0}^i\mathtt{110}^j \notin L$. We conclude that $\mathtt{0}^*\mathtt{1}$ is a fooling set for L. Because $\mathtt{0}^*\mathtt{1}$ is infinite, L cannot be regular. □

Lemma 3.3. *The language $L = \{\mathtt{0}^{2^n} \mid n \geq 0\}$ is not regular.*

Proof: Let x and y be arbitrary distinct strings in L. Then we must have $x = \mathtt{0}^{2^i}$ and $y = \mathtt{0}^{2^j}$ for some integers $i \neq j$. The suffix $z = \mathtt{0}^{2^i}$ distinguishes x and y, because $xz = \mathtt{0}^{2^i+2^i} = \mathtt{0}^{2^{i+1}} \in L$, but $yz = \mathtt{0}^{2^i+2^j} \notin L$. We conclude that L itself is a fooling set for L. Because L is infinite, L cannot be regular. □

Lemma 3.4. *The language $L = \{\mathtt{0}^p \mid p \text{ is prime}\}$ is not regular.*

Proof: Again, we use $\mathtt{0}^*$ as our fooling set, but but the actual argument is somewhat more complicated than in our earlier examples.

Let x and y be arbitrary distinct strings in $\mathtt{0}^*$. Then we must have $x = \mathtt{0}^i$ and $y = \mathtt{0}^j$ for some integers $i \neq j$. Without loss of generality, assume that $i < j$. Let p be any prime number larger than i. Because $p + 0(j - i)$ is prime and $p + p(j - i) > p$ is not, there must be a positive integer $k \leq p$ such that $p + (k-1)(j-i)$ is prime but $p + k(j-i)$ is not. Then the suffix $\mathtt{0}^{p+(k-1)j-ki}$ distinguishes x and y:

$$xz = \mathtt{0}^i\,\mathtt{0}^{p+(k-1)j-ki} = \mathtt{0}^{p+(k-1)(j-i)} \in L \qquad \text{because } p+(k-1)(j-i) \text{ is prime;}$$
$$yz = \mathtt{0}^j\,\mathtt{0}^{p+(k-1)j-ki} = \mathtt{0}^{p+k(j-i)} \notin L \qquad \text{because } p+k(j-i) \text{ is not prime.}$$

(Because $i < j$ and $i < p$, the suffix $\mathtt{0}^{p+(k-1)j-ki} = \mathtt{0}^{(p-i)+(k-1)(j-i)}$ has positive length and therefore *actually exists!*) We conclude that $\mathtt{0}^*$ is indeed a fooling set for L, which implies that L is not regular. □

One natural question that many students ask is "How did you come up with that fooling set?" Perhaps the simplest rule of thumb is that for most languages L—in particular, for almost all languages that students are asked to prove non-regular on homeworks or exams—either some simple regular language like $\mathtt{0}^*$ or $\mathtt{10}^*\mathtt{1}$ is a fooling set, or the language L itself is a fooling set. (Of course, there are well-engineered counterexamples.)

*3.9 The Myhill-Nerode Theorem

The fooling set technique implies a *necessary* condition for a language to be accepted by a DFA—the language must have no infinite fooling sets. In fact, this condition is also *sufficient*. The following powerful theorem was first proved by Anil Nerode in 1958, strengthening a 1957 result of John Myhill.[1]

The Myhill-Nerode Theorem. *For any language L, the following are equal:*

[1]Myhill considered the finer equivalence relation $x \sim_L y$, meaning $wxz \in L$ if and only if $wyz \in L$ for all strings w and z, and proved that L is regular if and only if $\sim_L$ defines a finite number of equivalence classes. Like most of Myhill's early automata research, this result appears in an unpublished Air Force technical report. The modern Myhill-Nerode theorem appears (in an even more general form) as a minor lemma in Nerode's 1958 paper, which (not surprisingly) does not cite Myhill.

(a) the minimum number of states in a DFA that accepts L,
(b) the maximum size of a fooling set for L, and
(c) the number of equivalence classes of $\equiv_L$.
In particular, L is accepted by a DFA if and only if every fooling set for L is finite.

Proof: Let L be an arbitrary language.

We have already proved that the size of any fooling set for L is at most the number of states in any DFA that accepts L, so (a)≤(b). It also follows directly from the definitions that $F \subseteq \Sigma^*$ is a fooling set for L if and only if F contains at most one string in each equivalence class of $\equiv_L$; thus, (b)=(c). We complete the proof by showing that (a)≥(c).

We have already proved that if $\equiv_L$ has an infinite number of equivalence classes, there is no DFA that accepts L, so assume that the number of equivalence classes is finite. For any string w, let $[w]$ denote its equivalence class. We define a DFA $M_\equiv = (\Sigma, Q, s, A, \delta)$ as follows:

$$\begin{aligned}
Q &:= \big\{[w] \bigm| w \in \Sigma^*\big\}\\
s &:= [\varepsilon]\\
A &:= \big\{[w] \bigm| w \in L\big\}\\
\delta([w], a) &:= [w \bullet a]
\end{aligned}$$

We claim that this DFA accepts the language L; this claim completes the proof of the theorem.

But before we can prove anything about this DFA, we first need to verify that it is actually well-defined. Let x and y be two strings such that $[x] = [y]$. By definition of L-equivalence, for any string z, we have $xz \in L$ if and only if $yz \in L$. It immediately follows that for any symbol $a \in \Sigma$ and any string z', we have $xaz' \in L$ if and only if $yaz' \in L$. Thus, by definition of L-equivalence, we have $[xa] = [ya]$ for every symbol $a \in \Sigma$. We conclude that the function δ is indeed well-defined.

An easy inductive proof implies that $\delta^*([\varepsilon], x) = [x]$ for every string x. Thus, M accepts string x if and only if $[x] = [w]$ for some string $w \in L$. But if $[x] = [w]$, then by definition (setting $z = \varepsilon$), we have $x \in L$ if and only if $w \in L$. So M accepts x if and only if $x \in L$. In other words, M accepts L, as claimed, so the proof is complete. □

⋆3.10 Minimal Automata

Given a DFA $M = (\Sigma, Q, s, A, \delta)$, suppose we want to find another DFA $M' = (\Sigma, Q', s', A', \delta')$ with the fewest possible states that accepts the same language. In this final section, we describe an efficient algorithm to minimize DFAs, first described (in slightly different form) by Edward Moore in 1956. We analyze the running time of Moore's in terms of two parameters: $n = |Q|$ and $\sigma = |\Sigma|$.

In the preprocessing phase, we find and remove any states that cannot be reached from the start state s; this filtering can be performed in $O(n\sigma)$ time using any graph traversal algorithm. So from now on we assume that all states are reachable from s.

Now define two states p and q in the trimmed DFA to be ***distingusiable***, written $\boldsymbol{p \not\sim q}$, if at least one of the following conditions holds:

- $p \in A$ and $q \notin A$,
- $p \notin A$ and $q \in A$, or
- $\delta(p, a) \not\sim \delta(q, a)$ for some $a \in \Sigma$.

Equivalently, $p \not\sim q$ if and only if there is a string z such that exactly one of the states $\delta^*(p,z)$ and $\delta^*(q,z)$ is accepting. (Sound familiar?) Intuitively, the main algorithm assumes that all states are equivalent until proven otherwise, and then repeatedly looks for state pairs that can be proved distinguishable.

The main algorithm maintains a two-dimensional table, indexed by the states, where $Dist[p,q] = \text{True}$ indicates that we have proved states p and q are distinguishable. Initially, for all states p and q, we set $Dist[p,q] \leftarrow \text{True}$ if $p \in A$ and $q \notin A$ or vice versa, and $Dist[p,q] = \text{False}$ otherwise. Then we repeatedly consider each pair of states and each symbol to find more distinguished pairs, until we make a complete pass through the table without modifying it. The table-filling algorithm can be summarized as follows:[2]

```
MinDFATable(Σ, Q, s, A, δ):
  for all p ∈ Q
      for all q ∈ Q
          if (p ∈ A and q ∉ A) or (p ∉ A and q ∈ A)
              Dist[p, q] ← True
          else
              Dist[p, q] ← False
  notdone ← True
  while notdone
      notdone ← False
      for all p ∈ Q
          for all q ∈ Q
              if Dist[p, q] = False
              for all a ∈ Σ
                  if Dist[δ(p, a), δ(q, a)]
                      Dist[p, q] ← True
                      notdone ← True
  return Dist
```

The algorithm must eventually halt, because there are only a finite number of entries in the table that can be marked. In fact, the main loop is guaranteed to terminate after at most n iterations, which implies that the entire algorithm runs in $\boldsymbol{O(\sigma n^3)}$ time. Once the table is filled, any two states p and q such that $Dist(p,q) = \text{False}$ are equivalent and can be merged into a single state. The remaining details of constructing the minimized DFA are straightforward.

★★★ Need to prove that the main loop terminates in at most n iterations.

With more care, Moore's minimization algorithm can be modified to run in $O(\sigma n^2)$ time. A faster DFA minimization algorithm, due to John Hopcroft, runs in $O(\sigma n \log n)$ time.

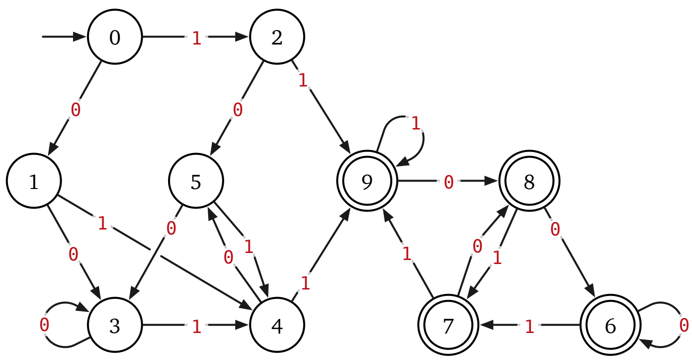
[2]More experienced readers should become queasy at the mere suggestion that any algorithm merely *fills in a table*, as opposed to *evaluating a recurrence*. This algorithm is no exception. Consider the boolean function $Dist(p,q,k)$, which equals True if and only if p and q can be distinguished by some string of length at most k. This function obeys the following recurrence:

$$Dist(p,q,k) = \begin{cases} (p \in A) \oplus (q \in A) & \text{if } k = 0, \\ Dist(p,q,k-1) \vee \bigvee_{a \in \Sigma} Dist\big(\delta(p,a), \delta(q,a), k-1\big) & \text{otherwise.} \end{cases}$$

Moore's "table-filling" algorithm is just a space-efficient dynamic programming algorithm to evaluate this recurrence.

Example

To get a better idea how this algorithm works, let's visualize the algorithm running on our earlier brute-force DFA for strings containing the substring `11`. This DFA has four unreachable states: (FALSE, `11`), (TRUE, ε), (TRUE, `0`), and (TRUE, `1`). We remove these states, and relabel the remaining states for easier reference. (In an actual implementation, the states would almost certainly be represented by indices into an array anyway, not by mnemonic labels.)



Our brute-force DFA for strings containing the substring `11`, after removing all four unreachable states

The main algorithm initializes (the bottom half of) a 10×10 table as follows. (In the implementation, cells marked $\not\sim$ have value TRUE and blank cells have value FALSE.)

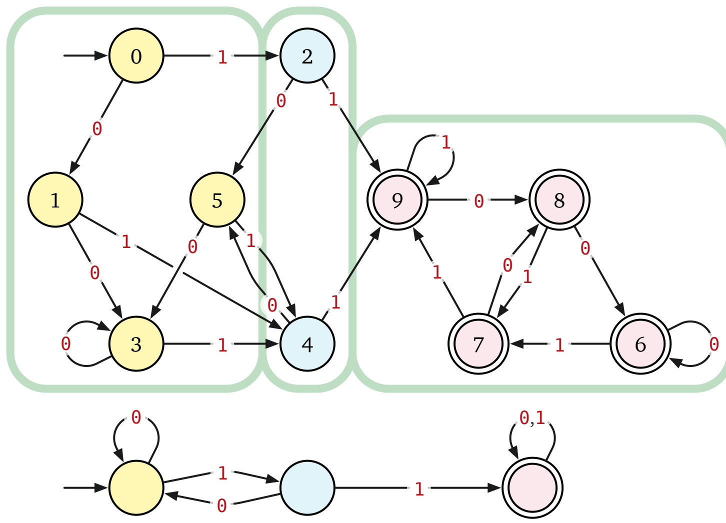
	0	1	2	3	4	5	6	7	8
1									
2									
3									
4									
5									
6	$\not\sim$	$\not\sim$	$\not\sim$	$\not\sim$	$\not\sim$	$\not\sim$			
7	$\not\sim$	$\not\sim$	$\not\sim$	$\not\sim$	$\not\sim$	$\not\sim$			
8	$\not\sim$	$\not\sim$	$\not\sim$	$\not\sim$	$\not\sim$	$\not\sim$			
9	$\not\sim$	$\not\sim$	$\not\sim$	$\not\sim$	$\not\sim$	$\not\sim$			

In the first iteration of the main loop, the algorithm discovers several distinguishable pairs of states. For example, the algorithm sets $Dist[0,2] \leftarrow$ TRUE because $Dist[\delta(0,1), \delta(2,1)] = Dist[2,9] =$ TRUE. After the iteration ends, the table looks like this:

	0	1	2	3	4	5	6	7	8
1									
2	$\not\sim$	$\not\sim$							
3			$\not\sim$						
4	$\not\sim$	$\not\sim$		$\not\sim$					
5			$\not\sim$		$\not\sim$				
6	$\not\sim$	$\not\sim$	$\not\sim$	$\not\sim$	$\not\sim$	$\not\sim$			
7	$\not\sim$	$\not\sim$	$\not\sim$	$\not\sim$	$\not\sim$	$\not\sim$			
8	$\not\sim$	$\not\sim$	$\not\sim$	$\not\sim$	$\not\sim$	$\not\sim$			
9	$\not\sim$	$\not\sim$	$\not\sim$	$\not\sim$	$\not\sim$	$\not\sim$			

The second iteration of the while loop makes no further changes to the table—We got lucky!—so the algorithm terminates.

The final table implies that the states of our trimmed DFA fall into exactly three equivalence classes: $\{0,1,3,5\}$, $\{2,4\}$, and $\{6,7,8,9\}$. Replacing each equivalence class with a single state gives us the three-state DFA that we already discovered.



Equivalence classes of states in the trimmed DFA, and the resulting minimal equivalent DFA.

Exercises

1. For each of the following languages in $\{0,1\}^*$, describe a deterministic finite-state machine that accepts that language. There are infinitely many correct answers for each language. "Describe" does not necessarily mean "draw".

   (a) Only the string `0110`.

   (b) Every string except `0110`.

   (c) Strings that contain the substring `0110`.

   (d) Strings that do not contain the substring `0110`.

   *(e) Strings that contain an even number of occurrences of the substring `0110`. (For example, this language contains the strings `0110110` and `01011`.)

   (f) Strings that contain the *subsequence* `0110`.

   (g) Strings that do not contain the *subsequence* `0110`.

   (h) Strings that contain an even number of `1`s and an odd number of `0`s.

   (i) Strings that represent a number divisible by 7 in binary.

   (j) Strings whose reversals represent a number divisible by 7 in binary.

   (k) Strings in which the substrings `01` and `10` appear the same number of times.

   (l) Strings such that in every prefix, the number of `0`s and the number of `1`s differ by at most 1.

   (m) Strings such that in every prefix, the number of `0`s and the number of `1`s differ by at most 4.

   (n) Strings that end with 0^{10} = `0000000000`.

(o) Strings in which the number of 1s is even, the number of 0s is divisible by 3, the overall length is divisible by 5, the binary value is divisible by 7, the binary value of the reversal is divisible by 11, and does not contain thirteen 1s in a row. *[Hint: This is more tedious than difficult.]*

2. (a) Let $L \subseteq 0^*$ be an arbitrary *unary* language. Prove that L^* is regular.
   (b) Prove that there is a binary language $L \subseteq (0+1)^*$ such that L^* is not regular.

3. Describe and analyze algorithms for the following problems. In each case, the input is a DFA M over the alphabet $\Sigma = \{0, 1\}$.
   (a) Does M accept any string whose length is a multiple of 5?
   (b) Does M accept every string that represents a number divisible by 7 in binary?
   (c) Does M accept an infinite number of strings containing an odd number of 0s?
   (d) Does M accept a finite number of strings that contain the substring 0110110 and whose length is divisible by five?
   (e) Does M accept *only* strings whose lengths are perfect squares?
   (f) Does M accept any string whose length is *composite*?
   ⋆(g) Does M accept any string whose length is *prime*?

★★★ Move these to the graph traversal notes?

4. Prove that each of the following languages cannot be accepted by a DFA.
   (a) $\{0^{n^2} \mid n \geq 0\}$
   (b) $\{0^{n^3} \mid n \geq 0\}$
   (c) $\{0^{f(n)} \mid n \geq 0\}$, where $f(n)$ is *any* fixed polynomial in n with degree at least 2.
   (d) $\{0^n \mid n \text{ is composite}\}$
   (e) $\{0^n 1 0^n \mid n \geq 0\}$
   (f) $\{0^i 1^j \mid i \neq j\}$
   (g) $\{0^i 1^j \mid i < 3j\}$
   (h) $\{0^i 1^j \mid i \text{ and } j \text{ are relatively prime}\}$
   (i) $\{0^i 1^j \mid j - i \text{ is a perfect square}\}$
   (j) $\{w\#w \mid w \in (0+1)^*\}$
   (k) $\{ww \mid w \in (0+1)^*\}$
   (l) $\{w\#0^{|w|} \mid w \in (0+1)^*\}$
   (m) $\{w0^{|w|} \mid w \in (0+1)^*\}$
   (n) $\{xy \mid w, x \in (0+1)^* \text{ and } |x| = |y| \text{ but } x \neq y\}$
   (o) $\{0^m 1^n 0^{m+n} \mid m, n \geq 0\}$

(p) $\{0^m 1^n 0^{mn} \mid m, n \geq 0\}$

(q) Strings in which the substrings 00 and 11 appear the same number of times.

(r) The set of all palindromes in $(0+1)^*$ whose length is divisible by 7.

(s) $\{w \in (0+1)^* \mid w \text{ is the binary representation of a perfect square}\}$

★(t) $\{w \in (0+1)^* \mid w \text{ is the binary representation of a prime number}\}$

5. For each of the following languages over the alphabet $\Sigma = \{0, 1\}$, either describe a DFA that accepts the language or prove that no such DFA exists. Recall that Σ^+ denotes the set of all *nonempty* strings over Σ. *[Hint: Believe it or not, most of these languages* ***can*** *be accepted by DFAs.]*

   (a) $\left\{0^n w 1^n \;\middle|\; w \in \Sigma^* \text{ and } n \geq 0\right\}$

   (b) $\left\{0^n 1^n w \;\middle|\; w \in \Sigma^* \text{ and } n \geq 0\right\}$

   (c) $\left\{w 0^n 1^n x \;\middle|\; w, x \in \Sigma^* \text{ and } n \geq 0\right\}$

   (d) $\left\{0^n w 1^n x \;\middle|\; w, x \in \Sigma^* \text{ and } n \geq 0\right\}$

   (e) $\left\{0^n w 1 x 0^n \;\middle|\; w, x \in \Sigma^* \text{ and } n \geq 0\right\}$

   (f) $\left\{w x w \;\middle|\; w, x \in \Sigma^*\right\}$

   (g) $\left\{w x w \;\middle|\; w, x \in \Sigma^+\right\}$

   (h) $\left\{w x w^R \;\middle|\; w, x \in \Sigma^+\right\}$

   (i) $\left\{w w x \;\middle|\; w, x \in \Sigma^+\right\}$

   (j) $\left\{w w^R x \;\middle|\; w, x \in \Sigma^+\right\}$

   (k) $\left\{w x w y \;\middle|\; w, x, y \in \Sigma^+\right\}$

   (l) $\left\{w x w^R y \;\middle|\; w, x, y \in \Sigma^+\right\}$

   (m) $\left\{x w w y \;\middle|\; w, x, y \in \Sigma^+\right\}$

   (n) $\left\{x w w^R y \;\middle|\; w, x, y \in \Sigma^+\right\}$

   (o) $\left\{w x x w \;\middle|\; w, x \in \Sigma^+\right\}$

   ⋆(p) $\left\{w x w^R x \;\middle|\; w, x \in \Sigma^+\right\}$

Caveat lector! This is the first edition of this lecture note. Some topics are incomplete, and there are almost certainly a few serious errors. Please send bug reports and suggestions to jeffe@illinois.edu.

Nothing is better than simplicity . . . .
nothing can make up for excess or for the lack of definiteness.
— Walt Whitman, Preface to *Leaves of Grass* (1855)

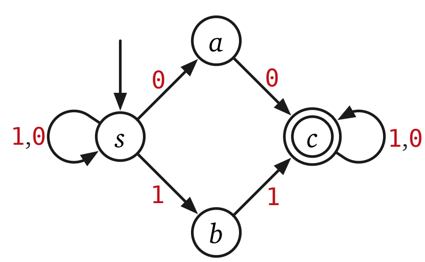
Freedom of choice
Is what you got.
Freedom from choice
Is what you want.
— Devo, "Freedom of Choice", *Freedom of Choice* (1980)

Nondeterminism means never having to say you are wrong.
— BSD 4.3 fortune(6) file (c.1985)

4 Nondeterminism

4.1 Nondeterministic State Machines

The following diagram shows something that looks like a finite-state machine over the alphabet {0, 1}, but on closer inspection, it is not consistent with our earlier definitions. On one hand, there are two transitions out of s for each input symbol. On the other hand, states a and b are each missing an outgoing transition.



A nondeterministic finite-state automaton

Nevertheless, there is a sense in which this machine "accepts" the set of all strings that contain either 00 or 11 as a substring. Imagine that when the machine reads a symbol in state s, it makes a choice about which transition to follow. If the input string contains the substring 00, then it is *possible* for the machine to end in the accepting state c, by *choosing* to move into state a when it reads a 0 immediately before another 0. Similarly, if the input string contains the substring 11, it is *possible* for the machine to end in the accepting state c. On the other hand, if the input string does not contain either 00 or 11—or in other words, if the input alternates between 0 and 1—there are no choices that lead the machine to the accepting state. If the machine incorrectly chooses to transition to state a and then reads a 1, or transitions to b and then reads 0, it explodes; the only way to avoid an explosion is to stay in state s.

This object is an example of a ***nondeterministic finite-state automaton***, or ***NFA***, so named because its behavior is not uniquely *determined* by the input string. Formally, every NFA has five components:

- An arbitrary finite set $\boldsymbol{\Sigma}$, called the ***input alphabet***.
- Another arbitrary finite set $\boldsymbol{Q}$, whose elements are called ***states***.
- An arbitrary ***transition*** function $\boldsymbol{\delta}: Q \times \Sigma \to 2^Q$.
- A ***start state*** $\boldsymbol{s} \in Q$.
- A subset $\boldsymbol{A} \subseteq Q$ of ***accepting states***.

The only difference from the formal definition of *deterministic* finite-state automata is the domain of the transition function. In a DFA, the transition function always returns a single state; in an NFA, the transition function returns a *set* of states, which could be empty, or all of Q, or anything in between.

Just like DFAs, the behavior of an NFA is governed by an ***input string*** $w \in \Sigma^*$, which the machine reads one symbol at a time, from left to right. Unlike DFAs, however, an NFA does not maintain a single current state, but rather a *set* of current states. Whenever the NFA reads a symbol a, its set of current states changes from C to $\bigcup_{q \in C} \delta(q, a)$. After all symbols have been read, the NFA ***accepts*** w if its current state set contains *at least one* accepting state and ***rejects*** w otherwise. In particular, if the set of current states ever becomes empty, it will stay empty forever, and the NFA will reject.

More formally, we define the function $\delta^*: Q \times \Sigma^* \to 2^Q$ that transitions on *strings* as follows:

$$\delta^*(q, w) := \begin{cases} \{q\} & \text{if } w = \varepsilon, \\ \displaystyle\bigcup_{r \in \delta(q,a)} \delta^*(r, x) & \text{if } w = ax. \end{cases}$$

The NFA $(Q, \Sigma, \delta, s, A)$ ***accepts*** $w \in \Sigma^*$ if and only if $\delta^*(s, w) \cap A \neq \varnothing$.

We can equivalently define an NFA as a directed graph whose vertices are the states Q, whose edges are labeled with symbols from Σ. We no longer require that every vertex has exactly one outgoing edge with each label; it may have several such edges or none. An NFA accepts a string w if the graph contains *at least one* walk from the start state to an accepting state whose label is w.

4.2 Intuition

There are at least three useful ways to think about non-determinism.

Clairvoyance. Whenever an NFA reads symbol a in state q, it *chooses* the next state from the set $\delta(q, a)$, always *magically* choosing a state that leads to the NFA accepting the input string, unless no such choice is possible. As the BSD fortune file put it, "Nondeterminism means never having to say you're wrong."[1] Of course real machines can't actually look into the future; that's why I used the word "magic".

Parallel threads. An arguably more "realistic" view is that when an NFA reads symbol a in state q, it spawns an independent execution thread for each state in $\delta(q, a)$. In particular, if $\delta(q, a)$ is empty, the current thread simply dies. The NFA accepts if *at least one* thread is in an accepting state after it reads the last input symbol.

Equivalently, we can imagine that when an NFA reads symbol a in state q, it branches into several parallel universes, one for each state in $\delta(q, a)$. If $\delta(q, a)$ is empty, the NFA destroys the

[1] This sentence is a riff on a horrible aphorism that was (sadly) popular in the US in the 70s and 80s. Fortunately, everyone seems to have forgotten the original saying, except for that one time it was parodied on the Simpsons.

universe (including itself). Similarly, if the NFA finds itself in a non-accepting state when the input ends, the NFA destroys the universe. Thus, when the input is gone, only universes in which the NFA somehow chose a path to an accept state still exist. One slight disadvantage of this metaphor is that if an NFA reads a string that is not in its language, it destroys *all* universes.

Proofs/oracles. Finally, we can treat NFAs not as a mechanism for *computing* something, but only as a mechanism for *verifying proofs*. If we want to *prove* that a string w contains one of the suffixes `00` or `11`, it suffices to demonstrate a single walk in our example NFA that starts at s and ends at c, and whose edges are labeled with the symbols in w. Equivalently, whenever the NFA faces a nontrivial choice, the prover can simply tell the NFA which state to move to next.

This intuition can be formalized as follows. Consider a *deterministic* finite state machine whose input alphabet is the product $\Sigma \times \Omega$ of an ***input*** alphabet Σ and an ***oracle*** alphabet Ω. Equivalently, we can imagine that this DFA reads simultaneously from two strings of the same length: the *input* string w and the *oracle* string ω. In either formulation, the transition function has the form $\delta : Q \times \Sigma \times \Omega \to Q$. As usual, this DFA accepts the pair $(w, \omega) \in (\Sigma \times \Gamma)^*$ if and only if $\delta^*(s, w, \omega) \in A$. Finally, M ***nondeterministically accepts*** the string $w \in \Sigma^*$ if there is an oracle string $\omega \in \Omega^*$ with $|\omega| = |w|$ such that $(w, \omega) \in L(M)$.

4.3 ε-Transitions

It is fairly common for NFAs to include so-called ***ε-transitions***, which allow the machine to change state without reading an input symbol. An NFA with ε-transitions accepts a string w if and only if there is a sequence of transitions $s \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} \cdots \xrightarrow{a_\ell} q_\ell$ where the final state q_ℓ is accepting, each a_i is either ε or a symbol in Σ, and $a_1 a_2 \cdots a_\ell = w$.

More formally, the transition function in an NFA with ε-transitions has a slightly larger domain $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q$. The ***ε-reach*** of a state $q \in Q$ consists of all states r that satisfy one of the following conditions:

- $r = q$
- $r \in \delta(q', \varepsilon)$ for some state q' in the ε-reach of q.

In other words, r is in the ε-reach of q if there is a (possibly empty) sequence of ε-transitions leading from q to r. Now we redefine the extended transition function $\delta^* : Q \times \Sigma^* \to 2^Q$, which transitions on arbitrary strings, as follows:

$$\delta^*(q, w) := \begin{cases} \{q\} & \text{if } w = \varepsilon, \\ \displaystyle\bigcup_{r \in \varepsilon\text{-reach}(q)} \bigcup_{r' \in \delta(r,a)} \delta^*(r', x) & \text{if } w = ax. \end{cases}$$

As usual, the modified NFA accepts a string w if and only if $\delta^*(s, w) \cap A \neq \varnothing$.

Given an NFA $M = (\Sigma, Q, s, A, \delta)$ with ε-transitions, we can easily construct an equivalent NFA $M' = (\Sigma, Q', s', A', \delta')$ without ε-transitions as follows:

$$\begin{aligned} Q' &:= Q \\ s' &= s \\ A' &= \big\{ q \in Q \bigm| \varepsilon\text{-reach}(q) \cap A \neq \varnothing \big\} \\ \delta'(q, a) &= \bigcup_{r \in \varepsilon\text{-reach}(q)} \delta(r, a) \end{aligned}$$

Straightforward definition-chasing implies that M and M' accept exactly the same language. Thus, whenever we reason about or design NFAs, we are free to either allow or forbid ε-transitions, whichever is more convenient for the task at hand.

4.4 Kleene's Theorem

We are now finally in a position to prove the following fundamental fact, first observed by Steven Kleene:

Theorem 4.1. *A language L can be described by a regular expression if and only if L is the language accepted by a DFA.*

We will prove Kleene's fundamental theorem in four stages:

- Every DFA can be transformed into an equivalent NFA.
- Every NFA can be transformed into an equivalent DFA.
- Every regular expression can be transformed into an NFA.
- Every NFA can be transformed into an equivalent regular expression.

The first of these four transformations is completely trivial; a DFA is just a special type of NFA where the transition function always returns a single state. Unfortunately, the other three transformations require a bit more work.

4.5 DFA from NFAs: The Subset Construction

In the parallel-thread model of NFA execution, an NFA does not have a single current state, but rather a *set* of current states. The evolution of this set of states is *determined* by a modified transition function $\delta'\colon 2^Q \times \Sigma \to 2^Q$, defined by setting $\delta'(P, a) := \bigcup_{p\in P} \delta(p, a)$ for any set of states $P \subseteq Q$ and any symbol $a \in \Sigma$. When the NFA finishes reading its input string, it accepts if and only if the current set of states intersects the set A of accepting states.

This formulation makes the NFA completely deterministic! We have just shown that any NFA $M = (\Sigma, Q, s, A, \delta)$ is equivalent to a DFA $M' = (\Sigma, Q', s', A', \delta')$ defined as follows:

$$\begin{aligned} Q' &:= 2^Q \\ s' &:= \{s\} \\ A' &:= \left\{ S \subseteq Q \,\middle|\, S \cap A \neq \varnothing \right\} \\ \delta'(q', a) &:= \bigcup_{p \in q'} \delta(p, a) \qquad \text{for all } q' \subseteq Q \text{ and } a \in \Sigma. \end{aligned}$$

Similarly, any NFA with ε-transitions is equivalent to a DFA with the transition function

$$\delta'(q', a) = \bigcup_{p\in q'} \; \bigcup_{r \in \varepsilon\text{-reach}(p)} \delta(r, a)$$

for all $q' \subseteq Q$ and $a \in \Sigma$. This conversion from NFA to DFA is often called the ***subset construction***, but that name is somewhat misleading; it's not a "construction" so much as a change in perspective.

One disadvantage of this "construction" is that it usually leads to DFAs with far more states than necessary, in part because most of those states are unreachable. These unreachable states can be avoided by constructing the DFA incrementally, essentially by performing a breadth-first search of the DFA graph, starting at its start state.

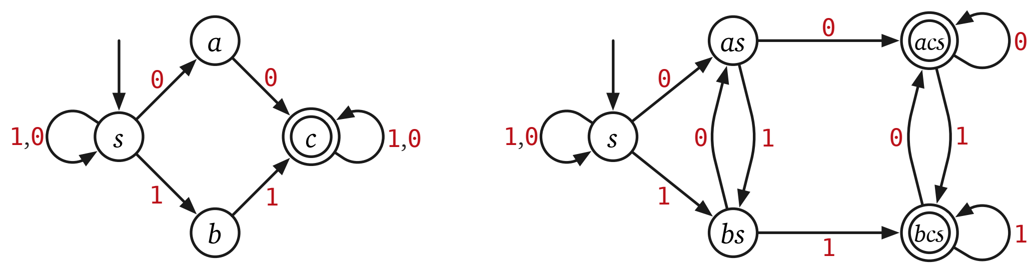
To execute this algorithm by hand, we prepare a table with $|\Sigma| + 3$ columns, with one row for each DFA state we discover. In order, these columns record the following information:

- The DFA state (as a subset of NFA states)
- The ε-reach of the corresponding subset of NFA states
- Whether the DFA state is accepting (that is, whether the ε-reach intersects A)
- The output of the transition function for each symbol in Σ.

We start with DFA-state s in the first row and first column. Whenever we discover an unexplored state in one of the last $|\Sigma|$ columns, we copy it to the left column in a new row.

For example, given the NFA on the first page of this note, this incremental algorithm produces the following table, yielding a five-state DFA. For this example, the second column is redundant, because the NFA has no ε-transitions, but we will see another example with ε-transitions in the next subsection. To simplify notation, we write each set of states as a simple string, omitting braces and commas.

q'	ε-reach(q')	$q' \in A'$?	$\delta'(q', 0)$	$\delta'(q', 1)$
s	s		as	bs
as	as		acs	bs
bs	bs		as	bcs
acs	acs	✓	acs	bcs
bcs	bcs	✓	acs	bcs



Our example NFA, and the output of the incremental subset construction for that NFA.

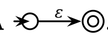
4.6 NFAs from Regular Expressions: Thompson's Algorithm

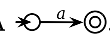
Lemma 4.2. *Every regular language is accepted by a non-deterministic finite automaton.*

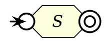
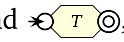
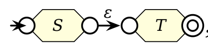
Proof: In fact, we will prove the following stronger claim: Every regular language is accepted by an NFA *with exactly one accepting state, which is different from its start state*. The following construction was first described by Ken Thompson in 1968. Thompson's algorithm actually proves a stronger statement: For any regular language L, there is an NFA that accepts L that has exactly one accepting state t, which is distinct from the starting state s.

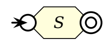
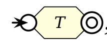
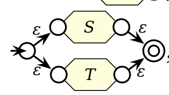
Let R be an arbitrary regular expression over an arbitrary finite alphabet Σ. Assume that for any sub-expression S of R, the language described by S is accepted by an NFA with one accepting state distinct from its start state, which we denote pictorially by S. There are six cases to consider—three base cases and three recursive cases—mirroring the recursive definition of a regular expression.

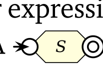
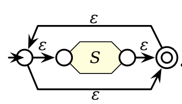
- If $R = \varnothing$, then $L(R) = \varnothing$ is accepted by the empty NFA: .

- If $R = \varepsilon$, then $L(R) = \{\varepsilon\}$ is accepted by the NFA .

- If $R = a$ for some symbol $a \in \Sigma$, then $L(R) = \{a\}$ is accepted by the NFA . (The case where R is a single *string* with length greater than 1 reduces to the single-symbol case by concatenation, as described in the next case.)

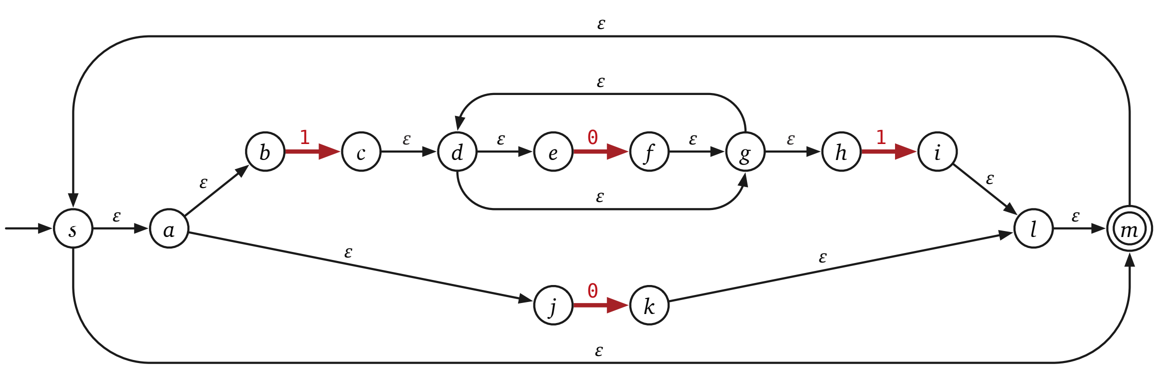
- Suppose $R = ST$ for some regular expressions S and T. The inductive hypothesis implies that the languages $L(S)$ and $L(T)$ are accepted by NFAs  and , respectively. Then $L(R) = L(ST) = L(S) \bullet L(T)$ is accepted by the NFA , built by connecting the two component NFAs in series.

- Suppose $R = S + T$ for some regular expressions S and T. The inductive hypothesis implies that the language $L(S)$ and $L(T)$ are accepted by NFAs  and , respectively. Then $L(R) = L(S + T) = L(S) \cup L(T)$ is accepted by the NFA , built by connecting the two component NFAs in parallel with new start and accept states.

- Finally, suppose $R = S^*$ for some regular expression S. The inductive hypothesis implies that the language $L(S)$ is accepted by an NFA . Then the language $L(R) = L(S^*) = L(S)^*$ is accepted by the NFA .

In all cases, the language $L(R)$ is accepted by an NFA with one accepting state, which is different from its start state, as claimed. □

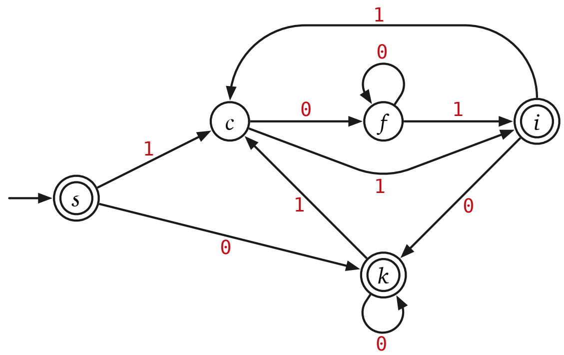
As an example, given the regular expression $(\texttt{0} + \texttt{10}^*\texttt{1})^*$ of strings containing an even number of $\texttt{1}$s, Thompson's algorithm produces a 14-state NFA shown on the next page. As this example shows, Thompson's algorithm tends to produce NFAs with many redundant states. Fortunately, just as there are for DFAs, there are algorithms that can reduce any NFA to an equivalent NFA with the smallest possible number of states.



The NFA constructed by Thompson's algorithm for the regular expression $(\texttt{0} + \texttt{10}^*\texttt{1})^*$.
The four non-ε-transitions are drawn with with bold red arrows for emphasis.

Interestingly, applying the incremental subset algorithm to Thompson's NFA tends to yield a DFA with relatively *few* states, in part because the states in Thompson's NFA tend to have large ε-reach, and in part because relatively few of those states are the targets of non-ε-transitions. Starting with the NFA shown above, for example, the incremental subset construction yields a DFA for the language $(\texttt{0} + \texttt{10}^*\texttt{1})^*$ with just five states:

q'	ε-reach(q')	$q' \in A'$?	$\delta'(q', 0)$	$\delta'(q', 1)$
s	$sabjm$	✓	k	c
k	$sabjklm$	✓	k	c
c	$cdegh$		f	i
f	$degh$		f	i
i	$sabjilm$	✓	k	c



The DFA computed by the incremental subset algorithm from Thompson's NFA for $(0 + 10^*1)^*$.

This DFA can be further simplified to just two states, by observing that all three accepting states are equivalent, and that both non-accepting states are equivalent. But still, five states is pretty good, especially compared with the $2^{13} = 8096$ states that the naïve subset construction would yield!

*4.7 NFAs from Regular Expressions: Glushkov's Algorithm

Thompson's algorithm is actually a modification of an earlier algorithm, which was independently discovered by Robert McNaughton and Hisao Yamada in 1960 and by V. I. Glushkov in 1961. Given a regular expression containing n symbols (not counting the parentheses and pluses and stars), Glushkov's algorithm produces an NFA with exactly $n+1$ states.

Glushkov's algorithm combines six functions on regular expressions:

- *index*(R) is the regular expression obtained by replacing the symbols in R with the integers 1 through n, in order from left to right. For example, *index*$((0 + 10^*1)^*) = (1 + 23^*4)^*$.
- *symbols*(R) denotes the string obtained by removing all non-symbols from R. For example, *symbols*$((0 + 10^*1)^*) = 0101$.
- *has-ε*(R) is True if $\varepsilon \in L(R)$ and False otherwise.
- *first*(R) is the set of all initial symbols of strings in $L(R)$.
- *last*(R) is the set of all final symbols of strings in $L(R)$.
- *middle*(R) is the set of all pairs (a, b) such that ab is a substring of some string in $L(R)$.

The last four functions obey the following recurrences:

$$
\begin{aligned}
\textit{has-}\varepsilon(\varnothing) &= \varnothing \\
\textit{has-}\varepsilon(w) &= \begin{cases} \textsc{True} & \text{if } w = \varepsilon \\ \textsc{False} & \text{otherwise} \end{cases} \\
\textit{has-}\varepsilon(S+T) &= \textit{has-}\varepsilon(S) \lor \textit{has-}\varepsilon(T) \\
\textit{has-}\varepsilon(ST) &= \textit{has-}\varepsilon(S) \land \textit{has-}\varepsilon(T) \\
\textit{has-}\varepsilon(S^*) &= \textsc{True}
\end{aligned}
$$

$$
\begin{aligned}
\textit{first}(\varnothing) &= \varnothing \\
\textit{first}(w) &= \begin{cases} \varnothing & \text{if } w = \varepsilon \\ \{a\} & \text{if } w = ax \end{cases} \\
\textit{first}(S+T) &= \textit{first}(S) \cup \textit{first}(T) \\
\textit{first}(ST) &= \begin{cases} \textit{first}(S) \cup \textit{first}(T) & \text{if } \textit{has-}\varepsilon(S) \\ \textit{first}(T) & \text{otherwise} \end{cases} \\
\textit{first}(S^*) &= \textit{first}(S)
\end{aligned}
$$

$$
\begin{aligned}
\textit{last}(\varnothing) &= \varnothing \\
\textit{last}(w) &= \begin{cases} \varnothing & \text{if } w = \varepsilon \\ \{a\} & \text{if } w = xa \end{cases} \\
\textit{last}(S+T) &= \textit{last}(S) \cup \textit{last}(T) \\
\textit{last}(ST) &= \begin{cases} \textit{last}(S) \cup \textit{last}(T) & \text{if } \textit{has-}\varepsilon(T) \\ \textit{last}(T) & \text{otherwise} \end{cases} \\
\textit{last}(S^*) &= \textit{last}(S)
\end{aligned}
$$

$$
\begin{aligned}
\textit{middle}(\varnothing) &= \varnothing \\
\textit{middle}(w) &= \begin{cases} \varnothing & \text{if } |w| \le 1 \\ \{(a,b)\} \cup \textit{middle}(bx) & \text{if } w = abx \end{cases} \\
\textit{middle}(S+T) &= \textit{middle}(S) \cup \textit{middle}(T) \\
\textit{middle}(ST) &= \textit{middle}(S) \cup (\textit{last}(S) \times \textit{first}(T)) \cup \textit{middle}(T) \\
\textit{middle}(S^*) &= \textit{middle}(S) \cup (\textit{last}(S) \times \textit{first}(S))
\end{aligned}
$$

For example, the set $\textit{middle}((1+23^*4)^*)$ can be computed recursively as follows. If we're doing this by hand, we can skip many of the steps in this derivation, because we know what the functions *first*, *middle*, *last*, and *has-*ε actually mean, but a mechanical recursive evaluation would necessarily evaluate every step.

$$
\begin{aligned}
&\textit{middle}((1+23^*4)^*) \\
&\quad = \textit{middle}(1+23^*4) \cup \big(\textit{last}(1+23^*4) \times \textit{first}(1+23^*4)\big) \\
&\quad = \textit{middle}(1) \cup \textit{middle}(23^*4) \cup \big(\textit{last}(1+23^*4) \times \textit{first}(1+23^*4)\big) \\
&\quad = \varnothing \cup \textit{middle}(23^*4) \cup \big(\textit{last}(1+23^*4) \times \textit{first}(1+23^*4)\big) \\
&\quad = \textit{middle}(2) \cup \big(\textit{last}(2) \times \textit{first}(3^*4)\big) \cup \textit{middle}(3^*4) \cup \big(\textit{last}(1+23^*4) \times \textit{first}(1+23^*4)\big) \\
&\quad = \varnothing \cup \big(\{2\} \times \textit{first}(3^*4)\big) \cup \textit{middle}(3^*4) \cup \big(\textit{last}(1+23^*4) \times \textit{first}(1+23^*4)\big) \\
&\quad = \big(\{2\} \times \big(\textit{first}(3^*) \cup \textit{first}(4)\big)\big) \cup \textit{middle}(3^*4) \cup \big(\textit{last}(1+23^*4) \times \textit{first}(1+23^*4)\big) \\
&\quad = \big(\{2\} \times \big(\textit{first}(3) \cup \textit{first}(4)\big)\big) \cup \textit{middle}(3^*4) \cup \big(\textit{last}(1+23^*4) \times \textit{first}(1+23^*4)\big) \\
&\quad = \big(\{2\} \times \{3,4\}\big) \cup \textit{middle}(3^*4) \cup \big(\textit{last}(1+23^*4) \times \textit{first}(1+23^*4)\big) \\
&\quad = \big\{(2,3),(3,4)\big\} \cup \textit{middle}(3^*4) \cup \big(\textit{last}(1+23^*4) \times \textit{first}(1+23^*4)\big) \\
&\quad\ \vdots \\
&\quad = \big\{(1,1),(1,2),(2,3),(2,4),(3,3),(3,4),(4,1),(4,2)\big\}
\end{aligned}
$$

Finally, given any regular expression R, Glushkov's algorithm constructs the NFA $M_R = (\Sigma, Q, s, A, \delta)$ that accepts exactly the language $L(R)$ as follows:

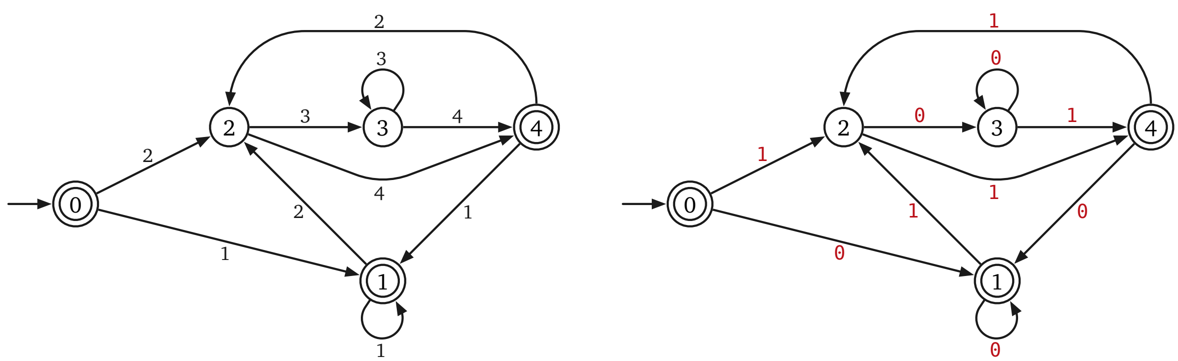
$$\begin{aligned}
Q &= \big\{0, 1, \ldots, |\mathit{symbols}(R)|\big\} \\
s &= 0 \\
A &= \begin{cases} \{0\} \cup \mathit{last}(\mathit{index}(R)) & \text{if } \mathit{has}\text{-}\varepsilon(R) \\ \mathit{last}(\mathit{index}(R)) & \text{otherwise} \end{cases} \\
\delta(0, a) &= \big\{j \in \mathit{first}(\mathit{index}(R)) \bigm| a = \mathit{symbols}(R)[j]\big\} \\
\delta(i, a) &= \big\{j \bigm| (i, j) \in \mathit{middle}(\mathit{index}(R)) \text{ and } a = \mathit{symbols}(R)[j]\big\}
\end{aligned}$$

There are a few natural ways to think about Glushkov's algorithm that are somewhat less impenetrable than the previous wall of definitions. One viewpoint is that Glushkov's algorithm first computes a *DFA* for the indexed regular expression *index*(R)—in fact, a DFA with the fewest possible states, except for an extra start state—and then replaces each index with the corresponding symbol in *symbols*(R) to get an NFA for the original expression R. Another useful observation is that Glushkov's NFA is identical to the result of removing all ε-transitions from Thompson's NFA for the same regular expression.

For example, given the regular expression $R = (\mathtt{0} + \mathtt{10}^*\mathtt{1})^*$, Glushkov's algorithm computes

$$\begin{aligned}
\mathit{index}(R) &= (1 + 23^*4)^* \\
\mathit{symbols}(R) &= \mathtt{0101} \\
\mathit{has}\text{-}\varepsilon(R) &= \textsc{True} \\
\mathit{first}(\mathit{index}(R)) &= \{1, 2\} \\
\mathit{last}(\mathit{index}(R)) &= \{1, 4\} \\
\mathit{middle}(\mathit{index}(R)) &= \big\{(1,1), (1,2), (2,3), (2,4), (3,3), (3,4), (4,1), (4,2)\big\}
\end{aligned}$$

and then constructs the following five-state NFA.

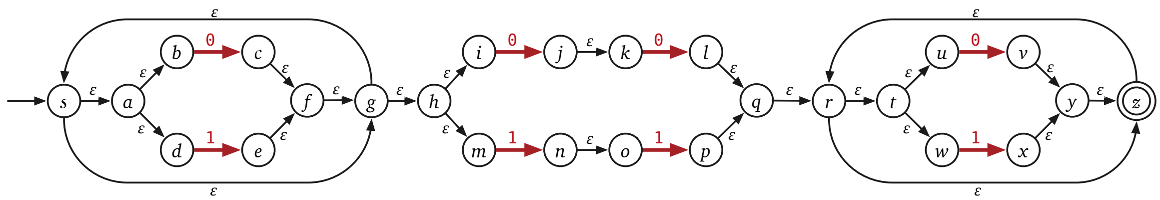


Glushkov's DFA for the index expression $(1 + 23^*4)^*$ and Glushkov's NFA for the regular expression $(\mathtt{0} + \mathtt{10}^*\mathtt{1})^*$.

Hey, look, Glushkov's algorithm actually gave us a DFA! In fact, it gave us *precisely* the same DFA that we constructed earlier by sending Thompson's NFA through the incremental subset algorithm! Unfortunately, that's just a coincidence; in general the output of Glushkov's algorithm is *not* deterministic. We'll see a more typical example in the next section.

4.8 Another Example

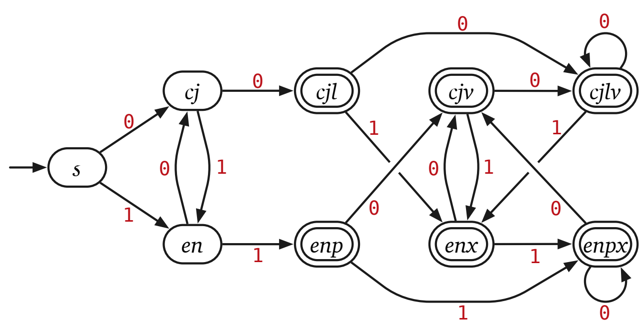
Here is another example of all the algorithms we’ve seen so far, starting with the regular expression $(0+1)^*(00+11)(0+1)^*$, which describes the language accepted by our very first example NFA. Thompson’s algorithm constructs the following 26-state monster:



Thompson’s NFA for the regular expression $(0+1)^*(00+11)(0+1)^*$

Given this NFA as input, the incremental subset construction computes the following table, leading to a DFA with just nine states. Yeah, the ε-reaches get a bit ridiculous; unfortunately, this *is* typical for Thompson’s NFA.

q'	ε-reach(q')	$q' \in A'$?	$\delta'(q', 0)$	$\delta'(q', 1)$
s	$sabdghim$		cj	en
cj	$sabdfghijkm$		cjl	en
en	$sabdfghmno$		cj	enp
cjl	$sabdfghijklmqrtuwz$	✓	$cjlv$	enx
enp	$sabdfghmnopqrtuwz$	✓	cjv	$enpx$
$cjlv$	$sabdfghijklmqrtuvwyz$	✓	$cjlv$	enx
enx	$sabdfghmnopqrtuwxyz$	✓	cjv	$enpx$
cjv	$sabdfghijkmrtuvwyz$	✓	$cjlv$	enx
$enpx$	$sabdfghmnopqrtuwxyz$	✓	cjv	$enpx$



The DFA computed by the incremental subset algorithm from Thompson’s NFA for $(0+1)^*(00+11)(0+1)^*$.

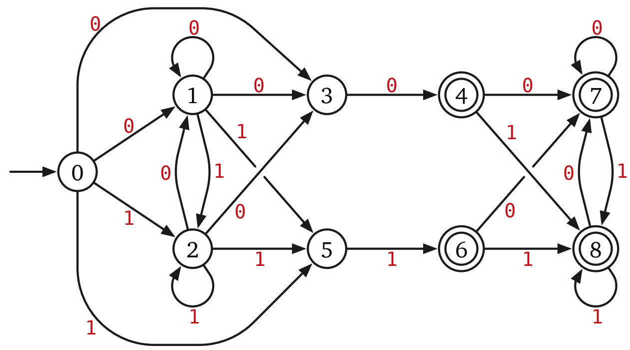
This DFA has far more states that necessary, intuitively because it keeps looking for 00 and 11 substrings even after it’s already found one. After all, when Thompson’s NFA finds a00 and 11 substring, it doesn’t kill all the other parallel threads, because it *can’t*. NFAs often have significantly fewer states than equivalent DFAs, but that efficiency also makes them kind of stupid.

Glushkov’s algorithm recursively computes the following values for the same regular expression

$R = (\texttt{0}+\texttt{1})^*(\texttt{00}+\texttt{11})(\texttt{0}+\texttt{1})^*$:

$$\begin{aligned}
index(R) &= (1+2)^*(34+56)(7+8)^* \\
symbols(R) &= \texttt{01001101} \\
has\text{-}\varepsilon(R) &= \textsc{False} \\
first(index(R)) &= \{1,2,3,5\} \\
last(index(R)) &= \{4,6,7,8\} \\
middle(index(R)) &= \big\{(1,1),(1,2),(2,1),(2,2),(1,3),(1,5),(2,3),(2,5),(3,4), \\
&\qquad (5,6),(4,7),(4,8),(6,7),(6,8),(7,7),(7,8),(8,7),(8,8)\big\}
\end{aligned}$$

These values imply the nine-state NFA shown below. Careful readers should confirm that running the incremental subset construction on this NFA yields exactly the same DFA (with different state names) as it did for Thompson's NFA.



Glushkov's NFA for $(\texttt{0}+\texttt{1})^*(\texttt{00}+\texttt{11})(\texttt{0}+\texttt{1})^*$

*4.9 Regular Expressions from NFAs: Han and Wood's Algorithm

The only component of Kleene's theorem we still have to prove is that every language accepted by a DFA or NFA is regular. As usual, it is actually easier to prove a stronger result. We consider a more general class of finite-state machines called ***expression automata***, introduced by Yo-Sub Han and Derick Wood in 2005.[2] Formally, an expression automaton consists of the following components:

- A finite set Σ called the ***input alphabet***
- Another finite set Q whose elements are called ***states***
- A ***start state*** $s \in Q$
- A single ***terminal state*** $t \in Q \setminus \{s\}$
- A ***transition function*** $R\colon (Q\setminus\{t\}) \times (Q\setminus\{s\}) \to Reg(\Sigma)$, where $Reg(\Sigma)$ is the set of regular expressions over Σ

Less formally, an expression automaton is a directed graph that includes a directed edge $p \to q$ labeled with a regular expression $R(p \to q)$, from *every* vertex p to *every* vertex q (including $q = p$), where by convention, we require that $R(q \to s) = R(t \to q) = \varnothing$ for every vertex q.

[2] Yo-Sub Han* and Derick Wood. The generalization of generalized automata: Expression automata. International Journal of Foundations of Computer Science 16(3):499–510, 2005.

We say that string w ***matches*** a transition $p \to q$ if w matches the regular expression $R(p \to q)$. In particular, if $R(p \to q) = \varnothing$, then no string matches $p \to q$. More generally, w matches a sequence of states $q_0 \to q_1 \to \cdots \to q_k$ if w matches the regular expression $R(q_0 \to q_1) \bullet R(q_1 \to q_2) \bullet \cdots \bullet R(q_{k-1} \to q_k)$. Equivalently, w matches the sequence $q_0 \to q_1 \to \cdots \to q_k$ if either

- $w = \varepsilon$ and the sequence has only one state ($k = 0$), or
- $w = xy$ for some string x that matches the regular expression $R(q_0 \to q_1)$ and some string y that matches the remaining sequence $q_1 \to \cdots \to q_k$.

An expression automaton ***accepts*** any string that matches at least one sequence of states that starts at s and ends at t. The ***language*** of an expression automaton E is the set of all strings that E accepts.

Expression automata are nondeterministic. A single string could match several (even infinitely many) state sequences that start with s, and it could match each of those state sequences in several different ways. A string is accepted if *at least one* of the state sequences it matches ends at t. Conversely, a string might match *no* state sequences; all such strings are rejected.

Two special cases of expression automata are already familiar. First, every regular language is clearly the language of an expression automaton with exactly two states. Second, with only minor modifications, any DFA or NFA can be converted into an expression automaton with trivial transition expressions. Thompson's algorithm can be used to transform any expression automaton into an NFA, by recursively expanding any nontrivial transition. To complete the proof of Kleene's theorem, we show how to convert any expression automaton into a regular expression by repeatedly deleting vertices.

Lemma 4.3. *Every expression automaton accepts a regular language.*

Proof: Let $E = (Q, \Sigma, R, s, t)$ be an arbitrary expression automaton. Assume that any expression automaton with fewer states than E accepts a regular language. There are two cases to consider, depending on the number of states in Q:

- If $Q = \{s, t\}$, then trivially, E accepts the regular language $R(s \to t)$.
- On the other hand, suppose there is a state $q \in Q \setminus \{s, a\}$. We can modify the expression automaton so that state q is redundant and can be removed. Define a new transition function $R' \colon Q \times Q \to \mathit{Reg}(\Sigma)$ by setting

  $$R'(p \to r) := R(p \to r) + R(p \to q)\, R(q \to q)^*\, R(q \to r).$$

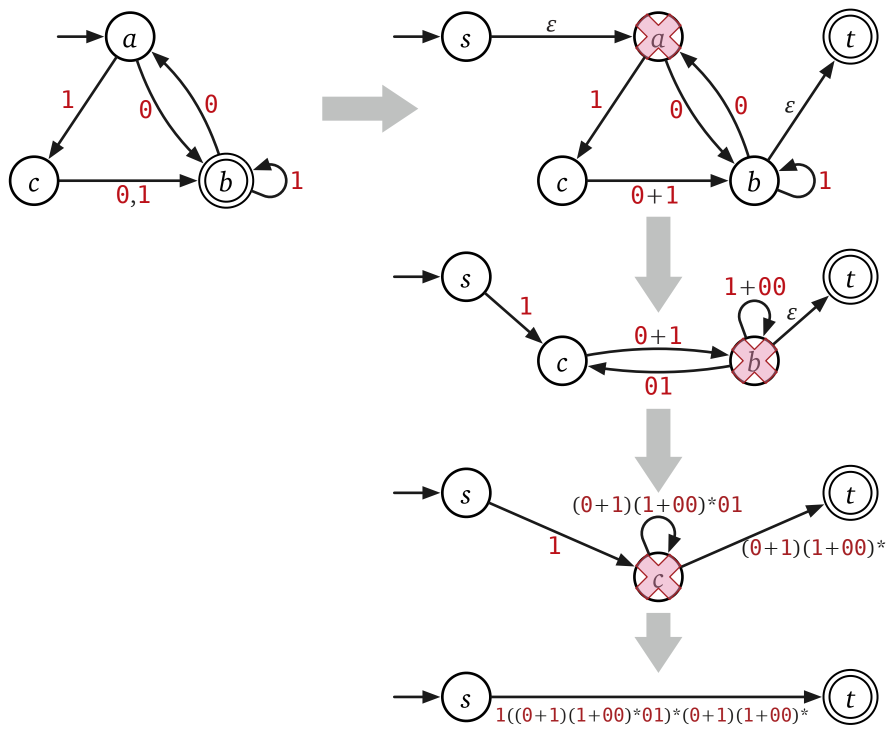
  With this modified transition function in place, any string w that matches the sequence $p \to q \to q \to \cdots \to q \to r$ with any number of q's also matches the single transition $p \to r$. Thus, by induction, if w matches a sequence of states, it also matches the subsequence obtained by removing all q's. Let E' be the expression automaton with states $Q' = Q \setminus \{q\}$ that uses this modified transition function R'. This new automaton accepts exactly the same strings as the original automaton E. Because E' has fewer states than E, the inductive hypothesis implies E' accepts a regular language.

In both cases, we conclude that E accepts a regular language. $\square$

This proof can be mechanically translated into an algorithm to convert any NFA—in particular, any DFA—into an equivalent regular expression. Given an NFA with n states (including s and

a), the algorithm iteratively removes $n-2$ states, updating $O(n^2)$ transition expressions in each iteration. If the concatenation and Kleene star operations could be performed in constant time, the resulting algorithm would run in $O(n^3)$ time. However, in each iteration, the transition expressions grows in length by roughly a factor of 4 in the worst case, so the final expression has length $\Theta(4^n)$. If we insist on representing the expressions as explicit strings, the worst-case running time is actually $\Theta(4^n)$.

A figure on the next page shows this conversion algorithm in action for a simple DFA. First we convert the DFA into an expression automaton by adding new start and accept states and merging two transitions, and then we remove each of the three original states, updating the transition expressions between any remaining states at each iteration. For the sake of clarity, edges $p \to q$ with $R(p \to q) = \varnothing$ are omitted from the figures.



Converting a DFA into an equivalent regular expression.

Exercises

1. For each of the following NFAs, describe an equivalent DFA. ("Describe" does not necessarily mean "draw"!)

★★★ Half a dozen examples.

2. For each of the following regular expressions, draw an equivalent NFA.

★★★ Half a dozen examples.

3. For each of the following regular expressions, describe an equivalent DFA. ("Describe" does not necessarily mean "draw"!)

★★★

Half a dozen examples.

4. Let $L \subseteq \Sigma^*$ be an arbitrary regular language. Prove that the following languages are regular.

   (a) $ones(L) := \left\{ w \in 1^* \;\middle|\; |w| = |x| \text{ for some } x \in L \right\}$

   (b) $reverse(L) := \left\{ w \in \Sigma^* \;\middle|\; w^R \in L \right\}$. (Recall that w^R denotes the reversal of string w.)

   (c) $prefix(L) := \{x \in \Sigma^* \mid xy \in L \text{ for some } y \in \Sigma^*\}$

   (d) $suffix(L) := \{y \in \Sigma^* \mid xy \in L \text{ for some } x \in \Sigma^*\}$

   (e) $substring(L) := \{y \in \Sigma^* \mid xyz \in L \text{ for some } x, z \in \Sigma^*\}$

   (f) $cycle(L) := \{xy \mid x, y \in \Sigma^* \text{ and } yx \in L\}$

   (g) $prefmax(L) := \{x \in L \mid xy \in L \iff y = \varepsilon\}$.

   (h) $sufmin(L) := \{xy \in L \mid y \in L \iff x = \varepsilon\}$.

   (i) $everyother(L) := \{everyother(w) \mid w \in L\}$, where $everyother(w)$ is the subsequence of w containing every other symbol. For example, $everyother(\mathtt{EVERYOTHER}) = \mathtt{VROHR}$.

   (j) $rehtoyreve(L) := \{w \in \Sigma^* \mid everyother(w) \in L\}$.

   (k) $subseq(L) := \{x \in \Sigma^* \mid x \text{ is a subsequence of some } y \in L\}$

   (l) $superseq(L) := \{x \in \Sigma^* \mid \text{some } y \in L \text{ is a subsequence of } x\}$

   (m) $left(L) := \{x \in \Sigma^* \mid xy \in L \text{ for some } y \in \Sigma^* \text{ where } |x| = |y|\}$

   (n) $right(L) := \{y \in \Sigma^* \mid xy \in L \text{ for some } x \in \Sigma^* \text{ where } |x| = |y|\}$

   (o) $middle(L) := \{y \in \Sigma^* \mid xyz \in L \text{ for some } x, z \in \Sigma^* \text{ where } |x| = |y| = |z|\}$

   (p) $half(L) := \{w \in \Sigma^* \mid ww \in L\}$

   (q) $third(L) := \{w \in \Sigma^* \mid www \in L\}$

   (r) $reflect(L) := \left\{ w \in \Sigma^* \;\middle|\; ww^R \in L \right\}$

   ⋆(s) $sqrt(L) := \left\{ x \in \Sigma^* \;\middle|\; xy \in L \text{ for some } y \in \Sigma^* \text{ such that } |y| = |x|^2 \right\}$

   ⋆(t) $log(L) := \left\{ x \in \Sigma^* \;\middle|\; xy \in L \text{ for some } y \in \Sigma^* \text{ such that } |y| = 2^{|x|} \right\}$

   ⋆(u) $flog(L) := \left\{ x \in \Sigma^* \;\middle|\; xy \in L \text{ for some } y \in \Sigma^* \text{ such that } |y| = F_{|x|} \right\}$, where F_n is the nth Fibonacci number.

⋆5. Let $L \subseteq \Sigma^*$ be an arbitrary regular language. Prove that the following languages are regular. *[Hint: For each language, there is an accepting NFA with at most q^q states, where q is the number of states in some DFA that accepts L.]*

(a) $repeat(L) := \{w \in \Sigma^* \mid w^n \in L \text{ for some } n \geq 0\}$

(b) $allreps(L) := \{w \in \Sigma^* \mid w^n \in L \text{ for every } n \geq 0\}$

(c) $manyreps(L) := \{w \in \Sigma^* \mid w^n \in L \text{ for infinitely many } n \geq 0\}$

(d) $fewreps(L) := \{w \in \Sigma^* \mid w^n \in L \text{ for finitely many } n \geq 0\}$

(e) $powers(L) := \left\{w \in \Sigma^* \;\middle|\; w^{2^n} \in L \text{ for some } n \geq 0\right\}$

★(f) $whatthe_N(L) := \{w \in \Sigma^* \mid w^n \in L \text{ for some } n \in N\}$, where N is an ***arbitrary*** fixed set of non-negative integers. *[Hint: You only have to prove that an accepting NFA exists; you don't have to describe how to construct it.]*

6. For each of the following expression automata, describe an equivalent DFA *and* an equivalent regular expression.

★★★

Half a dozen examples.

Caveat lector: This is the first edition of this lecture note. Please send bug reports and suggestions to jeffe@illinois.edu.

Imagine a piano keyboard, eh, 88 keys, only 88 and yet, and yet, hundreds of new melodies, new tunes, new harmonies are being composed upon hundreds of different keyboards every day in Dorset alone. Our language, tiger, our language: hundreds of thousands of available words, frillions of legitimate new ideas, so that I can say the following sentence and be utterly sure that nobody has ever said it before in the history of human communication: "***Hold the newsreader's nose squarely, waiter, or friendly milk will countermand my trousers.***" *Perfectly ordinary words, but never before put in that precise order. A unique child delivered of a unique mother.*

— Stephen Fry, *A Bit of Fry and Laurie*, Series 1, Episode 3 (1989)

5 Context-Free Languages and Grammars

5.1 Definitions

Intuitively, a language is regular if it can be built from individual strings by concatenation, union, and repetition. In this note, we consider a wider class of ***context-free*** languages, which are languages that can be built from individual strings by concatenation, union, and *recursion*.

Formally, a language is context-free if and only if it has a certain type of recursive description known as a ***context-free grammar***, which is a structure with the following components:

- A finite set Σ, whose elements are called ***symbols*** or ***terminals***.
- A finite set Γ disjoint from Σ, whose elements are called ***non-terminals***.
- A finite set R of ***production rules*** of the form $A \rightarrow w$, where $A \in \Gamma$ is a non-terminal and $w \in (\Sigma \cup \Gamma)^*$ is a string of symbols and variables.
- A ***starting*** non-terminal, typically denoted S.

For example, the following eight production rules describe a context free grammar with terminals $\Sigma = \{\mathtt{0}, \mathtt{1}\}$ and non-terminals $\Gamma = \{S, A, B\}$:

$$\begin{array}{llll} S \rightarrow A & A \rightarrow \mathtt{0}A & B \rightarrow B\mathtt{1} & C \rightarrow \varepsilon \\ S \rightarrow B & A \rightarrow \mathtt{0}C & B \rightarrow C\mathtt{1} & C \rightarrow \mathtt{0}C\mathtt{1} \end{array}$$

Normally we write grammars more compactly by combining the right sides of all rules for each non-terminal into one list, with alternatives separated by vertical bars.[1] For example, the previous grammar can be written more compactly as follows:

$$\begin{aligned} S &\rightarrow A \mid B \\ A &\rightarrow \mathtt{0}A \mid \mathtt{0}C \\ B &\rightarrow B\mathtt{1} \mid C\mathtt{1} \\ C &\rightarrow \varepsilon \mid \mathtt{0}C\mathtt{1} \end{aligned}$$

For the rest of this lecture, I will *almost* always use the following notational conventions.

[1] Yes, this means we now have *three* symbols $\cup$, $+$, and $\mid$ with exactly the same meaning. Sigh.

- Monospaced digits (`0`, `1`, `2`, ...), and symbols (◇, `$`, `#`, •, ...) are explicit terminals.
- Early lower-case Latin letters ($a, b, c, \ldots$) represent unknown or arbitrary terminals in Σ.
- Upper-case Latin letters ($A, B, C, \ldots$) and the letter S represent non-terminals in Γ.
- Late lower-case Latin letters ($\ldots, w, x, y, z$) represent strings in $(\Sigma \cup \Gamma)^*$, whose characters could be either terminals or non-terminals.

We can ***apply*** a production rule to a string in $(\Sigma \cup \Gamma)^*$ by replacing any instance of the non-terminal on the left of the rule with the string on the right. More formally, for any strings $x, y, z \in (\Sigma \cup \Gamma)^*$ and any non-terminal $A \in \Gamma$, applying the production rule $A \to y$ to the string xAz yields the string xyz. We use the notation $\boldsymbol{xAz \rightsquigarrow xyz}$ to describe this application. For example, we can apply the rule $C \to \mathtt{0}C\mathtt{1}$ to the string $\mathtt{00}C\mathtt{1}BAC\mathtt{0}$ in two different ways:

$$\mathtt{00}\underbrace{C}\mathtt{1}BAC\mathtt{0} \rightsquigarrow \mathtt{00}\underbrace{\mathtt{0}C\mathtt{1}}\mathtt{1}BAC\mathtt{0} \qquad \mathtt{00}C\mathtt{1}BA\underbrace{C}\mathtt{0} \rightsquigarrow \mathtt{00}C\mathtt{1}BA\underbrace{\mathtt{0}C\mathtt{1}}\mathtt{0}$$

More generally, for any strings $x, z \in (\Sigma \cup \Gamma)^*$, we say that ***$z$ derives from x***, written $\boldsymbol{x \rightsquigarrow^* z}$, if we can transform x into z by applying a finite sequence of production rules, or more formally, if either

- $x = z$, or
- $x \rightsquigarrow y$ and $y \rightsquigarrow^* z$ for some string $y \in (\Sigma \cup \Gamma)^*$.

Straightforward definition-chasing implies that, for any strings $w, x, y, z \in (\sigma \cup \gamma)^*$, if $x \rightsquigarrow^* y$, then $wxz \rightsquigarrow^* wyz$.

The ***language $L(w)$*** of any string $w \in (\Sigma \cup \Gamma)^*$ is the set of all strings in Σ^* that derive from w:

$$L(w) := \{x \in \Sigma^* \mid w \rightsquigarrow^* x\}.$$

The language ***generated by*** a context-free grammar G, denoted $\boldsymbol{L(G)}$, is the language of its starting non-terminal. Finally, a language is ***context-free*** if it is generated by some context-free grammar.

Context-free grammars are sometimes used to model natural languages. In this context, the symbols are *words*, and the strings in the languages are *sentences*. For example, the following grammar describes a simple subset of English sentences. (Here I diverge from the usual notation conventions. Strings in ⟨angle brackets⟩ are non-terminals, and regular strings are terminals.)

⟨sentence⟩ → ⟨noun phrase⟩⟨verb phrase⟩⟨noun phrase⟩
⟨noun phrase⟩ → ⟨adjective phrase⟩⟨noun⟩
⟨adj. phrase⟩ → ⟨article⟩ | ⟨possessive⟩ | ⟨adjective phrase⟩⟨adjective⟩
⟨verb phrase⟩ → ⟨verb⟩ | ⟨adverb⟩⟨verb phrase⟩

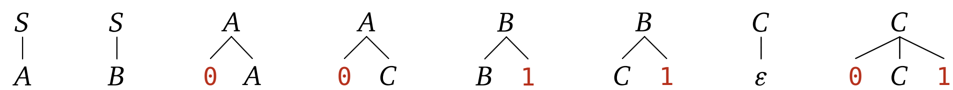
⟨noun⟩ → dog | trousers | daughter | nose | homework | time lord | pony | ⋯
⟨article⟩ → the | a | some | every | that | ⋯
⟨possessive⟩ → ⟨noun phrase⟩'s | my | your | his | her | ⋯
⟨adjective⟩ → friendly | furious | moist | green | severed | timey-wimey | little | ⋯
⟨verb⟩ → ate | found | wrote | killed | mangled | saved | invented | broke | ⋯
⟨adverb⟩ → squarely | incompetently | barely | sort of | awkwardly | totally | ⋯

5.2 Parse Trees

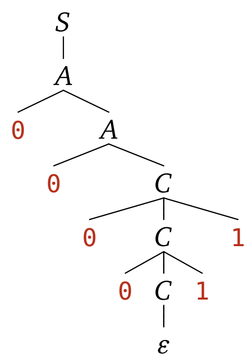
It is often useful to visualize derivations of strings in $L(G)$ using a ***parse tree***. The parse tree for a string $w \in L(G)$ is a rooted ordered tree where

- Each leaf is labeled with a terminal or the empty string ε. Concatenating these in order from left to right yields the string w.
- Each internal node is labeled with a non-terminal. In particular, the root is labeled with the start non-terminal S.
- For each internal node v, there is a production rule $A \to \omega$ where A is the label of v and the symbols in ω are the labels of the children of v in order from left to right.

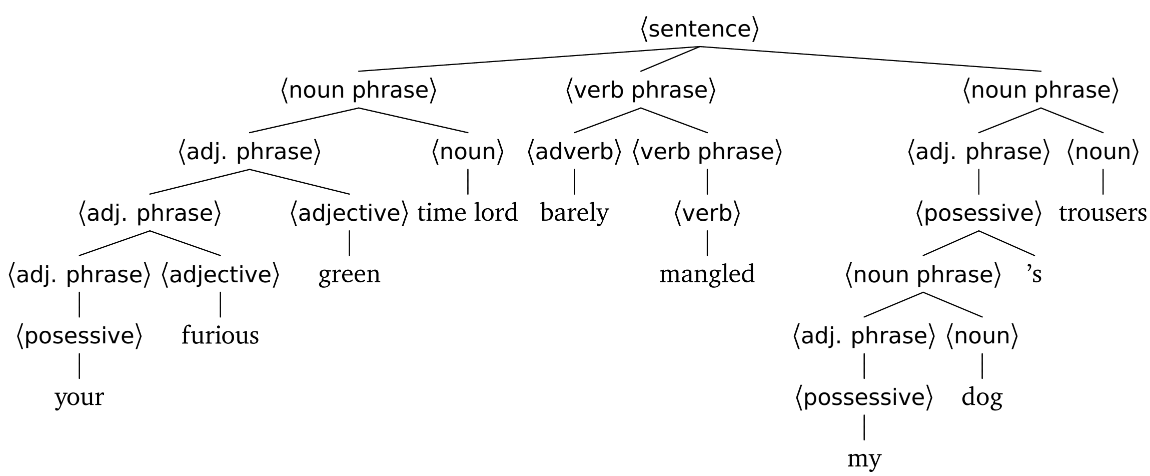
In other words, the production rules of the grammar describe *template trees* that can be assembled into larger parse trees. For example, the simple grammar on the previous page has the following templates, one for each production rule:



The same grammar gives us the following parse tree for the string `000011`:

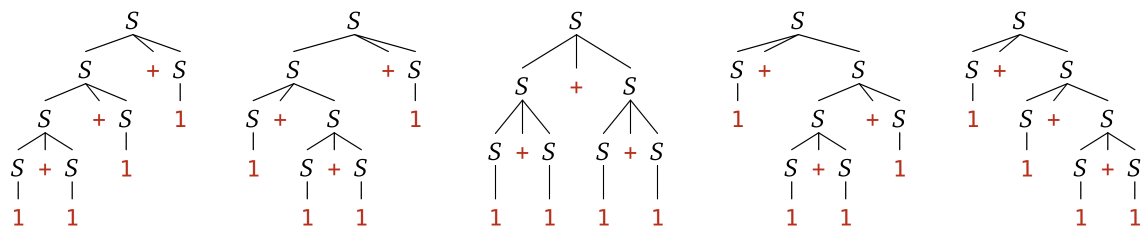


Our more complicated "English" grammar gives us parse trees like the following:



Any parse tree that contains at least one node with more than one non-terminal child corresponds to several different derivations. For example, when deriving an "English" sentence, we have a choice of whether to expand the first ⟨noun phrase⟩ ("your furious green time lord") before or after the second ("my dog's trousers").

A string w is ***ambiguous*** with respect to a grammar if there is more than one parse tree for w, and a grammar G is ***ambiguous*** is some string is ambiguous with respect to G. Neither of the previous example grammars is ambiguous. However, the grammar $S \to \texttt{1} \mid S\texttt{+}S$ is ambiguous, because the string `1+1+1+1` has five different parse trees:



A context-free language L is ***inherently ambiguous*** if every context-free grammar that generates L is ambiguous. The language generated by the previous grammar (the regular language $(\texttt{1+})^*\texttt{1}$) is *not* inherently ambiguous, because the unambiguous grammar $S \to \texttt{1} \mid \texttt{1+}S$ generates the same language.

5.3 From Grammar to Language

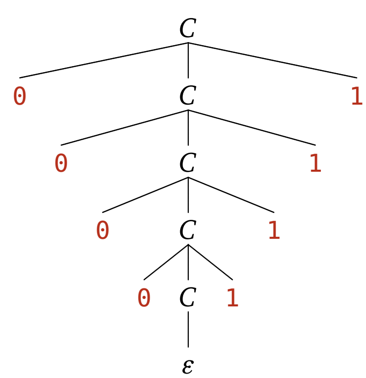
Let's figure out the language generated by our first example grammar

$$S \to A \mid B \qquad A \to \texttt{0}A \mid \texttt{0}C \qquad B \to B\texttt{1} \mid C\texttt{1} \qquad C \to \varepsilon \mid \texttt{0}C\texttt{1}.$$

Since the production rules for non-terminal C do not refer to any other non-terminal, let's begin by figuring out $L(C)$. After playing around with the smaller grammar $C \to \varepsilon \mid \texttt{0}C\texttt{1}$ for a few seconds, you can probably guess that its language is $\{\varepsilon, \texttt{01}, \texttt{0011}, \texttt{000111}, \ldots\}$, that is, the set all of strings of the form $\texttt{0}^n\texttt{1}^n$ for some integer n. For example, we can derive the string `00001111` from the start non-terminal S using the following derivation:

$$C \leadsto \texttt{0}C\texttt{1} \leadsto \texttt{00}C\texttt{11} \leadsto \texttt{000}C\texttt{111} \leadsto \texttt{0000}C\texttt{1111} \leadsto \texttt{0000}\varepsilon\texttt{1111} = \texttt{00001111}$$

The same derivation can be viewed as the following parse tree:



In fact, it is not hard to *prove* by induction that $L(C) = \{\texttt{0}^n\texttt{1}^n \mid n \ge 0\}$ as follows. As usual when we prove that two sets X and Y are equal, the proof has two stages: one stage to prove $X \subseteq Y$, the other to prove $Y \subseteq X$.

- First we prove that $C \rightsquigarrow^* \mathtt{0}^n\mathtt{1}^n$ for every non-negative integer n.

  Fix an arbitrary non-negative integer n. Assume that $C \rightsquigarrow^* \mathtt{0}^k\mathtt{1}^k$ for every non-negative integer $k < n$. There are two cases to consider.

  - If $n = 0$, then $\mathtt{0}^n\mathtt{1}^n = \varepsilon$. The rule $C \to \varepsilon$ implies that $C \rightsquigarrow \varepsilon$ and therefore $C \rightsquigarrow^* \varepsilon$.
  - Suppose $n > 0$. The inductive hypothesis implies that $C \rightsquigarrow^* \mathtt{0}^{n-1}\mathtt{1}^{n-1}$. Thus, the rule $C \to \mathtt{0}C\mathtt{1}$ implies that $C \rightsquigarrow \mathtt{0}C\mathtt{1} \rightsquigarrow^* \mathtt{0}(\mathtt{0}^{n-1}\mathtt{1}^{n-1})\mathtt{1} = \mathtt{0}^n\mathtt{1}^n$.

  In both cases, we conclude that that $C \rightsquigarrow^* \mathtt{0}^n\mathtt{1}^n$, as claimed.

- Next we prove that for every string $w \in \Sigma^*$ such that $C \rightsquigarrow^* w$, we have $w = \mathtt{0}^n\mathtt{1}^n$ for some non-negative integer n.

  Fix an arbitrary string w such that $C \rightsquigarrow^* w$. Assume that for any string x such that $|x| < |w|$ and $C \rightsquigarrow^* x$, we have $x = \mathtt{0}^k\mathtt{1}^k$ for some non-negative integer k. There are two cases to consider, one for each production rule.

  - If $w = \varepsilon$, then $w = \mathtt{0}^0\mathtt{1}^0$.
  - Suppose $w = \mathtt{0}x\mathtt{1}$ for some string x such that $C \rightsquigarrow^* x$. Because $|x| = |w| - 2 < |w|$, the inductive hypothesis implies that $x = \mathtt{0}^k\mathtt{1}^k$ for some integer k. Then we have $w = \mathtt{0}^{k+1}\mathtt{1}^{k+1}$.

  In both cases, we conclude that that $w = \mathtt{0}^n\mathtt{1}^n$ for some non-negative integer n, as claimed.

The first proof uses induction on strings, following the boilerplate proposed in the previous lecture; in particular, the case analysis mirrors the recursive definition of "string". The second proof uses *structural induction* on the grammar; the case analysis mirrors the recursive definition of the language of S, as described by the production rules. In both proofs, the inductive hypothesis is "Assume there is no smaller counterexample."

Similar analysis implies that $L(A) = \{\mathtt{0}^m\mathtt{1}^n \mid m > n\}$ and $L(B) = \{\mathtt{0}^m\mathtt{1}^n \mid m < n\}$, and therefore $L(S) = \{\mathtt{0}^m\mathtt{1}^n \mid m \neq n\}$.

5.4 More Examples

★★★

Give three or four examples of simple but interesting context-free grammars. Some possibilities:

- Same number of 0s and 1s
- Different number of 0s and 1s
- Palindromes
- Balanced parentheses
- Arithmetic/algebraic expressions
- Regular expressions

5.5 Regular Languages are Context-Free

The following inductive argument proves that every regular language is also a context-free language. Let L be an arbitrary regular language, encoded by some regular expression R. Assume that any regular expression shorter than R represents a context-free language. ("Assume no smaller counterexample.") We construct a context-free grammar for L as follows. There are several cases to consider.

- Suppose L is empty. Then L is generated by the trivial grammar $S \to S$.

- Suppose $L = \{w\}$ for some string $w \in \Sigma^*$. Then L is generated by the grammar $S \to w$.

- Suppose L is the union of some regular languages L_1 and L_2. The inductive hypothesis implies that L_1 and L_2 are context-free. Let G_1 be a context-free language for L_1 with starting non-terminal S_1, and let G_2 be a context-free language for L_2 with starting non-terminal S_2, where the non-terminal sets in G_1 and G_2 are disjoint. Then $L = L_1 \cup L_2$ is generated by the production rule $S \to S_1 \mid S_2$.

- Suppose L is the concatenation of some regular languages L_1 and L_2. The inductive hypothesis implies that L_1 and L_2 are context-free. Let G_1 be a context-free language for L_1 with starting non-terminal S_1, and let G_2 be a context-free language for L_2 with starting non-terminal S_2, where the non-terminal sets in G_1 and G_2 are disjoint. Then $L = L_1 L_2$ is generated by the production rule $S \to S_1 S_2$.

- Suppose L is the Kleene closure of some regular language L_1. The inductive hypothesis implies that L_1 is context-free. Let G_1 be a context-free language for L_1 with starting non-terminal S_1. Then $L = L_1^*$ is generated by the production rule $S \to \varepsilon \mid S_1 S$.

In every case, we have found a context-free grammar that generates L, which means L is context-free.

In the next lecture note, we will prove that the context-free language $\{\mathtt{0}^n\mathtt{1}^n \mid n \geq 0\}$ is not regular. (In fact, this is the *canonical example* of a non-regular language.) Thus, context-free grammars are strictly more expressive than regular expressions.

5.6 Not Every Language is Context-Free

Again, you may be tempted to conjecture that *every* language is context-free, but a variant of our earlier cardinality argument implies that this is not the case.

Any context-free grammar over the alphabet Σ can be encoded as a string over the alphabet $\Sigma \cup \Gamma \cup \{\mathtt{Ɛ}, \mathtt{\to}, \mathtt{|}, \mathtt{\$}\}$, where $\mathtt{\$}$ indicates the end of the production rules for each non-terminal. For example, our example grammar

$$S \to A \mid B \qquad A \to \mathtt{0}A \mid \mathtt{0}C \qquad B \to B\mathtt{1} \mid C\mathtt{1} \qquad C \to \varepsilon \mid \mathtt{0}C\mathtt{1}$$

can be encoded as the string

```
S→A|B$A→0A|0C$B→B1|C1$C→Ɛ|0C1$
```

We can further encode any such string as a *binary* string by associating each symbol in the set $\Sigma \cup \Gamma \cup \{\mathtt{Ɛ}, \mathtt{\to}, \mathtt{|}, \mathtt{\$}\}$ with a different binary substring. Specifically, if we encode each of the grammar symbols $\mathtt{Ɛ}, \mathtt{\to}, \mathtt{|}, \mathtt{\$}$ as a string of the form $\mathtt{11}^*\mathtt{0}$, each terminal in Σ as a string of the form $\mathtt{011}^*\mathtt{0}$, and each non-terminal as a string of the form $\mathtt{0011}^*\mathtt{0}$, we can unambiguously recover the grammar from the encoding. For example, applying the code

Ɛ ↦ 10	0 ↦ 010	S ↦ 0010
→ ↦ 110	1 ↦ 0110	A ↦ 00110
| ↦ 1110		B ↦ 001110
$ ↦ 11110		C ↦ 0011110

transforms our example grammar into the 135-bit string

```
00101100011011100011101111000110
11001000110111001000111101111000
11101100011100110111000111100110
11110001111011010111001000111100
1011110.
```

Adding a 1 to the start of this bit string gives us the binary encoding of the integer

$$51\,115\,617\,766\,581\,763\,757\,672\,062\,401\,233\,529\,937\,502.$$

Our construction guarantees that two different context-free grammars over the same language (ignoring changing the names of the non-terminals) yield different positive integers. Thus, the set of context-free grammars over any alphabet is *at most* as large as the set of integers, and is therefore countably infinite. (Most integers are not encodings of context-free grammars, but that only helps us.) It follows that the set of all context-free *languages* over any fixed alphabet is also countably infinite. But we already showed that the set of *all* languages over any alphabet is uncountably infinite. So almost all languages are non-context-free!

Although we will probably not see them in this course, there are techniques for proving that certain languages are not context-free, just as there are for proving certain languages are not regular. In particular, the $\{0^n1^n0^n \mid n \geq 0\}$ is not context-free. (In fact, this is the *canonical example* of a non-context-free language.)

⋆5.7 Recursive Automata

All the flavors of finite-state automata we have seen so far describe/encode/accept/compute *regular* languages; these are precisely the languages that can be constructed from individual strings by union, concatenation, and unbounded repetition. Just as context-free grammars are recursive generalizations of regular expressions, we can define a class of machines called *recursive automata*, which generalize (nondeterministic) finite-state automata.

Formally, a ***recursive automaton*** consists of the following components:

- A non-empty finite set Σ, called the ***input alphabet***
- Another non-empty finite set N, disjoint from Σ, whose elements are called ***module names***
- A ***start name*** $S \in N$
- A set $M = \{M_A \mid A \in N\}$ of NFAs over the alphabet $\Sigma \cup N$ called ***modules***, each with a single accepting state. Each module M_A has the following components:
  - A finite set Q_A of ***states***, such that $Q_A \cap Q_B = \varnothing$ for all $A \neq B$
  - A ***start*** state $s_A \in Q_A$
  - A ***terminal*** or ***accepting*** state $t_A \in Q_A$
  - A ***transition function*** $\delta_A \colon Q_A \times (\Sigma \cup \{\varepsilon\} \cup N) \to 2^{Q_A}$.

Equivalently, we have a single global transition function $\delta \colon Q \times (\Sigma \cup \{\varepsilon\} \cup N) \to 2^Q$, where $Q = \bigcup_{A \in N} Q_A$, such that for any name A and any state $q \in Q_A$ we have $\delta(q) \subseteq Q_A$. Machine M_S is called the ***main module***.

A ***configuration*** of a recursive automaton is a triple (w, q, s), where w is a string in Σ^* called the ***input***, q is a state in Q called the ***local state***, and σ is a string in Q^* called the ***stack***. The module containing the local state q is called the ***active module***. A configuration can be changed by three types of transitions.

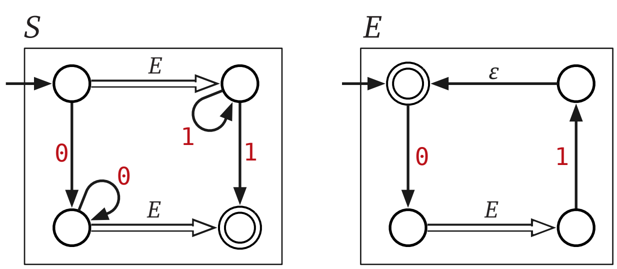
- A ***read*** transition consumes the first symbol in the input and changes the local state within the current module, just like a standard NFA.
- An ***epsilon*** transition changes the local state within the current module, without consuming any input characters, just like a standard NFA.
- A ***call*** transition chooses an arbitrary name A, changes the current state q_0 to some state in $\delta(q,A)$, and pushes the corresponding start state s_A onto the stack (thereby changing the active module to M_A), without consuming any input characters.
- Finally, if the current state is the terminal state of some module *and* the stack is non-empty, a ***return*** transition pops the top state off the stack and makes it the new local state (thereby possibly changing the active module), without consuming any input characters.

Symbolically, we can describe these transitions as follows:

$$\begin{array}{rcl}
\textbf{read:} & (ax,q,\sigma) \longmapsto (x,q',\sigma) & \text{for some } q' \in \delta(q,a) \\
\textbf{epsilon:} & (w,q,\sigma) \longmapsto (w,q',\sigma) & \text{for some } q' \in \delta(q,\varepsilon) \\
\textbf{call:} & (w,q,\sigma) \longmapsto (w,s_A,q'\cdot\sigma) & \text{for some } A \in N \text{ and } q' \in \delta(q,A) \\
\textbf{return:} & (w,t_A,q\cdot\sigma) \longmapsto (w,q,\sigma) &
\end{array}$$

A recursive automaton ***accepts*** a string w if there is a *finite* sequence of transitions starting at the start configuration (w, s_S, ε) and ending at the terminal configuration $(\varepsilon, t_S, \varepsilon)$.

For example, the following recursive automaton accepts the language $\{0^m1^n \mid m \neq n\}$. The recursive automaton has two component machines; the start machine named S and a "subroutine" named E (for "equal") that accepts the language $\{0^n1^n \mid n \geq 0\}$. White arrows indicate recursive transitions.



A recursive automaton for the language $\{0^m1^n \mid m \neq n\}$

Lemma 5.1. *Every context-free language is accepted by a recursive automaton.*

Proof:

★★★ Direct construction from the CFG, with one module per nonterminal. □

For example, the context-free grammar

$$\begin{aligned}
S &\rightarrow 0A \mid B1 \\
A &\rightarrow 0A \mid E \\
B &\rightarrow B1 \mid E \\
E &\rightarrow \varepsilon \mid 0E0
\end{aligned}$$

leads to the following recursive automaton with four modules:

★★★ Figure!

Lemma 5.2. *Every recursive automaton accepts a context-free language.*

Proof (sketch): Let $R = (\Sigma, N, S, \delta, M)$ be an arbitrary recursive automaton. We define a context-free grammar G that describes the language accepted by R as follows.

The set of nonterminals in G is isomorphic the state set Q; that is, for each state $q \in Q$, the grammar contains a corresponding nonterminal $[q]$. The language of $[q]$ will be the set of strings w such that there is a finite sequence of transitions starting at the start configuration (w, q, ε) and ending at the terminal configuration $(\varepsilon, t, \varepsilon)$, where t is the terminal state of the module containing q.

The grammar has four types of production rules, corresponding to the four types of transitions:

- ***read:*** For each symbol a and each pair of states p and q such that $p \in \delta(q, a)$, the grammar contains the production rule $[q] \to a[p]$.
- ***epsilon:*** For any two states p and q such that $p \in \delta(q, \varepsilon)$, the grammar contains the production rule $[q] \to [p]$.
- ***call:*** Each name A and each pair of states states p and q such that $p \in \delta(q, A)$, the grammar contains the production rule $[q] \to [s_A][p]$.
- ***return:*** Each name A, the grammar contains the production rule $[t_A] \to \varepsilon$.

Finally, the starting nonterminal of G is $[s_S]$, which corresponds to the start state of the main module.

We can now argue inductively that the grammar G and the recursive automaton R describe the same language. Specifically, any sequence of transitions in R from (w, s_S, ε) to $(\varepsilon, t_S, \varepsilon)$ can be transformed mechanically into a derivation of w from the nonterminal $[s_S]$ in G. Symmetrically, the ***leftmost*** derivation of any string w in G can be mechanically transformed into an accepting sequence of transitions in R. We omit the straightforward but tedious details. □

For example, the recursive automaton on the previous page gives us the following context-free grammar. To make the grammar more readable, I've renamed the nonterminals corresponding to start and terminal states: $S = [s_S]$, $T = [t_S]$, and $E = [s_E] = [t_E]$:

$$
\begin{array}{ll}
S \to EA \mid 0B & \qquad E \to \varepsilon \mid 0X \\
A \to 1A \mid 1T & \qquad X \to EY \\
B \to 0B \mid ET & \qquad Y \to 1Z \\
T \to \varepsilon & \qquad Z \to E
\end{array}
$$

Our earlier proofs imply that we can forbid ε-transitions or even allow regular-expression transitions in our recursive automata without changing the set of languages they accept.

*5.8 Chomsky Normal Form

For many algorithmic problems involving context-free grammars, it is helpful to consider grammars with a particular special structure called ***Chomsky normal form***, abbreviated ***CNF***:

- The starting non-terminal S does not appear on the right side of any production rule.

- The starting non-terminal S *may* have the production rule $S \to \varepsilon$.
- The right side of every other production rule is either a single terminal symbol or a string of exactly two non-terminals—that is, every other production rule has the form $A \to BC$ or $A \to a$.

A particularly attractive feature of CNF grammars is that they yield *full binary* parse trees; in particular, every parse tree for a string of length $n > 0$ has exactly $2n - 1$ non-terminal nodes. Consequently, any string of length n in the language of a CNF grammar can be derived in exactly $2n - 1$ production steps. It follows that we can actually determine whether a string belongs to the language of a CNF grammar by brute-force consideration of all possible derivations of the appropriate length.

For arbitrary context-free grammars, there is no similar upper bound on the length of a derivation, and therefore no similar brute-force membership algorithm, because the grammar may contain additional ***ε-productions*** of the form $A \to \varepsilon$ and/or ***unit productions*** of the form $A \to B$, where both A and B are non-terminals. Unit productions introduce nodes of degree 1 into any parse tree, and ε-productions introduce leaves that do not contribute to the word being parsed.

Fortunately, it *is* possible to determine membership in the language of an arbitrary context-free grammar, thanks to the following theorem. Two context-free grammars are ***equivalent*** if they define the same language.

Every context-free grammar is equivalent to a grammar in Chomsky normal form.

To be more specific, define the ***total length*** of a context-free grammar to be the number of symbols needed to write down the grammar; up to constant factors, the total length is the sum of the lengths of the production rules.

Theorem 5.3. *For every context-free grammar with total length L, there is an equivalent grammar in Chomsky normal form with total length $O(L^2)$, which can be computed in $O(L^2)$ time.*

Converting an arbitrary grammar into Chomsky normal form is a complex task. Fortunately, for most applications of context-free grammars, it's enough to know that the algorithm exists. For the sake of completeness, however, I will describe one such conversion algorithm here. This algorithm consists of several relatively straightforward stages. Efficient implementation of some of these stages requires standard graph-traversal algorithms, which we will describe much later in the course.

0. Add a new starting non-terminal. Add a new non-terminal S' and a production rule $S' \to S$, where S is the starting non-terminal for the given grammar. S' will be the starting non-terminal for the resulting CNF grammar. (In fact, this step is necessary only when $S \leadsto^* \varepsilon$, but at this point in the conversion process, we don't yet know whether that's true.)

1. Decompose long production rules. For each production rule $A \to \omega$ whose right side w has length greater than two, add new production rules of length two that still permit the derivation $A \leadsto^* \omega$. Specifically, suppose $\omega = \alpha\chi$ for some symbol $\alpha \in \Sigma \cup \Gamma$ and string $\chi \in (\Sigma \cup \Gamma)^*$. The algorithm replaces $A \to \omega$ with two new production rules $A \to \alpha B$ and $B \to \chi$, where B is a new non-terminal, and then (if necessary) recursively decomposes the production rule $B \to \chi$. For example, we would replace the long production rule $A \to \mathtt{0}BC\mathtt{1}CB$ with the following sequence of short production rules, where each A_i is a new non-terminal:

$$A \to \mathtt{0}A_1 \qquad A_1 \to BA_2 \qquad A_2 \to CA_3 \qquad A_3 \to \mathtt{1}A_4 \qquad A_4 \to CB$$

This stage can significantly increase the number of non-terminals and production rules, but it increases the *total length* of all production rules by at most a small constant factor.[2] Moreover, for the remainder of the conversion algorithm, every production rule has length at most two. The running time of this stage is $O(L)$.

2. Identify nullable non-terminals. A non-terminal A is ***nullable*** if and only if $A \rightsquigarrow^* \varepsilon$. The recursive definition of $\rightsquigarrow^*$ implies that A is nullable if and only if the grammar contains a production rule $A \to \omega$ where ω consists entirely of nullable non-terminals (in particular, if $\omega = \varepsilon$). You may be tempted to transform this recursive characterization directly into a recursive algorithm, but this is a bad idea; the resulting algorithm would fall into an infinite loop if (for example) the same non-terminal appeared on both sides of the same production rule. Instead, we apply the following ***fixed-point*** algorithm, which repeatedly scans through the entire grammar until a complete scan discovers no new nullable non-terminals.

```
NULLABLES(Σ, Γ, R, S):
  Γ_ε ← ∅        ⟨⟨known nullable non-terminals⟩⟩
  done ← FALSE
  while ¬done
      done ← TRUE
      for each non-terminal A ∈ Γ \ Γ_ε
          for each production rule A → ω
              if ω ∈ Γ_ε*
                  add A to Γ_ε
                  done ← FALSE
  return Γ_ε
```

At this point in the conversion algorithm, if S' is ***not*** identified as nullable, then we can safely remove it from the grammar and use the original starting nonterminal S instead.

As written, NULLABLES runs in $O(nL) = O(L^2)$ time, where n is the number of non-terminals in Γ. Each iteration of the main loop except the last adds at least one non-terminal to Γ_ε, so the algorithm halts after at most $n+1 \le L$ iterations, and in each iteration, we examine at most L production rules. There is a faster implementation of NULLABLES that runs in $O(n+L) = O(L)$ time,[3] but since other parts of the conversion algorithm already require $O(L^2)$ time, we needn't bother.

3. Eliminate ε-productions. First, remove every production rule of the form $A \to \varepsilon$. Then for each production rule $A \to w$, add all possible new production rules of the form $A \to w'$, where w'

[2]In most textbook descriptions of this conversion algorithm, this stage is performed *last*, after removing ε-productions and unit productions. But with the stages in that traditional order, removing ε-productions could *exponentially* increase the length of the grammar in the worst case! Consider the production rule $A \to (BC)^k$, where B is nullable but C is not. Decomposing this rule first and then removing ε-productions introduces about $3k$ new rules; whereas, removing ε-productions first introduces 2^k new rules, most of which then must then be further decomposed.

[3]Consider the bipartite graph whose vertices correspond to non-terminals and the right sides of production rules, with one edge per rule. The faster algorithm is a modified breadth-first search of this graph, starting at the vertex representing ε.

is a ***non-empty*** string obtained from w by removing one nullable non-terminal. For example, if if the grammar contained the production rule $A \to BC$, where B and C are both nullable, we would add two new production rules $A \to B \mid C$. Finally, if the starting nonterminal S' was identified as nullable in the previous stage, add the production rule $S' \to \varepsilon$; this will be the *only* ε-production in the final grammar. This phase of the conversion runs in $O(L)$ time and at most triples the number of production rules.

4. Merge equivalent non-terminals. We say that two non-terminals A and B are ***equivalent*** if they can be derived from each other: $A \rightsquigarrow^* B$ and $B \rightsquigarrow^* A$. Because we have already removed ε-productions, any such derivation must consist entirely of unit productions. For example, in the grammar

$$S \to B \mid C, \qquad A \to B \mid D \mid CC \mid \mathtt{0}, \qquad B \to C \mid AD \mid \mathtt{1}, \qquad C \to A \mid DA, \qquad D \to BA \mid CS,$$

non-terminals A, B, C are all equivalent, but S is not in that equivalence class (because we cannot derive S from A) and neither is D (because we cannot derive A from D).

Construct a directed graph G whose vertices are the non-terminals and whose edges correspond to unit productions, in $O(L)$ time. Then two non-terminals are equivalent if and only if they are in the same strong component of G. Compute the strong components of G in $O(L)$ time using, for example, the algorithm of Kosaraju and Sharir. Then merge all the non-terminals in each equivalence class into a single non-terminal. Finally, remove any unit productions of the form $A \to A$. The total running time for this phase is $O(L)$. Starting with our example grammar above, merging B and C with A and removing the production $A \to A$ gives us the simpler grammar

$$S \to A, \qquad A \to AA \mid D \mid DA \mid \mathtt{0} \mid \mathtt{1}, \qquad D \to AA \mid AS.$$

We could further simplify the grammar by merging all non-terminals reachable from S using only unit productions (in this case, merging non-terminals S and S), but this further simplification is unnecessary.

5. Remove unit productions. Once again, we construct a directed graph G whose vertices are the non-terminals and whose edges correspond to unit productions, in $O(L)$ time. Because no two non-terminals are equivalent, G is acyclic. Thus, using topological sort, we can index the non-terminals $A_1, A_2, \ldots, A_n$ such that for every unit production $A_i \to A_j$ we have $i < j$, again in $O(L)$ time; moreover, we can assume that the starting non-terminal is A_1. (In fact, both the dag G and the linear ordering of non-terminals was already computed in the previous phase!!)

Then for each index j in decreasing order, for each unit production $A_i \to A_j$ and each production $A_j \to \omega$, we add a new production rule $A_i \to \omega$. At this point, all unit productions are redundant and can be removed. Applying this algorithm to our example grammar above gives us the grammar

$$S \to AA \mid AS \mid DA \mid \mathtt{0} \mid \mathtt{1}, \qquad A \to AA \mid AS \mid DA \mid \mathtt{0} \mid \mathtt{1}, \qquad D \to AA \mid AS.$$

In the worst case, each production rule for A_n is copied to each of the other $n-1$ non-terminals. Thus, this phase runs in $\Theta(nL) = O(L^2)$ time and increases the length of the grammar to $\Theta(nL) = O(L^2)$ in the worst case.

This phase dominates the running time of the CNF conversion algorithm. Unlike previous phases, no faster algorithm for removing unit transformations is known! There are grammars of length L with unit productions such that any equivalent grammar without unit productions has length $\Omega(L^{1.499999})$ (for any desired number of 9s), but this lower bound does not rule out the possibility of an algorithm that runs in only $O(L^{3/2})$ time. Closing the gap between $\Omega(L^{3/2-\varepsilon})$ and $O(L^2)$ has been an open problem since the early 1980s.

6. Protect terminals. Finally, for each terminal $a \in \Sigma$, we introduce a new non-terminal A_a and a new production rule $A_a \to a$, and then replace a with A_a in every production rule of length 2. This completes the conversion to Chomsky normal form. As claimed, the total running time of the algorithm is $O(L^2)$, and the total length of the output grammar is also $O(L^2)$.

CNF Conversion Example

As a running example, let's apply these stages one at a time to our first example grammar.

$$S \to A \mid B \qquad A \to 0A \mid 0C \qquad B \to B1 \mid C1 \qquad C \to \varepsilon \mid 0C1$$

0. Add a new starting non-terminal S'.

$$\underline{S' \to S} \qquad S \to A \mid B \qquad A \to 0A \mid 0C \qquad B \to B1 \mid C1 \qquad C \to \varepsilon \mid 0C1$$

1. Decompose the long production rule $C \to 0C1$.

$$S' \to S \qquad S \to A \mid B \qquad A \to 0A \mid 0C \qquad B \to B1 \mid C1 \qquad \underline{C \to \varepsilon \mid 0D \qquad D \to C1}$$

2. Identify C as the only nullable non-terminal. Because S' is not nullable, remove the production rule $S' \to S$.

3. Eliminate the ε-production $C \to \varepsilon$.

$$S \to A \mid B \qquad A \to 0A \mid 0C \,\underline{\mid 0} \qquad B \to B1 \mid C1 \,\underline{\mid 1} \qquad C \to 0D \qquad D \to C1 \,\underline{\mid 1}$$

4. No two non-terminals are equivalent, so there's nothing to merge.

5. Remove the unit productions $S' \to S$, $S \to A$, and $S \to B$.

$$S \to \underline{0A \mid 0C \mid B1 \mid C1 \mid 0 \mid 1}$$
$$A \to 0A \mid 0C \mid 0 \qquad B \to B1 \mid C1 \mid 1 \qquad C \to 0D \qquad D \to C1 \mid 1.$$

6. Finally, protect the terminals 0 and 1 to obtain the final CNF grammar.

$$S \to \underline{EA \mid EC \mid BF \mid CF \mid 0 \mid 1}$$
$$A \to \underline{EA \mid EC} \mid 0 \qquad B \to \underline{BF \mid CF} \mid 1$$
$$C \to \underline{ED} \qquad D \to \underline{CF} \mid 1$$
$$\underline{E \to 0} \qquad \underline{F \to 1}$$

Exercises

1. Describe context-free grammars that generate each of the following languages. The function $\#(x, w)$ returns the number of occurrences of the ***substring*** x in the string w. For example, $\#(\texttt{0}, \texttt{101001}) = 3$ and $\#(\texttt{010}, \texttt{1010100011}) = 2$.

   (a) All strings in $\{\texttt{0}, \texttt{1}\}^*$ whose length is divisible by 5.

   (b) All strings in $\{\texttt{0}, \texttt{1}\}^*$ representing a non-negative multiple of 5 in binary.

   (c) $\{w \in \{\texttt{0}, \texttt{1}\}^* \mid \#(\texttt{0}, w) = \#(\texttt{1}, w)\}$

   (d) $\{w \in \{\texttt{0}, \texttt{1}\}^* \mid \#(\texttt{0}, w) \neq \#(\texttt{1}, w)\}$

   (e) $\{w \in \{\texttt{0}, \texttt{1}\}^* \mid \#(\texttt{00}, w) = \#(\texttt{11}, w)\}$

   (f) $\{w \in \{\texttt{0}, \texttt{1}\}^* \mid \#(\texttt{01}, w) = \#(\texttt{10}, w)\}$

   (g) $\{w \in \{\texttt{0}, \texttt{1}\}^* \mid \#(\texttt{0}, w) = \#(\texttt{1}, w)$ and $|w|$ is a multiple of 3$\}$

   (h) $\{\texttt{0}, \texttt{1}\}^* \setminus \{\texttt{0}^n\texttt{1}^n \mid n \geq 0\}$

   (i) $\{\texttt{0}^n\texttt{1}^{2n} \mid n \geq 0\}$

   (j) $\{\texttt{0}, \texttt{1}\}^* \setminus \{\texttt{0}^n\texttt{1}^{2n} \mid n \geq 0\}$

   (k) $\{\texttt{0}^n\texttt{1}^m \mid 0 \leq 2m \leq n < 3m\}$

   (l) $\{\texttt{0}^i\texttt{1}^j\texttt{2}^{i+j} \mid i, j \geq 0\}$

   (m) $\{\texttt{0}^i\texttt{1}^j\texttt{2}^k \mid i = j \text{ or } j = k\}$

   (n) $\{\texttt{0}^i\texttt{1}^j\texttt{2}^k \mid i \neq j \text{ or } j \neq k\}$

   (o) $\{\texttt{0}^i\texttt{1}^j\texttt{0}^j\texttt{1}^i \mid i, j \geq 0\}$

   (p) $\left\{w\texttt{\$}\texttt{0}^{\#(\texttt{0},w)} \,\middle|\, w \in \{\texttt{0}, \texttt{1}\}^*\right\}$

   (q) $\{xy \mid x, y \in \{\texttt{0}, \texttt{1}\}^* \text{ and } x \neq y \text{ and } |x| = |y|\}$

   (r) $\left\{x\texttt{\$}y^R \,\middle|\, x, y \in \{\texttt{0}, \texttt{1}\}^* \text{ and } x \neq y\right\}$

   (s) $\{x\texttt{\$}y \mid x, y \in \{\texttt{0}, \texttt{1}\}^* \text{ and } \#(\texttt{0}, x) = \#(\texttt{1}, y)\}$

   (t) $\{\texttt{0}, \texttt{1}\}^* \setminus \{ww \mid w \in \{\texttt{0}, \texttt{1}\}^*\}$

   (u) All strings in $\{\texttt{0}, \texttt{1}\}^*$ that are *not* palindromes.

   (v) All strings in $\{\texttt{(}, \texttt{)}, \diamond\}^*$ in which the parentheses are balanced and the symbol $\diamond$ appears at most four times. For example, `()(())` and `(◇◇(()()◇)()())◇` and `◇◇◇` are strings in this language, but `)(()` and `(◇◇◇)◇◇` are not.

2. Describe recursive automata for each of the languages in problem 1. ("Describe" does not necessarily mean "draw"!)

3. Prove that if L is a context-free language, then L^R is also a context-free language. *[Hint: How do you reverse a context-free grammar?]*

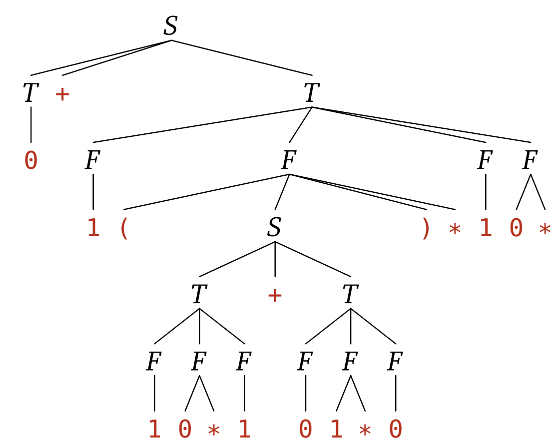
4. Consider a generalization of context-free grammars that allows any *regular expression* over $\Sigma \cup \Gamma$ to appear on the right side of a production rule. Without loss of generality, for each non-terminal $A \in \Gamma$, the generalized grammar contains a single regular expression $R(A)$. To apply a production rule to a string, we replace any non-terminal A with an arbitrary word

in the language described by $R(A)$. As usual, the language of the generalized grammar is the set of all strings that can be derived from its start non-terminal.

For example:, the following generalized context-free grammar describes the language of all regular expressions over the alphabet $\{\mathtt{0}, \mathtt{1}\}$:

$$S \to (T\,\mathtt{+})^* T + \mathtt{Ø} \qquad \text{(Regular expressions)}$$
$$T \to \mathtt{\varepsilon} + F^* F \qquad \text{(Terms = summable expressions)}$$
$$F \to (\mathtt{0} + \mathtt{1} + \mathtt{(}S\mathtt{)})(\mathtt{*} + \varepsilon) \qquad \text{(Factors = concatenable expressions)}$$

Here is a parse tree for the regular expression `0+1(10*1+01*0)*10*` (which represents the set of all binary numbers divisible by 3):



Prove that every *generalized* context-free grammar describes a context-free language. In other words, show that allowing regular expressions to appear in production rules does not increase the expressive power of context-free grammars.

Think globally, act locally.

— Attributed to Patrick Geddes (c.1915), among many others.

We can only see a short distance ahead,
but we can see plenty there that needs to be done.

— Alan Turing, "Computing Machinery and Intelligence" (1950)

Never worry about theory as long as the machinery does what it's supposed to do.

— Robert Anson Heinlein, *Waldo & Magic, Inc.* (1950)

6 Turing Machines

In 1936, a few months before his 24th birthday, Alan Turing launched computer science as a modern intellectual discipline. In a single remarkable paper, Turing provided the following results:

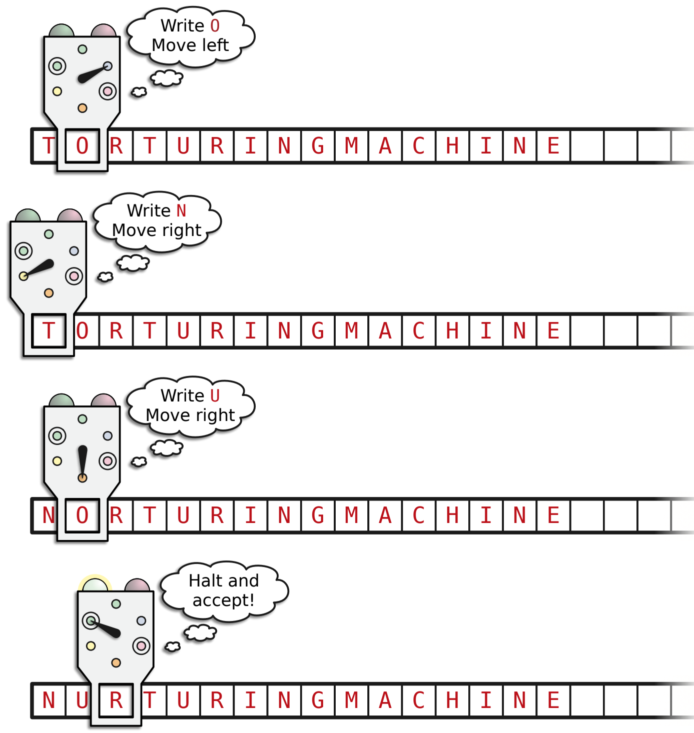
- A simple formal model of mechanical computation now known as *Turing machines*.
- A description of a single *universal* machine that can be used to compute *any* function computable by *any* other Turing machine.
- A proof that no Turing machine can solve the *halting problem*—Given the formal description of an arbitrary Turing machine M, does M halt or run forever?
- A proof that no Turing machine can determine whether an arbitrary given proposition is provable from the axioms of first-order logic. This Hilbert and Ackermann's famous *Entscheidungsproblem* ("decision problem")
- Compelling arguments[1] that his machines can execute *arbitrary* "calculation by finite means".

Turing's paper was not the first to prove that the *Entscheidungsproblem* had no algorithmic solution. Alonzo Church published the first proof just a new months earlier, using a very different model of computation, now called the ***untyped λ-calculus***. Turing and Church developed their results independently; indeed, Turing rushed the submission of his own paper immediately after receiving a copy of Church's paper, pausing only long enough to prove that any function computable via λ-calculus can also be computed by a Turing machine and vice versa. Church was the referee for Turing's paper; between the paper's submission and its acceptance, Turing was admitted to Princeton, where he became Church's PhD student. He finished his PhD two years later.

Informally, Turing described a device with a finite number of ***internal states*** that has access to memory in the form of a ***tape***. The tape consists of a semi-infinite sequence of ***cells***, each

[1]As Turing put it, "All arguments which can be given are bound to be, fundamentally, appeals to intuition, and for this reason rather unsatisfactory mathematically." The claim that anything that can be computed can be computing using Turing machines is now known as the ***Church-Turing thesis***.

containing a single symbol from some arbitrary finite alphabet. The Turing machine can access the tape only through its ***head***, which is positioned over a single cell. Initially, the tape contains an arbitrary finite ***input string*** followed by an infinite sequence of ***blanks***, and the head is positioned over the first cell on the tape. In a single iteration, the machine reads the symbol in that cell, possibly write a new symbol into that cell, possibly changes its internal state, possibly moves the head to a neighboring cell, and possibly halts. The precise behavior of the machine at each iteration is entirely determined by its internal state and the symbol that it reads. When the machine halts, it indicates whether it has ***accepted*** or ***rejected*** the original input string.



A few iterations of a six-state Turing machine.

6.1 Why Bother?

Students used to thinking of computation in terms of higher-level operations like random memory accesses, function calls, and recursion may wonder why we should even consider a model as simple and constrained as Turing machines. Admittedly, Turing machines are a terrible model for thinking about ***fast*** computation; simple operations that take constant time in the standard random-access model can require *arbitrarily* many steps on a Turing machine. Worse, seemingly minor variations in the precise definition of "Turing machine" can have significant impact on problem complexity. As a simple example (which will make more sense later), we can reverse a string of n bits in $O(n)$ time using a two-tape Turing machine, but the same task provably requires $\Omega(n^2)$ time on a single-tape machine.

But here we are not interested in finding *fast* algorithms, or indeed in finding algorithms at all, but rather in proving that some problems cannot be solved by *any* computational means. Such a bold claim requires a formal definition of "computation" that is simple enough to support formal argument, but still powerful enough to describe arbitrary algorithms. Turing machines are ideal for this purpose. In particular, Turing machines are powerful enough to *simulate other*

Turing machines, while still simple enough to let us build up this self-simulation from scratch, unlike more complex but efficient models like the standard random-access machine

(Arguably, self-simulation is even simpler in Church's λ-calculus, or in Schönfinkel and Curry's combinator calculus, which is one of many reasons those models are more common in the design and analysis of programming languages than Turing machines. Those models much more abstract; in particular, they are harder to show equivalent to standard iterative models of computation.)

6.2 Formal Definitions

Formally, a Turing machine consists of the following components. (Hang on; it's a long list.)

- An arbitrary finite set Γ with at least two elements, called the ***tape alphabet***.
- An arbitrary symbol $\square \in \Gamma$, called the ***blank symbol*** or just the ***blank***.
- An arbitrary nonempty subset $\Sigma \subseteq (\Gamma \setminus \{\square\})$, called the ***input alphabet***.
- Another arbitrary finite set Q whose elements are called ***states***.
- Three distinct special states $\mathsf{start}, \mathsf{accept}, \mathsf{reject} \in Q$.
- A ***transition*** function $\delta : (Q \setminus \{\mathsf{accept}, \mathsf{reject}\}) \times \Gamma \to Q \times \Gamma \times \{-1, +1\}$.

A ***configuration*** or ***global state*** of a Turing machine is represented by a triple $(q, x, i) \in Q \times \Gamma^* \times \mathbb{N}$, indicating that the machine's internal state is q, the tape contains the string x followed by an infinite sequence of blanks, and the head is located at position i. Trailing blanks in the tape string are ignored; the triples (q, x, i) and $(q, x\square, i)$ describe exactly the same configuration.

The transition function δ describes the evolution of the machine. For example, $\delta(q, a) = (p, b, -1)$ means that when the machine reads symbol a in state q, it changes its internal state to p, writes symbol b onto the tape at its current location (replacing a), and then decreases its position by 1 (or more intuitively, moves one step to the left). If the position of the head becomes negative, no further transitions are possible, and the machine ***crashes***.

We write $\boldsymbol{(p, x, i) \Rightarrow_M (q, y, j)}$ to indicate that Turing machine M transitions from the first configuration to the second in one step. (The symbol $\Rightarrow$ is often pronounced "yields"; I will omit the subscript M if the machine is clear from context.) For example, $\delta(p, a) = (q, b, \pm 1)$ means that

$$(p, xay, i) \;\Rightarrow\; (q, xby, i \pm 1)$$

for any non-negative integer i, any string x of length i, and any string y. The evolution of any Turing machine is ***deterministic***; each configuration C yields a *unique* configuration C'. We write $\boldsymbol{C \Rightarrow^* C'}$ to indicate that there is a (possibly empty) sequence of transitions from configuration C to configuration C'. (The symbol $\Rightarrow^*$ can be pronounced "eventually yields".)

The initial configuration is $(w, \mathsf{start}, 0)$ for some arbitrary (and possibly empty) ***input string*** $w \in \Sigma^*$. If M eventually reaches the accept state—more formally, if $(w, \mathsf{start}, 0) \Rightarrow^* (x, \mathsf{accept}, i)$ for some string $x \in \Gamma^*$ and some integer i—we say that M ***accepts*** the original input string w. Similarly, if M eventually reaches the reject state, we say that M ***rejects*** w. We must emphasize that "rejects" and "does not accept" are *not* synonyms; if M crashes or runs forever, then M neither accepts nor rejects w.

We distinguish between two different senses in which a Turing machine can "accept" a language. Let M be a Turing machine with input alphabet Σ, and let $L \subseteq \Sigma^*$ be an arbitrary language over Σ.

- M ***recognizes*** or ***accepts*** L if and only if M accepts every string in L but nothing else. A language is ***recognizable*** (or *semi-computable* or *recursively enumerable*) if it is recognized by some Turing machine.
- M ***decides*** L if and only if M accepts every string in L and rejects every string in $\Sigma^* \setminus L$. Equivalently, M decides L if and only if M recognizes L and halts (without crashing) on all inputs. A language is ***decidable*** (or *computable* or *recursive*) if it is decided by some Turing machine.

Trivially, every decidable language is recognizable, but (as we will see later), not every recognizable language is decidable.

6.3 A First Example

Consider the language $L = \{\mathtt{0}^n\mathtt{1}^n\mathtt{0}^n \mid n \geq 0\}$. This language is neither regular nor context-free, but it can be decided by the following six-state Turing machine. The alphabets and states of the machine are defined as follows:

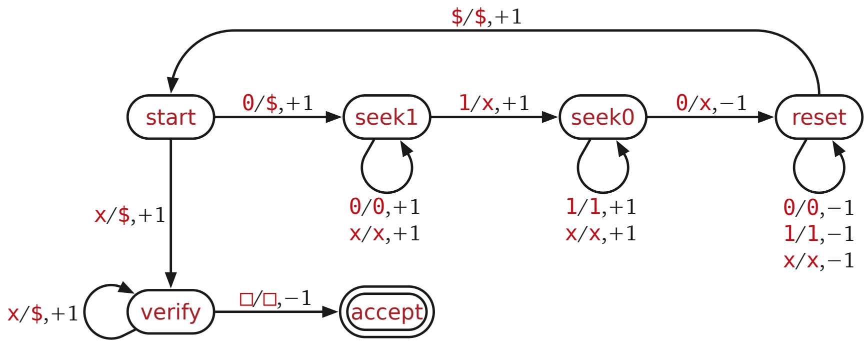
$$\Gamma = \{\mathtt{0}, \mathtt{1}, \$, \mathtt{x}, \square\}$$
$$\Sigma = \{\mathtt{0}, \mathtt{1}\}$$
$$Q = \{\mathsf{start}, \mathsf{seek1}, \mathsf{seek0}, \mathsf{reset}, \mathsf{verify}, \mathsf{accept}, \mathsf{reject}\}$$

The transition function is described in the following table; all unspecified transitions lead to the reject state. We also give a graphical representation of the same machine, which resembles a drawing of a DFA, but with output symbols and actions specified on each edge. For example, we indicate the transition $\delta(p, \mathtt{0}) = (q, \mathtt{1}, +1)$ by writing $\mathtt{0}/\mathtt{1}, +1$ next to the arrow from state p to state q.

$\delta(\ p\ , a) = (\ q\ , b, \Delta)$	explanation
$\delta($start$, \mathtt{0}) = ($seek1$, \$, +1)$	mark first 0 and scan right
$\delta($start$, \mathtt{x}) = ($verify$, \$, +1)$	looks like we're done, but let's make sure
$\delta($seek1$, \mathtt{0}) = ($seek1$, \mathtt{0}, +1)$	scan rightward for 1
$\delta($seek1$, \mathtt{x}) = ($seek1$, \mathtt{x}, +1)$	
$\delta($seek1$, \mathtt{1}) = ($seek0$, \mathtt{x}, +1)$	mark 1 and continue right
$\delta($seek0$, \mathtt{1}) = ($seek0$, \mathtt{1}, +1)$	scan rightward for 0
$\delta($seek0$, \mathtt{x}) = ($seek0$, \mathtt{x}, +1)$	
$\delta($seek0$, \mathtt{0}) = ($reset$, \mathtt{x}, +1)$	mark 0 and scan left
$\delta($reset$, \mathtt{0}) = ($reset$, \mathtt{0}, -1)$	scan leftward for \$
$\delta($reset$, \mathtt{1}) = ($reset$, \mathtt{1}, -1)$	
$\delta($reset$, \mathtt{x}) = ($reset$, \mathtt{x}, -1)$	
$\delta($reset$, \$) = ($start$, \$, +1)$	step right and start over
$\delta($verify$, \mathtt{x}) = ($verify$, \$, +1)$	scan right for any unmarked symbol
$\delta($verify$, \square) = ($accept$, \square, -1)$	success!

The transition function for a Turing machine that decides the language $\{\mathtt{0}^n\mathtt{1}^n\mathtt{0}^n \mid n \geq 0\}$.

Finally, we trace the execution of this machine on two input strings: $\mathtt{001100} \in L$ and $\mathtt{00100} \notin L$. In each configuration, we indicate the position of the head using a small triangle



A graphical representation of the example Turing machine

instead of listing the position explicitly. Notice that we automatically add blanks to the tape string as necessary. Proving that this machine actually decides L—and in particular, that it never crashes or infinite-loops—is a straightforward but tedious exercise in induction.

$$
\begin{aligned}
&(\text{start}, \underset{\blacktriangle}{\texttt{0}}\texttt{01100}) \Rightarrow (\text{seek1}, \texttt{\$}\underset{\blacktriangle}{\texttt{0}}\texttt{1100}) \Rightarrow (\text{seek1}, \texttt{\$0}\underset{\blacktriangle}{\texttt{1}}\texttt{100}) \Rightarrow (\text{seek0}, \texttt{\$0x}\underset{\blacktriangle}{\texttt{1}}\texttt{00}) \Rightarrow (\text{seek0}, \texttt{\$0x1}\underset{\blacktriangle}{\texttt{0}}\texttt{0})\\
&\quad \Rightarrow (\text{reset}, \texttt{\$0x}\underset{\blacktriangle}{\texttt{1}}\texttt{x0}) \Rightarrow (\text{reset}, \texttt{\$0}\underset{\blacktriangle}{\texttt{x}}\texttt{1x0}) \Rightarrow (\text{reset}, \texttt{\$}\underset{\blacktriangle}{\texttt{0}}\texttt{x1x0}) \Rightarrow (\text{reset}, \underset{\blacktriangle}{\texttt{\$}}\texttt{0x1x0})\\
&\quad \Rightarrow (\text{start}, \texttt{\$}\underset{\blacktriangle}{\texttt{0}}\texttt{x1x0})\\
&\quad \Rightarrow (\text{seek1}, \texttt{\$\$}\underset{\blacktriangle}{\texttt{x}}\texttt{1x0}) \Rightarrow (\text{seek1}, \texttt{\$\$x}\underset{\blacktriangle}{\texttt{1}}\texttt{x0}) \Rightarrow (\text{seek0}, \texttt{\$\$xx}\underset{\blacktriangle}{\texttt{x}}\texttt{0}) \Rightarrow (\text{seek0}, \texttt{\$\$xxx}\underset{\blacktriangle}{\texttt{0}})\\
&\quad \Rightarrow (\text{reset}, \texttt{\$\$xx}\underset{\blacktriangle}{\texttt{x}}\texttt{x}) \Rightarrow (\text{reset}, \texttt{\$\$x}\underset{\blacktriangle}{\texttt{x}}\texttt{xx}) \Rightarrow (\text{reset}, \texttt{\$\$}\underset{\blacktriangle}{\texttt{x}}\texttt{xxx}) \Rightarrow (\text{reset}, \texttt{\$}\underset{\blacktriangle}{\texttt{\$}}\texttt{xxxx})\\
&\quad \Rightarrow (\text{verify}, \texttt{\$\$}\underset{\blacktriangle}{\texttt{x}}\texttt{xxx}) \Rightarrow (\text{verify}, \texttt{\$\$\$}\underset{\blacktriangle}{\texttt{x}}\texttt{xx}) \Rightarrow (\text{verify}, \texttt{\$\$\$\$}\underset{\blacktriangle}{\texttt{x}}\texttt{x})\\
&\quad \Rightarrow (\text{verify}, \texttt{\$\$\$\$\$}\underset{\blacktriangle}{\texttt{x}}) \Rightarrow (\text{verify}, \texttt{\$\$\$\$\$\$}\underset{\blacktriangle}{\square}) \Rightarrow (\text{accept}, \texttt{\$\$\$\$\$}\underset{\blacktriangle}{\texttt{\$}}) \Rightarrow \textbf{accept!}
\end{aligned}
$$

The evolution of the example Turing machine on the input string `001100` $\in L$

$$
\begin{aligned}
&(\text{start}, \underset{\blacktriangle}{\texttt{0}}\texttt{0100}) \Rightarrow (\text{seek1}, \texttt{\$}\underset{\blacktriangle}{\texttt{0}}\texttt{100}) \Rightarrow (\text{seek1}, \texttt{\$0}\underset{\blacktriangle}{\texttt{1}}\texttt{00}) \Rightarrow (\text{seek0}, \texttt{\$0x}\underset{\blacktriangle}{\texttt{0}}\texttt{0})\\
&\quad \Rightarrow (\text{reset}, \texttt{\$0}\underset{\blacktriangle}{\texttt{x}}\texttt{x0}) \Rightarrow (\text{reset}, \texttt{\$}\underset{\blacktriangle}{\texttt{0}}\texttt{xx0}) \Rightarrow (\text{reset}, \underset{\blacktriangle}{\texttt{\$}}\texttt{0xx0})\\
&\quad \Rightarrow (\text{start}, \texttt{\$}\underset{\blacktriangle}{\texttt{0}}\texttt{xx0})\\
&\quad \Rightarrow (\text{seek1}, \texttt{\$\$}\underset{\blacktriangle}{\texttt{x}}\texttt{x0}) \Rightarrow (\text{seek1}, \texttt{\$\$x}\underset{\blacktriangle}{\texttt{x}}\texttt{0}) \Rightarrow (\text{seek1}, \texttt{\$\$xx}\underset{\blacktriangle}{\texttt{0}}) \Rightarrow \textbf{reject!}
\end{aligned}
$$

The evolution of the example Turing machine on the input string `00100` $\notin L$

6.4 Variations

There are actually several formal models that all fall under the name "Turing machine", each with small variations on the definition we've given. Although we do need to be explicit about *which* variant we want to use for any particular problem, the differences between the variants are relatively unimportant. For any machine defined in one model, there is an equivalent machine in each of the other models; in particular, all of these variants recognize the same languages and decide the same languages. For example:

- **Halting conditions.** Some models allow multiple accept and reject states, which (depending on the precise model) trigger acceptance or rejection either when the machine enters the state, or when the machine has no valid transitions out of such a state. Others include only explicit accept states, and either equate crashing with rejection or do not define a rejection mechanism at all. Still other models include halting as one of the possible *actions* of the machine, in addition to moving left or moving right; in these models, the machine accepts/rejects its input if and only if it halts in an accepting/non-accepting state.

- **Actions.** Some Turing machine models allow transitions that do not move the head, or that move the head by more than one cell in a single step. Others insist that a single step of the machine *either* writes a new symbol onto the tape *or* moves the head one step. Finally, as mentioned above, some models include halting as one of the available actions.

- **Transition function.** Some models of Turing machines, including Turing's original definition, allow the transition function to be undefined on some state-symbol pairs. In this formulation, the transition function is given by a set $\delta \subset Q \times \Gamma \times Q \times \Gamma \times \{+1, -1\}$, such that for each state q and symbol a, there is at most one transition $(q, a, \cdot, \cdot, \cdot) \in \delta$. If the machine enters a configuration from which there is no transition, it halts and (depending on the precise model) either crashes or rejects. Others define the transition function as $\delta \colon Q \times \Gamma \to Q \times (\Gamma \cup \{-1, +1\})$, allowing the machine to *either* write a symbol to the tape *or* move the head in each step.

- **Beginning of the tape.** Some models forbid the head to move past the beginning of the tape, either by starting the tape with a special symbol that cannot be overwritten and that forces a rightward transition, or by declaring that a leftward transition at position 0 leaves the head in position 0, or even by pure fiat—declaring any machine that performs a leftward move at position 0 to be invalid.

To prove that any two of these variant "species" of Turing machine are equivalent, we must show how to transform a machine of one species into a machine of the other species that accepts and rejects the same strings. For example, let $M = (\Gamma, \square, \Sigma, Q, s, \mathsf{accept}, \mathsf{reject}, \delta)$ be a Turing machine with explicit accept and reject states. We can define an equivalent Turing machine M' that halts only when it moves left from position 0, and accepts only by halting while in an accepting state, as follows. We define the set of accepting states for M' as $A = \{\mathsf{accept}\}$ and define a new transition function

$$\delta'(q, a) := \begin{cases} (\mathsf{accept}, a, -1) & \text{if } q = \mathsf{accept} \\ (\mathsf{reject}, a, -1) & \text{if } q = \mathsf{reject} \\ \delta(q, a) & \text{otherwise} \end{cases}$$

Similarly, suppose someone gives us a Turing machine $M = (\Gamma, \square, \Sigma, Q, s, \mathsf{accept}, \mathsf{reject}, \delta)$ whose transition function $\delta \colon Q \times \Gamma \to Q \times \Gamma \times \{-1, 0, +1\}$ allows the machine to transition without moving its head. We can construct an equivalent Turing machine $M' = (\Gamma, \square, \Sigma, Q', s, \mathsf{accept}, \mathsf{reject}, \delta')$ that moves its head at every transition by defining $Q' := Q \times \{0, 1\}$ and

$$\begin{aligned} \delta'((p, 0), a) &:= \begin{cases} ((q, 1), b, +1) & \text{if } \delta(p, a) = (q, b, 0), \\ ((q, 0), b, \Delta) & \text{if } \delta(p, a) = (q, b, \Delta) \text{ and } \Delta \neq 0, \end{cases} \\ \delta'((p, 1), a) &:= ((p, 0), a, -1). \end{aligned}$$

6.5 Computing Functions

Turing machines can also be used to compute functions from strings to strings, instead of just accepting or rejecting strings. Since we don't care about acceptance or rejection, we replace the explicit accept and reject states with a single halt state, and we define the ***output*** of the Turing machine to be the contents of the tape when the machine halts, after removing the infinite sequence of trailing blanks. More formally, for any Turing machine M, any string $w \in \Sigma^*$, and any string $x \in \Gamma^*$ that does not end with a blank, we write $\boldsymbol{M(w) = x}$ if and only if $(w, s, 0) \Rightarrow_M^* (x, \mathsf{halt}, i)$ for some integer i. If M does not halt on input w, then we write $\boldsymbol{M(w)} \nearrow$, which can be read either "M diverges on w" or "$M(w)$ is undefined." We say that M ***computes*** the function $f : \Sigma^* \to \Sigma^*$ if and only if $M(w) = f(w)$ for every string w.

6.5.1 Shifting

One basic operation that is used in many Turing machine constructions is ***shifting*** the input string a constant number of steps to the right or to the left. For example, given any input string $w \in \{\mathtt{0}, \mathtt{1}\}^*$, we can compute the string $\mathtt{0}w$ using a Turing machine with tape alphabet $\Gamma = \{\mathtt{0}, \mathtt{1}, \square\}$, state set $Q = \{0, 1, \mathsf{halt}\}$, start state 0, and the following transition function:

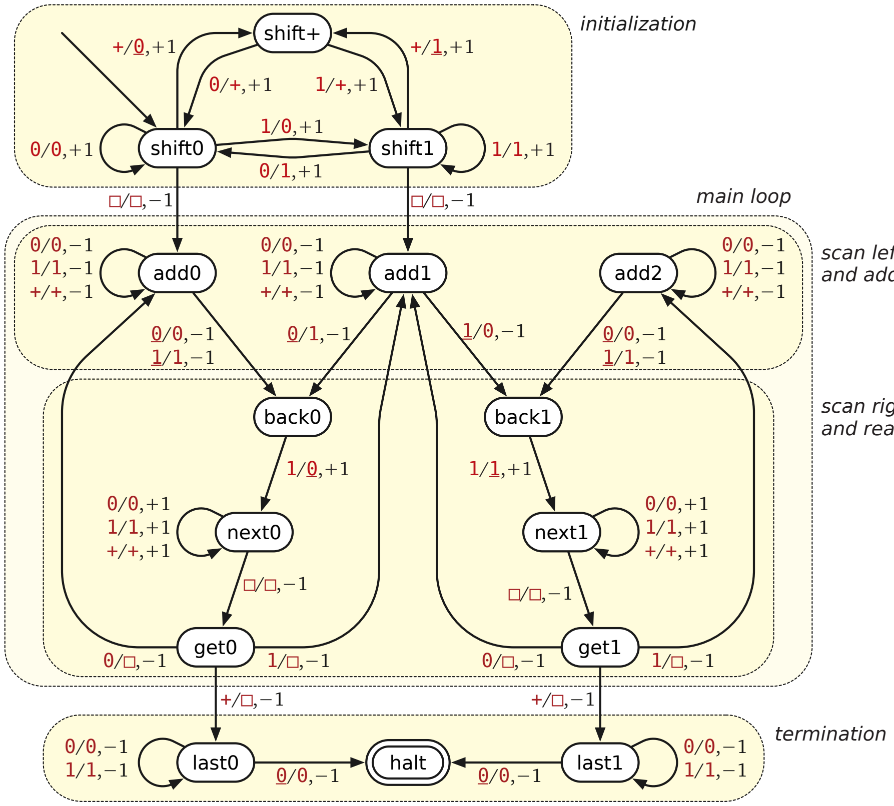
$$
\begin{array}{l}
\delta(p,\ a) = (\ q\ ,\ b,\ \Delta\ ) \\
\hline
\delta(0,\ \mathtt{0}) = (\ 0\ ,\ \mathtt{0},\ +1) \\
\delta(0,\ \mathtt{1}) = (\ 1\ ,\ \mathtt{0},\ +1) \\
\delta(0,\ \square) = (\mathsf{halt},\ \mathtt{0},\ +1) \\
\hdashline
\delta(1,\ \mathtt{0}) = (\ 0\ ,\ \mathtt{1},\ +1) \\
\delta(1,\ \mathtt{1}) = (\ 1\ ,\ \mathtt{1},\ +1) \\
\delta(1,\ \square) = (\mathsf{halt},\ \mathtt{1},\ +1)
\end{array}
$$

By increasing the number of states, we can build a Turing machine that shifts the input string any fixed number of steps in either direction. For example, a machine that shifts its input to the left by five steps might read the string from right to left, storing the five most recently read symbols in its internal state. A typical transition for such a machine would be $\delta(12345, 0) = (01234, \mathtt{5}, -1)$.

6.5.2 Binary Addition

With a more complex Turing machine, we can implement binary addition. The input is a string of the form $w\mathtt{+}x$, where $w, x \in \{\mathtt{0}, \mathtt{1}\}^n$, representing two numbers in binary; the output is the binary representation of $w + x$. To simplify our presentation, we assume that $|w| = |x| > 0$; however, this restrictions can be removed with the addition of a few more states. The following figure shows the entire Turing machine at a glance. The machine uses the tape alphabet $\Gamma = \{\square, \mathtt{0}, \mathtt{1}, \mathtt{+}, \underline{\mathtt{0}}, \underline{\mathtt{1}}\}$; the start state is shift0. All missing transitions go to a fail state, indicating that the input was badly formed.

Execution of this Turing machine proceeds in several phases, each with its own subset of states, as indicated in the figure. The initialization phase scans the entire input, shifting it to the right to make room for the output string, marking the rightmost bit of w, and reading and erasing the last bit of x.



A Turing machine that adds two binary numbers of the same length.

$\delta($ p , $a)$	=	$($ q , b, $\Delta)$
δ(shift0, 0)	=	(shift0, 0, +1)
δ(shift0, 1)	=	(shift1, 0, +1)
δ(shift0, +)	=	(shift+, $\underline{0}$, +1)
δ(shift0, □)	=	(add0, □, −1)
δ(shift1, 0)	=	(shift0, 1, +1)
δ(shift1, 1)	=	(shift1, 1, +1)
δ(shift1, +)	=	(shift+, $\underline{1}$, +1)
δ(shift1, □)	=	(add1, □, −1)
δ(shift+, 0)	=	(shift0, +, +1)
δ(shift+, 1)	=	(shift1, +, +1)

The first part of the main loop scans left to the marked bit of w, adds the bit of x that was just erased plus the carry bit from the previous iteration, and records the carry bit for the next iteration in the machines internal state.

$\delta($ p , $a)$	=	$($ q , b, $\Delta)$
δ(add0, 0)	=	(add0, 0, −1)
δ(add0, 1)	=	(add0, 0, −1)
δ(add0, +)	=	(add0, 0, −1)
δ(add0, $\underline{0}$)	=	(back0, 0, −1)
δ(add0, $\underline{1}$)	=	(back0, 1, −1)

$\delta($ p , $a)$	=	$($ q , b, $\Delta)$
δ(add1, 0)	=	(add1, 0, −1)
δ(add1, 1)	=	(add1, 0, −1)
δ(add1, +)	=	(add1, 0, −1)
δ(add1, $\underline{0}$)	=	(back0, 1, −1)
δ(add1, $\underline{1}$)	=	(back1, 0, −1)

$\delta($ p , $a)$	=	$($ q , b, $\Delta)$
δ(add2, 0)	=	(add2, 0, −1)
δ(add2, 1)	=	(add2, 0, −1)
δ(add2, +)	=	(add2, 0, −1)
δ(add2, $\underline{0}$)	=	(back1, 0, −1)
δ(add2, $\underline{1}$)	=	(back1, 1, −1)

The second part of the main loop marks the previous bit of w, scans right to the end of x, and then reads and erases the last bit of x, all while maintaining the carry bit.

$\delta(\ p\ ,\ a)$	=	$(\ q\ ,\ b,\ \Delta)$
δ(back0, 0)	=	(next0, $\underline{0}$, +1)
δ(back0, 1)	=	(next0, $\underline{1}$, +1)
δ(next0, 0)	=	(next0, 0, +1)
δ(next0, 1)	=	(next0, 0, +1)
δ(next0, +)	=	(next0, 0, +1)
δ(next0, □)	=	(get0, □, −1)
δ(get0, 0)	=	(add0, □, −1)
δ(get0, 1)	=	(add1, □, −1)
δ(get0, +)	=	(last0, □, −1)

$\delta(\ p\ ,\ a)$	=	$(\ q\ ,\ b,\ \Delta)$
δ(back1, 0)	=	(next1, $\underline{0}$, +1)
δ(back1, 1)	=	(next1, $\underline{1}$, +1)
δ(next1, 0)	=	(next1, 0, +1)
δ(next1, 1)	=	(next1, 0, +1)
δ(next1, +)	=	(next1, 0, +1)
δ(next1, □)	=	(get1, □, −1)
δ(get1, 0)	=	(add1, □, −1)
δ(get1, 1)	=	(add2, □, −1)
δ(get1, +)	=	(last1, □, −1)

Finally, after erasing the + in the last iteration of the main loop, the termination phase adds the last carry bit to the leftmost output bit and halts.

$\delta(\ p\ ,\ a)$	=	$(\ q\ ,\ b,\ \Delta)$
δ(last0, 0)	=	(last0, 0, −1)
δ(last0, 1)	=	(last0, 0, −1)
δ(last0, $\underline{0}$)	=	(halt, 0,)
δ(last1, 0)	=	(last1, 0, −1)
δ(last1, 1)	=	(last1, 0, −1)
δ(last1, $\underline{0}$)	=	(halt, 1,)

6.6 Variations on Tracks, Heads, and Tapes

Multiple Tracks

It is sometimes convenient to endow the Turing machine tape with multiple ***tracks***, each with its own tape alphabet, and allow the machine to read from and write to the same position on all tracks simultaneously. For example, to define a Turing machine with three tracks, we need three tape alphabets Γ_1, Γ_2, and Γ_3, each with its own blank symbol, where (say) Γ_1 contains the input alphabet Σ as a subset; we also need a transition function of the form

$$\delta : Q \times \Gamma_1 \times \Gamma_2 \times \Gamma_3 \to Q \times \Gamma_1 \times \Gamma_2 \times \Gamma_3 \times \{-1, +1\}$$

Describing a configuration of this machine requires a quintuple (q, x_1, x_2, x_3, i), indicating that each track i contains the string x_i followed by an infinite sequence of blanks. The initial configuration is $(\text{start}, w, \varepsilon, \varepsilon, 0)$, with the input string written on the first track, and the other two tracks completely blank.

But any such machine is equivalent (if not *identical*) to a single-track Turing machine with the (still finite!) tape alphabet $\Gamma := \Gamma_1 \times \Gamma_2 \times \Gamma_3$. Instead of thinking of the tape as three infinite sequences of symbols, we think of it as a single infinite sequence of "records", each containing three symbols. Moreover, there's nothing special about the number 3 in this construction; a Turing machine with *any* constant number of tracks is equivalent to a single-track machine.

Doubly-Infinite Tape

It is also sometimes convenient to allow the tape to be infinite in both directions, for example, to avoid boundary conditions. There are several ways to simulate a doubly-infinite tape on a machine with only a semi-infinite tape. Perhaps the simplest method is to use a semi-infinite tape with two tracks, one containing the cells with positive index and the other containing the cells with negative index in reverse order, with a special marker symbol at position zero to indicate the transition.

0	+1	+2	+3	+4	···
►	−1	−2	−3	−4	···

Another method is to shuffle the positive-index and negative-index cells onto a single track, and add additional states to allow the Turing machine to move two steps in a single transition. Again, we need a special symbol at the left end of the tape to indicate the transition:

►	0	−1	+1	−2	+2	−3	+3	···

A third method maintains two sentinel symbols ► and ◄ that surround all other non-blank symbols on the tape. Whenever the machine reads the right sentinel ◄, we write a blank, move right, write ◄, move left, and then proceed as if we had just read a blank. On the other hand, when the machine reads the left sentinel ►, we shift the entire contents of the tape (up to and including the right sentinel) one step to the right, then move back to the left sentinel, move right, write a blank, and finally proceed as if we had just read a blank. Since the Turing machine does not actually have access to the position of the head *as an integer*, shifting the head and the tape contents one step right has no effect on its future evolution.

►	−3	−2	−1	0	+1	+2	+3	+4	+5	◄	···

Using either of the first two methods, we can simulate t steps of an arbitrary Turing machine with a doubly-infinite tape using only $O(t)$ steps on a standard Turing machine. The third method, unfortunately, requires $\Theta(t^2)$ steps in the worst case.

Insertion and Deletion

We can also allow Turing machines to insert and delete cells on the tape, in addition to simply overwriting existing symbols. We've already seen how to insert a new cell: Leave a special mark on the tape (perhaps in a second track), shift everything to the right of this mark one cell to the right, scan left to the mark, erase the mark, and finally write the correct character into the new cell. Deletion is similar: Mark the cell to be deleted, shift everything to the right of the mark one step to the left, scan left to the mark, and erase the mark. We may also need to maintain a mark in some cell to the right every non-blank symbol, indicating that all cells further to the right are blank, so that we know when to stop shifting left or right.

Multiple Heads

Another convenient extension is to allow machines simultaneous access to more than one position on the tape. For example, to define a Turing machine with *three* heads, we need a transition function of the form

$$\delta : Q \times \Gamma^3 \to Q \times \Gamma^3 \times \{-1, +1\}^3.$$

Describing a configuration of such a machine requires a quintuple (q, x, i, j, k), indicating that the machine is in state q, the tape contains string x, and the three heads are at positions i, j, k. The transition function tells us, given q and the three symbols $x[i], x[j], x[k]$, which three symbols to write on the tape and which direction to move each of the heads.

We can simulate this behavior with a single head by adding additional tracks to the tape that record the positions of each head. To simulate a machine M with three heads, we use a tape with four tracks: track 0 is the actual work tape; each of the remaining tracks has a single non-blank symbol recording the position of one of the heads. We also insert a special marker symbols at the left end of the tape.

▶	M	Y	W	O	R	K	T	A	P	E	⋯
▶									▲		⋯
▶		▲									⋯
▶						▲					⋯

We can simulate any single transition of M, starting with our single head at the left end of the tape, as follows. Throughout the simulation, we maintain the internal state of M as one of the components of our current state. First, for each i, we read the symbol under the ith head of M as follows:

> Scan to the right to find the mark on track i, read the corresponding symbol from track 0 into our internal state, and then return to the left end of the tape.

At this point, our internal state records M's current internal state and the three symbols under M's heads. After one more transition (using M's transition function), our internal state records M's *next* state, the symbol to be written by each head, and the direction to move each head. Then, for each i, we write with and move the ith head of M as follows:

> Scan to the right to find the mark on track i, write the correct symbol onto on track 0, move the mark on track i one step left or right, and then return to the left end of the tape.

Again, there is nothing special about the number 3 here; we can simulate machines with *any* fixed number of heads.

Careful analysis of this technique implies that for any integer k, we can simulate t steps of an arbitrary Turing machine with k independent heads in $\Theta(t^2)$ time on a standard Turing machine with only one head. Unfortunately, this quadratic blowup is unavoidable. It is relatively easy to recognize the language of *marked palindromes* $\{w \bullet w^R \mid w \in \{0,1\}^*\}$ in $O(n)$ time using a Turing machine with two heads, but recognizing this language provably requires $\Omega(n^2)$ time on a standard machine with only one head. On the other hand, with much more sophisticated techniques, it is possible to simulate t steps of a Turing machine with k head, for any fixed integer k, using only $O(t \log t)$ steps on a Turing machine with just *two* heads.

Multiple Tapes

We can also allow machines with multiple independent tapes, each with its own head. To simulate such a machine with a single tape, we simply maintain each tape as an independent track with its own head. Equivalently, we can simulate a machine with k tapes using a single tape with $2k$ tracks, half storing the contents of the k tapes and half storing the positions of the k heads.

▶	T	A	P	E	#	O	N	E			⋯
▶									▲		⋯
▶	T	A	P	E	#	T	W	O			⋯
▶			▲								⋯
▶	T	A	P	E	#	T	H	R	E	E	⋯
▶						▲					⋯

Just as for multiple tracks, for any constant k, we can simulate t steps of an arbitrary Turing machine with k independent tapes in $\Theta(t^2)$ steps on a standard Turing machine with one tape, and this quadratic blowup is unavoidable. Moreover, it is possible to simulate t steps on a k-tape Turing machine using only $O(t \log t)$ steps on a *two*-tape Turing machine using more sophisticated techniques. (This faster simulation is easier to obtain for multiple independent tapes than for multiple heads on the same tape.)

By combining these tricks, we can simulate a Turing machine with any fixed number of tapes, each of which may be infinite in one or both directions, each with any fixed number of heads and any fixed number of tracks, with at most a quadratic blowup in the running time.

6.7 Simulating a Real Computer

6.7.1 Subroutines and Recursion

★★★

Use a second tape/track as a "call stack". Add save and restore actions. In the simplest formulation, subroutines do not have local memory. To call a subroutine, save the current state onto the call stack and jump to the first state of the subroutine. To return, restore (and remove) the return state from the call stack. We can simulate t steps of any recursive Turing machine with $O(t)$ steps on a multitape standard Turing machine, or in $O(t^2)$ steps on a standard Turing machine.

More complex versions of this simulation can adapt to

6.7.2 Random-Access Memory

★★★

Keep [address•data] pairs on a separate "memory" tape. Write address to an "address" tape; read data from or write data to a "data" tape. Add new or changed [address•data] pairs at the end of the memory tape. (Semantics of reading from an address that has never been written to?)

Suppose all memory accesses require at most ℓ address and data bits. Then we can simulate the kth memory access in $O(k\ell)$ steps on a multitape Turing machine or in $O(k^2\ell^2)$ steps on a single-tape machine. Thus, simulating t memory accesses in a random-access machine with ℓ-bit words requires $O(t^2\ell)$ time on a multitape Turing machine, or $O(t^3\ell^2)$ time on a single-tape machine.

6.8 Universal Turing Machines

With all these tools in hand, we can now describe the pinnacle of Turing machine constructions: the ***universal*** Turing machine. For modern computer scientists, it's useful to think of a universal Turing machine as a "Turing machine *interpreter* written in Turing machine". Just as the input to a Python interpreter is a string of Python source code, the input to our universal Turing machine U is a string $\langle M, w\rangle$ that encodes both an arbitrary Turing machine M and a string w in the input alphabet of M. Given these encodings, U simulates the execution of M on input w; in particular,

- U accepts $\langle M, w\rangle$ if and only if M accepts w.
- U rejects $\langle M, w\rangle$ if and only if M rejects w.

In the next few pages, I will sketch a universal Turing machine U that uses the input alphabet {0, 1, [,], •, |} and a somewhat larger tape alphabet (via marks on additional tracks). However, I do *not* require that the Turing machines that U simulates have similarly small alphabets, so we first need a method to encode *arbitrary* input and tape alphabets.

Encodings

Let $M = (\Gamma, \square, \Sigma, Q, \mathit{start}, \mathit{accept}, \mathit{reject}, \delta)$ be an arbitrary Turing machine, with a single half-infinite tape and a single read-write head. (I will consistently indicate the states and tape symbols of M in *slanted green* to distinguish them from the upright red states and tape symbols of U.)

We encode each symbol $a \in \Gamma$ as a unique string $|a|$ of $\lceil \lg(|\Gamma|) \rceil$ bits. Thus, if $\Gamma = \{\mathit{0}, \mathit{1}, \mathit{\$}, \mathit{x}, \square\}$, we might use the following encoding:

$$\langle \mathit{0} \rangle = \mathtt{001}, \qquad \langle \mathit{1} \rangle = \mathtt{010}, \qquad \langle \mathit{\$} \rangle = \mathtt{011}, \qquad \langle \mathit{x} \rangle = \mathtt{100}, \qquad \langle \square \rangle = \mathtt{000}.$$

The input string w is encoded by its sequence of symbol encodings, with separators • between every pair of symbols and with brackets [and] around the whole string. For example, with this encoding, the input string *001100* would be encoded on the input tape as

$$\langle \mathit{001100} \rangle = \texttt{[001}\bullet\texttt{001}\bullet\texttt{010}\bullet\texttt{010}\bullet\texttt{001}\bullet\texttt{001]}$$

Similarly, we encode each state $q \in Q$ as a distinct string $\langle q \rangle$ of $\lceil \lg |Q| \rceil$ bits. Without loss of generality, we encode the start state with all 1s and the reject state with all 0s. For example, if $Q = \{\mathit{start}, \mathit{seek1}, \mathit{seek0}, \mathit{reset}, \mathit{verify}, \mathit{accept}, \mathit{reject}\}$, we might use the following encoding:

$$\langle \mathit{start} \rangle = \mathtt{111} \qquad \langle \mathit{seek1} \rangle = \mathtt{010} \qquad \langle \mathit{seek0} \rangle = \mathtt{011} \qquad \langle \mathit{reset} \rangle = \mathtt{100}$$
$$\langle \mathit{verify} \rangle = \mathtt{101} \qquad \langle \mathit{accept} \rangle = \mathtt{110} \qquad \langle \mathit{reject} \rangle = \mathtt{000}$$

We encode the machine M itself as the string $\langle M \rangle = \texttt{[}\langle \mathit{reject} \rangle \bullet \langle \square \rangle\texttt{]}\langle \delta \rangle$, where $\langle \delta \rangle$ is the concatenation of substrings $\texttt{[}\langle p \rangle \bullet \langle a \rangle \texttt{|} \langle q \rangle \bullet \langle b \rangle \bullet \langle \Delta \rangle\texttt{]}$ encoding each transition $\delta(p, a) = (q, b, \Delta)$ such that $q \neq \mathrm{reject}$. We encode the actions $\Delta = \pm 1$ by defining $\langle -1 \rangle := \mathtt{0}$ and $\langle +1 \rangle := \mathtt{1}$. Conveniently, every transition string has exactly the same length. For example, with the symbol and state encodings described above, the transition $\delta(\mathit{reset}, \mathit{\$}) = (\mathit{start}, \mathit{\$}, +1)$ would be encoded as

$$\texttt{[100}\bullet\texttt{011|001}\bullet\texttt{011}\bullet\texttt{1]}.$$

Our first example Turing machine for recognizing $\{\mathit{0}^n \mathit{1}^n \mathit{0}^n \mid n \geq 0\}$ would be represented by the following string (here broken into multiple lines for readability):

```
[000•000][[001•001|010•011•1][001•100|101•011•1]
         [010•001|010•001•1][010•100|010•100•1]
         [010•010|011•100•1][011•010|011•010•1]
         [011•100|011•100•1][011•001|100•100•1]
         [100•001|100•001•0][100•010|100•010•0]
         [100•100|100•100•0][100•011|001•011•1]
         [101•100|101•011•1][101•000|110•000•0]]
```

Finally, we encode any *configuration* of M on U's work tape by alternating between encodings of states and encodings of tape symbols. Thus, each tape cell is represented by the string $\texttt{[}\langle q \rangle \bullet \langle a \rangle\texttt{]}$ indicating that (1) the cell contains symbol a; (2) if $q \neq \mathit{reject}$, then M's head is located at this cell, and M is in state q; and (3) if $q = \mathit{reject}$, then M's head is located somewhere else. Conveniently, each cell encoding uses exactly the same number of bits. We also surround the entire tape encoding with brackets [and].

For example, with the encodings described above, the initial configuration $(\mathit{start}, \underset{\blacktriangle}{\mathit{0}}\mathit{01100}, 0)$ for our first example Turing machine would be encoded on U's tape as follows.

$$\langle \mathit{start}, \underset{\blacktriangle}{\mathit{0}}\mathit{01100}, 0 \rangle = \texttt{[}\underbrace{\texttt{[111}\bullet\texttt{001]}}_{\textbf{\textit{start}}\ \mathit{0}}\underbrace{\texttt{[000}\bullet\texttt{001]}}_{\mathit{reject}\ \mathit{0}}\underbrace{\texttt{[000}\bullet\texttt{010]}}_{\mathit{reject}\ \mathit{1}}\underbrace{\texttt{[000}\bullet\texttt{010]}}_{\mathit{reject}\ \mathit{1}}\underbrace{\texttt{[000}\bullet\texttt{001]}}_{\mathit{reject}\ \mathit{0}}\underbrace{\texttt{[000}\bullet\texttt{001]}}_{\mathit{reject}\ \mathit{0}}\texttt{]}$$

Similarly, the intermediate configuration (reset, $\mathtt{\$0x}\underset{\blacktriangle}{\mathtt{1}}\mathtt{x0}$, 3) would be encoded as follows:

$$\langle \mathit{reset}, \mathtt{\$\$x}\underset{\blacktriangle}{\mathtt{1}}\mathtt{x0}, 3\rangle = \mathtt{[}\underbrace{\mathtt{[000{\bullet}011]}}_{\textit{reject \$}}\underbrace{\mathtt{[000{\bullet}011]}}_{\textit{reject 0}}\underbrace{\mathtt{[000{\bullet}100]}}_{\textit{reject x}}\underbrace{\mathtt{[010{\bullet}010]}}_{\textbf{\textit{reset}}\textit{ 1}}\underbrace{\mathtt{[000{\bullet}100]}}_{\textit{reject x}}\underbrace{\mathtt{[000{\bullet}001]}}_{\textit{reject 0}}\mathtt{]}$$

Input and Execution

Without loss of generality, we assume that the input to our universal Turing machine U is given on a separate read-only ***input tape***, as the encoding of an arbitrary Turing machine M followed by an encoding of its input string x. Notice the substrings `[[` and `]]` each appear only only once on the input tape, immediately before and after the encoded transition table, respectively. U also has a read-write ***work tape***, which is initially blank.

We start by initializing the work tape with the encoding $\langle \mathit{start}, x, 0\rangle$ of the initial configuration of M with input x. First, we write `[[`$\langle \mathit{start}\rangle$`•`. Then we copy the encoded input string $\langle x\rangle$ onto the work tape, but we change the punctuation as follows:

- Instead of copying the left bracket `[`, write `[[`$\langle \mathit{start}\rangle$`•`.
- Instead of copying each separator `•`, write `][`$\langle \mathit{reject}\rangle$`•`
- Instead of copying the right bracket `]`, write two right brackets `]]`.

The state encodings $\langle \mathit{start}\rangle$ and $\langle \mathit{reject}\rangle$ can be copied directly from the beginning of $\langle M\rangle$ (replacing `0`s for `1`s for $\langle \mathit{start}\rangle$). Finally, we move the head back to the start of U's tape.

At the start of each step of the simulation, U's head is located at the start of the work tape. We scan through the work tape to the unique encoded cell `[`$\langle p\rangle$`•`$\langle a\rangle$`]` such that $p \neq \mathit{reject}$. Then we scan through the encoded transition function $\langle \delta\rangle$ to find the unique encoded tuple `[`$\langle p\rangle$`•`$\langle a\rangle$`|`$\langle q\rangle$`•`$\langle b\rangle$`•`$\langle \Delta\rangle$`]` whose left half matches our the encoded tape cell. If there is no such tuple, then U immediately halts and rejects. Otherwise, we copy the right half $\langle q\rangle$`•`$\langle b\rangle$ of the tuple to the work tape. Now if $q = \mathit{accept}$, then U immediately halts and accepts. (We don't bother to encode *reject* transformations, so we know that $q \neq \mathit{reject}$.) Otherwise, we transfer the state encoding to either the next or previous encoded cell, as indicated by M's transition function, and then continue with the next step of the simulation.

During the final state-copying phase, we ever read two right brackets `]]`, indicating that we have reached the right end of the tape encoding, we replace the second right bracket with `[`$\langle \mathit{reject}\rangle$`•`$\langle \square\rangle$`]]` (mostly copied from the beginning of the machine encoding $\langle M\rangle$) and then scan back to the left bracket we just wrote. This trick allows our universal machine to *pretend* that its tape contains an infinite sequence of *encoded* blanks `[`$\langle \mathit{reject}\rangle$`•`$\langle \square\rangle$`]` instead of *actual* blanks $\square$.

Example

As an illustrative example, suppose U is simulating our first example Turing machine M on the input string `001100`. The execution of M on input w eventually reaches the configuration ($\mathit{seek1}$, $\mathtt{\$\$x}\underset{\blacktriangle}{\mathtt{1}}\mathtt{x0}$, 3). At the start of the corresponding step in U's simulation, U is in the following configuration:

$$\underset{\blacktriangle}{\mathtt{[}}\mathtt{[000{\bullet}011][000{\bullet}011][000{\bullet}100][010{\bullet}010][000{\bullet}100][000{\bullet}001]]}$$

First U scans for the first encoded tape cell whose state is not *reject*. That is, U repeatedly compares the first half of each encoded state cell on the work tape with the prefix `[`$\langle \mathit{reject}\rangle$`•` of the machine encoding $\langle M\rangle$ on the input tape. U finds a match in the fourth encoded cell.

$$\mathtt{[[000{\bullet}011][000{\bullet}011][000{\bullet}100][010}\underset{\blacktriangle}{\bullet}\mathtt{010][000{\bullet}100][000{\bullet}001]]}$$

Next, U scans the machine encoding $\langle M\rangle$ for the substring `[010•010` matching the current encoded cell. U eventually finds a match in the left size of the the encoded transition `[010•010|011•100•1]`. U copies the state-symbol pair `011•100` from the right half of this encoded transition into the current encoded cell. (The underline indicates which symbols are changed.)

`[[000•011][000•011][000•100][011•100][000•100][000•001]]`

The encoded transition instructs U to move the current state encoding one cell to the right. (The underline indicates which symbols are changed.)

`[[000•011][000•011][000•100][000•100][011•100][000•001]]`

Finally, U scans left until it reads two left brackets `[[`; this returns the head to the left end of the work tape to start the next step in the simulation. U's tape now holds the encoding of M's configuration (*seek0*, \$\$xx*x*0, 4), as required.

`[[000•011][000•011][000•100][000•100][011•100][000•001]]`

Exercises

1. Describe Turing machines that decide each of the following languages:

   (a) Palindromes over the alphabet $\{0,1\}$

   (b) $\{ww \mid w \in \{0,1\}^*\}$

   (c) $\left\{0^a 1^b 0^{ab} \mid a, b \in \mathbb{N}\right\}$

2. Let $\langle n\rangle_2$ denote the binary representation of the non-negative integer n. For example, $\langle 17\rangle_2 = 10001$ and $\langle 42\rangle_2 = 101010$. Describe Turing machines that compute the following functions from $\{0,1\}^*$ to $\{0,1\}^*$:

   (a) $w \mapsto www$

   (b) $1^n 0 1^m \mapsto 1^{mn}$

   (c) $1^n \mapsto 1^{2^n}$

   (d) $1^n \mapsto \langle n\rangle_2$

   (e) $0^*\langle n\rangle_2 \mapsto 1^n$

   (f) $\langle n\rangle_2 \mapsto \langle n^2\rangle_2$

3. Describe Turing machines that write each of the following infinite streams of bits onto their tape. Specifically, for each integer n, there must be a finite time after which the first n symbols on the tape always match the first n symbols in the target stream.

   (a) An infinite stream of 1s

   (b) 0101101110111101111101111110..., where the nth block of 1s has length n.

   (c) The stream of bits whose nth bit is 1 if and only if n is prime.

(d) The ***Thue-Morse sequence*** $T_0 \bullet T_1 \bullet T_2 \bullet T_3 \cdots$, where

$$T_n := \begin{cases} \mathtt{0} & \text{if } n = 0 \\ \mathtt{1} & \text{if } n = 1 \\ T_{n-1} \bullet \overline{T_{n-1}} & \text{otherwise} \end{cases}$$

where $\overline{w}$ indicates the binary string obtained from w by flipping every bit. Equivalently, the nth bit of the Thue Morse sequence if `0` if the binary representation of n has an even number of `1`s and `1` otherwise.

`0 1 10 1001 10010110 1001011001101001 10010110011010010110100110010110...`

(e) The ***Fibonacci*** sequence $F_0 \bullet F_1 \bullet F_2 \bullet F_3 \cdots$, where

$$F_n := \begin{cases} \mathtt{0} & \text{if } n = 0 \\ \mathtt{1} & \text{if } n = 1 \\ F_{n-2} \bullet F_{n-1} & \text{otherwise} \end{cases}$$

`0 1 01 101 01101 10101101 0110110101101 101011010110110101101...`

4. A ***two-stack machine*** is a Turing machine with two tapes with the following restricted behavior. At all times, on each tape, every cell to the right of the head is blank, and every cell at or to the left of the head is non-blank. Thus, a head can only move right by writing a non-blank symbol into a blank cell; symmetrically, a head can only move left by erasing the rightmost non-blank cell. Thus, each tape behaves like a stack. To avoid underflow, there is a special symbol at the start of each tape that cannot be overwritten. Initially, one tape contains the input string, with the head at its *last* symbol, and the other tape is empty (except for the start-of-tape symbol).

   Prove formally that any standard Turing machine can be simulated by a two-stack machine. That is, given any standard Turing machine M, describe a two-stack machine M' that accepts and rejects exactly the same input strings as M.

★★★

Counter machines. Configuration consists of k rational numbers and an internal state (from some finite set Q). Transition function $\delta\colon Q \times \{=0, >0\}^k \to Q \times \{-1, 0, +1\}^k$ takes internal state and signs of counters as input, and produces new internal state and changes to counters as output.

- Prove that any Turing machine can be simulated by a three-counter machine. One counter holds the binary representation of the tape after the head; another counter holds the reversed binary representation of the tape before the head. Implement transitions via halving, doubling, and parity, using the third counter for scratch work.
- Prove that two counters can simulate three. Store $2^a 3^b 5^c$ in one counter, use the other for scratch work.
- Prove that a three-counter machine can compute any computable function: Given input $(n, 0, 0)$, we can compute $(f(n), 0, 0)$ for *any* computable function f. First transform $(n, 0, 0)$ to $(2^n, 0, 0)$ using all three counters; then run two- (or three-)counter TM simulation to obtain $(2^{f(n)}, 0, 0)$; and finally transform $(2^{f(n)}, 0, 0)$ to $(f(n), 0, 0)$ using all three counters.
- **HARD:** Prove that a two-counter machine cannot transform $(n, 0)$ to $(2^n, 0)$. [Barzdin' 1963, Yao 1971, Schröpel 1972, Ibarra+Trân 1993]

★★★

FRACTRAN [Conway 1987]: One-counter machine whose "program" is a sequence of rational numbers. The counter is initially 1. At each iteration, multiply the counter by the first rational number that yields an integer; if there is no such number, halt.

- Prove that for any computable function $f : \mathbb{N} \to \mathbb{N}$, there is a FRACTRAN program that transforms 2^{n+1} into $3^{f(n)+1}$, for all natural numbers n.
- Prove that every FRACTRAN program, given the integer 1 as input, either outputs 1 or loops forever. It follows that there is no FRACTRAN program for the increment function $n \mapsto n+1$.

5. A ***tag***-Turing machine has two heads: one can only read, the other can only write. Initially, the read head is located at the left end of the tape, and the write head is located at the first blank after the input string. At each transition, the read head can either move one cell to the right or stay put, but the write head *must* write a symbol to its current cell and move one cell to the right. Neither head can ever move to the left.

   Prove that any standard Turing machine can be simulated by a tag-Turing machine. That is, given any standard Turing machine M, describe a tag-Turing machine M' that accepts and rejects exactly the same input strings as M.

6. ⋆(a) Prove that any standard Turing machine can be simulated by a Turing machine with only three states. *[Hint: Use the tape to store an encoding of the state of the machine yours is simulating.]*

   ★(b) Prove that any standard Turing machine can be simulated by a Turing machine with only *two* states.

7. A ***two-dimensional*** Turing machine uses an infinite two-dimensional grid of cells as the tape; at each transition, the head can move from its current cell to any of its four neighbors on the grid. The transition function of such a machine has the form $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\uparrow, \leftarrow, \downarrow, \rightarrow\}$, where the arrows indicate which direction the head should move.

   (a) Prove that any two-dimensional Turing machine can be simulated by a standard Turing machine.

   (b) Suppose further that we endow our two-dimensional Turing machine with the following additional actions, in addition to moving the head:

   - Insert row: Move all symbols on or above the row containing the head up one row, leaving the head's row blank.
   - Insert column: Move all symbols on or to the right of the column containing the head one column to the right, leaving the head's column blank.
   - Delete row: Move all symbols above the row containing the head down one row, deleting the head's row of symbols.
   - Delete column: Move all symbols the right of the column containing the head one column to the right, deleting the head's column of symbols.

   Show that any two-dimensional Turing machine that can add an delete rows can be simulated by a standard Turing machine.

8. A ***binary-tree*** Turing machine uses an infinite binary tree as its tape; that is, *every* cell in the tape has a left child and a right child. At each step, the head moves from its current

cell to its Parent, its Left child, or to its Right child. Thus, the transition function of such a machine has the form $\delta: Q \times \Gamma \to Q \times \Gamma \times \{\mathsf{P}, \mathsf{L}, \mathsf{R}\}$. The input string is initially given along the left spine of the tape.

Show that any binary-tree Turing machine can be simulated by a standard Turing machine.

9. A ***stack-tape*** Turing machine uses an semi-infinite tape, where every cell is actually the top of an independent stack. The behavior of the machine at each iteration is governed by its internal state and the symbol *at the top* of the current cell's stack. At each transition, the head can optionally push a new symbol onto the stack, or pop the top symbol off the stack. (If a stack is empty, its "top symbol" is a blank and popping has no effect.)

   Show that any stack-tape Turing machine can be simulated by a standard Turing machine. (Compare with Problem 4!)

10. A ***tape-stack*** Turing machine has two actions that modify its work tape, in addition to simply writing individual cells: it can ***save*** the entire tape by pushing in onto a stack, and it can ***restore*** the entire tape by popping it off the stack. Restoring a tape returns the content of every cell to its content when the tape was saved. Saving and restoring the tape do not change the machine's state or the position of its head. If the machine attempts to "restore" the tape when the stack is empty, the machine crashes.

    Show that any tape-stack Turing machine can be simulated by a standard Turing machine.

★★★

- Tape alphabet = $\mathbb{N}$.
  - Read: zero or positive. Write: $+1$, -1
  - Read: even or odd. Write: $+1$, -1, $\times 2$, $\div 2$
  - Read: positive, negative, or zero. Write: $x + y$ (merge), $x - y$ (merge), 1, 0
- Never three times in a row in the same direction
- Hole-punch TM: tape alphabet $\{\square, \blacksquare\}$, and only $\square \mapsto \blacksquare$ transitions allowed.

Caveat lector: This note is not even a first draft, but more of a rough sketch, with many topics still to be written and/or unwritten. But the semester is over, so it's time to put it down. Please send bug reports and suggestions to jeffe@illinois.edu.

Any sufficiently advanced technology is indistinguishable from magic.
— Arthur C. Clarke, "Hazards of Prophecy: The Failure of Imagination" (1962)

Any technology that is distinguishable from magic is insufficiently advanced.
— Barry Gehm, quoted by Stan Schmidt in *ANALOG* magazine (1991)

7 Universal Models of Computation

★★★

Remind about the Church-Turing thesis.

There is some confusion here between **universal models of computation** and the somewhat wider class of **undecidable problems**.

7.1 Universal Turing Machines

The pinnacle of Turing machine constructions is the ***universal*** Turing machine. For modern computer scientists, it's useful to think of a universal Turing machine as a "Turing machine *interpreter* written in Turing machine". Just as the input to a Python interpreter is a string of Python source code, the input to our universal Turing machine U is a string $\langle M, w\rangle$ that encodes both an arbitrary Turing machine M and a string w in the input alphabet of M. Given these encodings, U simulates the execution of M on input w; in particular,

- U accepts $\langle M, w\rangle$ if and only if M accepts w.
- U rejects $\langle M, w\rangle$ if and only if M rejects w.

In the next few pages, I will sketch a universal Turing machine U that uses the input alphabet $\{\mathtt{0}, \mathtt{1}, \mathtt{[}, \mathtt{]}, \bullet, |\}$ and a somewhat larger tape alphabet. However, I do *not* require that the Turing machines that U simulates have similarly small alphabets, so we first need a method to encode *arbitrary* input and tape alphabets.

Encodings

Let $M = (\Gamma, \square, \Sigma, Q, \mathit{start}, \mathit{accept}, \mathit{reject}, \delta)$ be an arbitrary Turing machine, with a single half-infinite tape and a single read-write head. (I will consistently indicate the states and tape symbols of M in *slanted green* to distinguish them from the upright red states and tape symbols of U.)

We encode each symbol $a \in \Gamma$ as a unique string $|a|$ of $\lceil \lg(|\Gamma|)\rceil$ bits. Thus, if $\Gamma = \{\mathit{0}, \mathit{1}, \$, x, \square\}$, we might use the following encoding:

$$\langle \mathit{0} \rangle = \mathtt{001}, \qquad \langle \mathit{1} \rangle = \mathtt{010}, \qquad \langle \$ \rangle = \mathtt{011}, \qquad \langle x \rangle = \mathtt{100}, \qquad \langle \square \rangle = \mathtt{000}.$$

The input string w is encoded by its sequence of symbol encodings, with separators $\bullet$ between every pair of symbols and with brackets **[** and **]** around the whole string. For example, with this encoding, the input string $\mathit{001100}$ would be encoded on the input tape as

$$\langle \mathit{001100} \rangle = \mathtt{[001{\bullet}001{\bullet}010{\bullet}010{\bullet}001{\bullet}001]}$$

Similarly, we encode each state $q \in Q$ as a distinct string $\langle q\rangle$ of $\lceil \lg|Q| \rceil$ bits. Without loss of generality, we encode the start state with all `1`s and the reject state with all `0`s. For example, if $Q = \{\mathit{start}, \mathit{seek1}, \mathit{seek0}, \mathit{reset}, \mathit{verify}, \mathit{accept}, \mathit{reject}\}$, we might use the following encoding:

$$\begin{array}{cccc}
\langle \mathit{start}\rangle = \texttt{111} & \langle \mathit{seek1}\rangle = \texttt{010} & \langle \mathit{seek0}\rangle = \texttt{011} & \langle \mathit{reset}\rangle = \texttt{100} \\
\langle \mathit{verify}\rangle = \texttt{101} & \langle \mathit{accept}\rangle = \texttt{110} & \langle \mathit{reject}\rangle = \texttt{000} &
\end{array}$$

We encode the machine M itself as the string $\langle M\rangle = \texttt{[}\langle \mathit{reject}\rangle\bullet\langle\square\rangle\texttt{]}\langle\delta\rangle$, where $\langle\delta\rangle$ is the concatenation of substrings $\texttt{[}\langle p\rangle\bullet\langle a\rangle\texttt{|}\langle q\rangle\bullet\langle b\rangle\bullet\langle\Delta\rangle\texttt{]}$ encoding each transition $\delta(p,a) = (q,b,\Delta)$ such that $q \neq$ reject. We encode the actions $\Delta = \pm 1$ by defining $\langle -1\rangle := \texttt{0}$ and $\langle +1\rangle := \texttt{1}$. Conveniently, every transition string has exactly the same length. For example, with the symbol and state encodings described above, the transition $\delta(\mathit{reset}, \texttt{\$}) = (\mathit{start}, \texttt{\$}, +1)$ would be encoded as

$$\texttt{[100}\bullet\texttt{011|001}\bullet\texttt{011}\bullet\texttt{1]}.$$

Our first example Turing machine for recognizing $\{\mathtt{0}^n\mathtt{1}^n\mathtt{0}^n \mid n \geq 0\}$ would be represented by the following string (here broken into multiple lines for readability):

```
[000•000][[001•001|010•011•1][001•100|101•011•1]
          [010•001|010•001•1][010•100|010•100•1]
          [010•010|011•100•1][011•010|011•010•1]
          [011•100|011•100•1][011•001|100•100•1]
          [100•001|100•001•0][100•010|100•010•0]
          [100•100|100•100•0][100•011|001•011•1]
          [101•100|101•011•1][101•000|110•000•0]]
```

Finally, we encode any *configuration* of M on U's work tape by alternating between encodings of states and encodings of tape symbols. Thus, each tape cell is represented by the string $\texttt{[}\langle q\rangle\bullet\langle a\rangle\texttt{]}$ indicating that (1) the cell contains symbol a; (2) if $q \neq \mathit{reject}$, then M's head is located at this cell, and M is in state q; and (3) if $q = \mathit{reject}$, then M's head is located somewhere else. Conveniently, each cell encoding uses exactly the same number of bits. We also surround the entire tape encoding with brackets `[` and `]`.

For example, with the encodings described above, the initial configuration $(\mathit{start}, \underset{\blacktriangle}{\mathtt{0}}\mathtt{01100}, 0)$ for our first example Turing machine would be encoded on U's tape as follows.

$$\langle \mathit{start}, \underset{\blacktriangle}{\mathtt{0}}\mathtt{01100}, 0\rangle = \texttt{[}\underbrace{\texttt{[111}\bullet\texttt{001]}}_{\textbf{start }\mathtt{0}}\underbrace{\texttt{[000}\bullet\texttt{001]}}_{\mathit{reject}\ \mathtt{0}}\underbrace{\texttt{[000}\bullet\texttt{010]}}_{\mathit{reject}\ \mathtt{1}}\underbrace{\texttt{[000}\bullet\texttt{010]}}_{\mathit{reject}\ \mathtt{1}}\underbrace{\texttt{[000}\bullet\texttt{001]}}_{\mathit{reject}\ \mathtt{0}}\underbrace{\texttt{[000}\bullet\texttt{001]}}_{\mathit{reject}\ \mathtt{0}}\texttt{]}$$

Similarly, the intermediate configuration $(\mathit{reset}, \mathtt{\$0x}\underset{\blacktriangle}{\mathtt{1}}\mathtt{x0}, 3)$ would be encoded as follows:

$$\langle \mathit{reset}, \mathtt{\$\$x}\underset{\blacktriangle}{\mathtt{1}}\mathtt{x0}, 3\rangle = \texttt{[}\underbrace{\texttt{[000}\bullet\texttt{011]}}_{\mathit{reject}\ \mathtt{\$}}\underbrace{\texttt{[000}\bullet\texttt{011]}}_{\mathit{reject}\ \mathtt{0}}\underbrace{\texttt{[000}\bullet\texttt{100]}}_{\mathit{reject}\ \mathtt{x}}\underbrace{\texttt{[010}\bullet\texttt{010]}}_{\textbf{reset }\mathtt{1}}\underbrace{\texttt{[000}\bullet\texttt{100]}}_{\mathit{reject}\ \mathtt{x}}\underbrace{\texttt{[000}\bullet\texttt{001]}}_{\mathit{reject}\ \mathtt{0}}\texttt{]}$$

Input and Execution

Without loss of generality, we assume that the input to our universal Turing machine U is given on a separate read-only ***input tape***, as the encoding of an arbitrary Turing machine M followed by an encoding of its input string x. Notice the substrings `[[` and `]]` each appear only only once on the input tape, immediately before and after the encoded transition table, respectively. U also has a read-write ***work tape***, which is initially blank.

We start by initializing the work tape with the encoding $\langle \mathit{start}, x, 0\rangle$ of the initial configuration of M with input x. First, we write `[[`$\langle \mathit{start}\rangle$`•`. Then we copy the encoded input string $\langle x\rangle$ onto the work tape, but we change the punctuation as follows:

- Instead of copying the left bracket `[`, write `[[`$\langle \mathit{start}\rangle$`•`.
- Instead of copying each separator `•`, write `][`$\langle \mathit{reject}\rangle$`•`
- Instead of copying the right bracket `]`, write two right brackets `]]`.

The state encodings $\langle \mathit{start}\rangle$ and $\langle \mathit{reject}\rangle$ can be copied directly from the beginning of $\langle M\rangle$ (replacing `0`s for `1`s for $\langle \mathit{start}\rangle$). Finally, we move the head back to the start of U's tape.

At the start of each step of the simulation, U's head is located at the start of the work tape. We scan through the work tape to the unique encoded cell `[`$\langle p\rangle$`•`$\langle a\rangle$`]` such that $p \neq \mathit{reject}$. Then we scan through the encoded transition function $\langle \delta\rangle$ to find the unique encoded tuple `[`$\langle p\rangle$`•`$\langle a\rangle$`|`$\langle q\rangle$`•`$\langle b\rangle$`•`$\langle \Delta\rangle$`]` whose left half matches our the encoded tape cell. If there is no such tuple, then U immediately halts and rejects. Otherwise, we copy the right half $\langle q\rangle$`•`$\langle b\rangle$ of the tuple to the work tape. Now if $q = \mathit{accept}$, then U immediately halts and accepts. (We don't bother to encode *reject* transformations, so we know that $q \neq \mathit{reject}$.) Otherwise, we transfer the state encoding to either the next or previous encoded cell, as indicated by M's transition function, and then continue with the next step of the simulation.

During the final state-copying phase, we ever read two right brackets `]]`, indicating that we have reached the right end of the tape encoding, we replace the second right bracket with `[`$\langle \mathit{reject}\rangle$`•`$\langle \square\rangle$`]]` (mostly copied from the beginning of the machine encoding $\langle M\rangle$) and then scan back to the left bracket we just wrote. This trick allows our universal machine to *pretend* that its tape contains an infinite sequence of *encoded* blanks `[`$\langle \mathit{reject}\rangle$`•`$\langle \square\rangle$`]` instead of *actual* blanks $\square$.

Example

As an illustrative example, suppose U is simulating our first example Turing machine M on the input string `001100`. The execution of M on input w eventually reaches the configuration $(\mathit{seek1}, \$\$x1x0, 3)$. At the start of the corresponding step in U's simulation, U is in the following configuration:

`[[000•011][000•011][000•100][010•010][000•100][000•001]]`

First U scans for the first encoded tape cell whose state is not *reject*. That is, U repeatedly compares the first half of each encoded state cell on the work tape with the prefix `[`$\langle \mathit{reject}\rangle$`•` of the machine encoding $\langle M\rangle$ on the input tape. U finds a match in the fourth encoded cell.

`[[000•011][000•011][000•100][010•010][000•100][000•001]]`

Next, U scans the machine encoding $\langle M\rangle$ for the substring `[010•010` matching the current encoded cell. U eventually finds a match in the left size of the the encoded transition `[010•010|011•100•1]`. U copies the state-symbol pair `011•100` from the right half of this encoded transition into the current encoded cell. (The underline indicates which symbols are changed.)

$\texttt{[[000•011][000•011][000•100][}\underline{\texttt{011•100}}\texttt{][000•100][000•001]]}$

The encoded transition instructs U to move the current state encoding one cell to the right. (The underline indicates which symbols are changed.)

$\texttt{[[000•011][000•011][000•100][}\underline{\texttt{000}}\texttt{•100][}\underline{\texttt{011}}\texttt{•100][000•001]]}$

Finally, U scans left until it reads two left brackets [[; this returns the head to the left end of the work tape to start the next step in the simulation. U's tape now holds the encoding of M's configuration (*seek0*, *$$xxx0*, 4), as required.

[[000•011][000•011][000•100][000•100][011•100][000•001]]

7.2 Two-Stack Machines

A ***two-stack machine*** is a Turing machine with two tapes with the following restricted behavior. At all times, on each tape, every cell to the right of the head is blank, and every cell at or to the left of the head is non-blank. Thus, a head can only move right by writing a non-blank symbol into a blank cell; symmetrically, a head can only move left by erasing the rightmost non-blank cell. Thus, each tape behaves like a stack. To avoid underflow, there is a special symbol at the start of each tape that cannot be overwritten. Initially, one tape contains the input string, with the head at its *last* symbol, and the other tape is empty (except for the start-of-tape symbol).

★★★

Simulate a doubly-infinite tape with two stacks, one holding the tape contents to the left of the head, the other holding the tape contents to the right of the head. For each transition of a standard Turing machine M, the stack machine pops the top symbol off the (say) left stack, changes its internal state according to the transition δ, and then either pushes a new symbol onto the right stack, or pushes a new symbol onto the left stack and then moves the top symbol from the right stack to the left stack.

7.3 Counter Machines

★★★

A configuration of a k-counter machine consists of k non-negative integers and an internal state from some finite set Q. The transition function $\delta\colon Q \times \{0,+1\}^k \to Q \times \{-1,0,+1\}^k$ takes an internal state and the signs of the counters as input, and produces a new internal state and changes to counters as output.

- Prove that any Turing machine can be simulated by a three-counter machine. One counter holds the binary representation of the tape after the head; another counter holds the reversed binary representation of the tape before the head. Implement transitions via halving, doubling, and parity, using the third counter for scratch work.
- Prove that two counters can simulate three. Store $2^a 3^b 5^c$ in one counter, use the other for scratch work.
- Prove that a three-counter machine can compute any computable function: Given input $(n,0,0)$, we can compute $(f(n),0,0)$ for *any* computable function f. First transform $(n,0,0)$ to $(2^n,0,0)$ using all three counters; then run two- (or three-)counter TM simulation to obtain $(2^{f(n)},0,0)$; and finally transform $(2^{f(n)},0,0)$ to $(f(n),0,0)$ using all three counters.
- **HARD:** Prove that a two-counter machine cannot transform $(n,0)$ to $(2^n,0)$. [Barzhdin 1963, Yao 1971, Schröpel 1972]

7.4 FRACTRAN

★★★

FRACTRAN [Conway 1987]: A one-counter machine whose "program" is a sequence of rational numbers. The counter is initially 1. At each iteration, multiply the counter by the first rational number that yields an integer; if there is no such number, halt.

- Prove that for any computable function $f : \mathbb{N} \to \mathbb{N}$, there is a FRACTRAN program that transforms 2^{n+1} into $3^{f(n)+1}$, for all natural numbers n.
- Prove that every FRACTRAN program, given the integer 1 as input, either outputs 1 or loops forever. It follows that there is no FRACTRAN program for the increment function $n \mapsto n+1$.

7.5 Post Correspondence Problem

Given n of pairs of strings $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$, is there a finite sequence of integers $(i_1, i_2, \ldots, i_k)$ such that $x_{i_1} x_{i_2} \cdots x_{i_k} = y_{i_1} y_{i_2} \cdots y_{i_k}$? For notation convenience, we write each pair vertically as $\left[\begin{smallmatrix} x \\ y \end{smallmatrix}\right]$ instead of horizontally as (x, y). For example, given the string pairs

$$a = \begin{bmatrix} \texttt{0} \\ \texttt{100} \end{bmatrix}, \; b = \begin{bmatrix} \texttt{01} \\ \texttt{00} \end{bmatrix}, \; c = \begin{bmatrix} \texttt{110} \\ \texttt{11} \end{bmatrix},$$

we should answer TRUE, because

$$cbca = \begin{bmatrix} \texttt{110} \\ \texttt{11} \end{bmatrix} \begin{bmatrix} \texttt{01} \\ \texttt{00} \end{bmatrix} \begin{bmatrix} \texttt{110} \\ \texttt{11} \end{bmatrix} \begin{bmatrix} \texttt{0} \\ \texttt{100} \end{bmatrix}$$

gives us `110110100` for both concatenations. As more extreme examples, the shortest solutions for the input

$$a = \begin{bmatrix} \texttt{0} \\ \texttt{001} \end{bmatrix}, \; b = \begin{bmatrix} \texttt{001} \\ \texttt{1} \end{bmatrix}, \; c = \begin{bmatrix} \texttt{1} \\ \texttt{0} \end{bmatrix}$$

have length 75; one such solution is $aacaacabbabccaaccaaaacbaabbaacbacbbccbbacbaccbcb$ $acbbacbaccbacbbbacccbabbccbaacaacaaacbabbaacacbccbbabacbcaaccbacabbbbabcccc$ $bcaababaaccbcbbbacccbabbccb$. The shortest solution for the instance

$$a = \begin{bmatrix} \texttt{0} \\ \texttt{000} \end{bmatrix}, \; b = \begin{bmatrix} \texttt{0} \\ \texttt{0101} \end{bmatrix}, \; c = \begin{bmatrix} \texttt{01} \\ \texttt{1} \end{bmatrix}, \; d = \begin{bmatrix} \texttt{1111} \\ \texttt{10} \end{bmatrix}$$

is the unbelievable $a^2b^8a^4c^{16}ab^4a^2b^4ad^4b^3c^8a^6c^8b^2c^4bc^6d^2a^{18}d^2c^4dcad^2cb^{54}c^3dca^2c^{111}dc$ $a^6d^{28}cb^{17}c^{63}d^{16}c^{16}d^4c^4dc$, which has total length 451. Finally, the shortest solution for the instance

$$a = \begin{bmatrix} \texttt{0} \\ \texttt{00010} \end{bmatrix}, \; b = \begin{bmatrix} \texttt{010} \\ \texttt{01} \end{bmatrix}, \; c = \begin{bmatrix} \texttt{100} \\ \texttt{0} \end{bmatrix},$$

has length 528.

★★★

The simplest universality proof simulates a tag-Turing machine.

7.6 Matrix Mortality

★★★

Given a set of integer matrices $A_1, \ldots, A_k$, is the product of any sequence of these matrices (with repetition) equal to 0? Undecidable by reduction from PCP, even for two 15×15 matrices or six 3×3 matrices [Cassaigne, Halava, Harju, Nicolas 2014]

7.7 Dynamical Systems

★★★

Ray Tracing [Reif, Tygar, and Yoshida 1994] The configuration of a Turing machine is encoded as the (x, y) coordinates of a light path crossing the unit square $[0,1] \times [0,1]$, where the x-(resp. y-)coordinate encodes the tape contents to the left (resp. right) of the head. Need either quadratic-surface mirrors or refraction to simulate transitions.

N-body problem [Smith 2006]: Similar idea

Skolem-Pisot reachability: Given an integer vector x and an integer matrix A, does $A^n x = (0, \ldots)$ for any integer n? [Halava, Harju, Hirvensalo, Karhumäki 2005] It's surprising that this problem is undecidable; the similar mortality problem for one matrix is not.

7.8 Wang Tiles

★★★

Turing machine simulation is straightforward. *Small* Turing-complete tile sets via affine maps (via two-stack machines) are a little harder.

7.9 Combinator Calculus

In the 1920s, Moses Schönfinkel developed what can now be interpreted as a model of computation now called *combinator calculus* or *combinatory logic*. Combinator calculus operates on ***terms***, where every term is either one of a finite number of ***combinators*** (represented here by upper case letters) or an ordered pair of terms. For notational convenience, we omit commas between components of every pair and parentheses around the *left* term in every pair. Thus, $\mathtt{SKK}(\mathtt{IS})$ is shorthand for the term $(((\mathtt{S},\mathtt{K}),\mathtt{K}),(\mathtt{I},\mathtt{S}))$.

We can "evaluate" any term by a sequence of rewriting rules that depend on its first primitive combinator. Schönfinkel defined three primitive combinators with the following evaluation rules:

- Identity: $\mathtt{I}x \mapsto x$
- Constant: $\mathtt{K}xy \mapsto x$
- Substitution: $\mathtt{S}xyz \mapsto xz(yz)$

Here, x, y, and z are variables representing unknown but arbitrary terms. "Computation" in the combinator calculus is performed by repeatedly evaluating arbitrary (sub)terms with one of these three structures, until all such (sub)terms are gone.

For example, the term $\mathtt{S}(\mathtt{K}(\mathtt{SI}))\mathtt{K}xy$ (for any terms x and y) evaluates as follows:

$$
\begin{aligned}
\underline{\mathtt{S}(\mathtt{K}(\mathtt{SI}))\mathtt{K}x}y &\mapsto \underline{\mathtt{K}(\mathtt{SI})x}(\mathtt{K}x)y && \text{Substitution}\\
&\mapsto \underline{\mathtt{SI}(\mathtt{K}x)y} && \text{Constant}\\
&\mapsto \underline{\mathtt{I}y}(\mathtt{K}xy) && \text{Substitution}\\
&\mapsto y(\underline{\mathtt{K}xy}) && \text{Identity}\\
&\mapsto yx && \text{Constant}
\end{aligned}
$$

Thus, we can define a new combinator $\mathtt{R} := \mathtt{S}(\mathtt{K}(\mathtt{SI}))\mathtt{K}$ that upon evaluation reverses the next two terms: $\mathtt{R}xy \mapsto yx$.

On the other hand, evaluating `SII(S(KI)(SII))` leads to an infinite loop:

$$
\begin{aligned}
\underline{\mathtt{SII(S(KI)(SII))}} &\mapsto \underline{\mathtt{I(S(KI)(SII))}}\mathtt{(I(S(KI)(SII)))} && \text{Substitution}\\
&\mapsto \mathtt{S(KI)(SII)(}\underline{\mathtt{I(S(KI)(SII))}}\mathtt{)} && \text{Identity}\\
&\mapsto \underline{\mathtt{S(KI)(SII)(S(KI)(SII))}} && \text{Identity}\\
&\mapsto \underline{\mathtt{KI(S(KI)(SII))}}\mathtt{(SII(S(KI)(SII)))} && \text{Substitution}\\
&\mapsto \underline{\mathtt{I(SII(S(KI)(SII)))}} && \text{Constant}\\
&\mapsto \mathtt{SII(S(KI)(SII))} && \text{Identity}
\end{aligned}
$$

★★★ Wikipedia sketches a direct *undecidability* proof. Is there a Turing-completeness proof that avoids λ-calculus?

Exercises

1. A ***tag***-Turing machine has two heads: one can only read, the other can only write. Initially, the read head is located at the left end of the tape, and the write head is located at the first blank after the input string. At each transition, the read head can either move one cell to the right or stay put, but the write head *must* write a symbol to its current cell and move one cell to the right. Neither head can ever move to the left.

   Prove that any standard Turing machine can be simulated by a tag-Turing machine. That is, given any standard Turing machine M, describe a tag-Turing machine M' that accepts and rejects exactly the same input strings as M.

2. *(a) Prove that any standard Turing machine can be simulated by a Turing machine with only three states. *[Hint: Use the tape to store an encoding of the state of the machine yours is simulating.]*

   $^\bigstar$(b) Prove that any standard Turing machine can be simulated by a Turing machine with only *two* states.

3. A ***two-dimensional*** Turing machine uses an infinite two-dimensional grid of cells as the tape; at each transition, the head can move from its current cell to any of its four neighbors on the grid. The transition function of such a machine has the form $\delta\colon Q \times \Gamma \to Q \times \Gamma \times \{\uparrow, \leftarrow, \downarrow, \rightarrow\}$, where the arrows indicate which direction the head should move.

   (a) Prove that any two-dimensional Turing machine can be simulated by a standard Turing machine.

   (b) Suppose further that we endow our two-dimensional Turing machine with the following additional actions, in addition to moving the head:

   - Insert row: Move all symbols on or above the row containing the head up one row, leaving the head's row blank.
   - Insert column: Move all symbols on or to the right of the column containing the head one column to the right, leaving the head's column blank.
   - Delete row: Move all symbols above the row containing the head down one row, deleting the head's row of symbols.
   - Delete column: Move all symbols the right of the column containing the head one column to the right, deleting the head's column of symbols.

Show that any two-dimensional Turing machine that can add an delete rows can be simulated by a standard Turing machine.

4. A ***binary-tree*** Turing machine uses an infinite binary tree as its tape; that is, *every* cell in the tape has a left child and a right child. At each step, the head moves from its current cell to its Parent, its Left child, or to its Right child. Thus, the transition function of such a machine has the form $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\mathsf{P}, \mathsf{L}, \mathsf{R}\}$. The input string is initially given along the left spine of the tape.

   Show that any binary-tree Turing machine can be simulated by a standard Turing machine.

5. A ***stack-tape*** Turing machine uses an semi-infinite tape, where every cell is actually the top of an independent stack. The behavior of the machine at each iteration is governed by its internal state and the symbol *at the top* of the current cell's stack. At each transition, the head can optionally push a new symbol onto the stack, or pop the top symbol off the stack. (If a stack is empty, its "top symbol" is a blank and popping has no effect.)

   Show that any stack-tape Turing machine can be simulated by a standard Turing machine. (Compare with Problem **??**!)

6. A ***tape-stack*** Turing machine has two actions that modify its work tape, in addition to simply writing individual cells: it can ***save*** the entire tape by pushing in onto a stack, and it can ***restore*** the entire tape by popping it off the stack. Restoring a tape returns the content of every cell to its content when the tape was saved. Saving and restoring the tape do not change the machine's state or the position of its head. If the machine attempts to "restore" the tape when the stack is empty, the machine crashes.

   Show that any tape-stack Turing machine can be simulated by a standard Turing machine.

★★★

- Tape alphabet = $\mathbb{N}$.
  - Read: zero or positive. Write: $+1$, -1
  - Read: even or odd. Write: $+1$, -1, $\times 2$, $\div 2$
  - Read: positive, negative, or zero. Write: $x + y$ (merge), $x - y$ (merge), 1, 0
- Never three times in a row in the same direction
- Hole-punch TM: tape alphabet $\{\square, \blacksquare\}$, and only $\square \mapsto \blacksquare$ transitions allowed.

Caveat lector: This is the zeroth (draft) edition of this lecture note. Please send bug reports and suggestions to jeffe@illinois.edu.

I said in my haste, All men are liars.

— Psalms 116:11 (King James Version)

yields falsehood when preceded by its quotation.

— William V. Quine, "Paradox", *Scientific American* (1962)

Some problems are so complex that you have to be highly intelligent and well informed just to be undecided about them.

— Laurence Johnston Peter, *Peter's Almanac* (September 24, 1982)

"Proving or disproving a formula—once you've encrypted the formula into numbers, that is—is just a calculation on that number. So it means that the answer to the question is, no! Some formulas cannot be proved or disproved by any mechanical process! So I guess there's some point in being human after all!"

Alan looked pleased until Lawrence said this last thing, and then his face collapsed. "Now there you go making unwarranted assumptions."

— Neal Stephenson, *Cryptonomicon* (1999)

No matter how P might perform, Q will scoop it:
Q uses P's output to make P look stupid.
Whatever P says, it cannot predict Q:
P is right when it's wrong, and is false when it's true!

— Geoffrey S. Pullum, "Scooping the Loop Sniffer" (2000)

This castle is in unacceptable condition! ***UNACCEPTABLE!!***

— Earl of Lemongrab [Justin Roiland], "Too Young"
Adventure Time (August 8, 2011)

8 Undecidability

Perhaps the single most important result in Turing's remarkable 1936 paper is his solution to Hilbert's *Entscheidungsproblem*, which asked for a general automatic procedure to determine whether a given statement of first-order logic is *provable*. Turing proved that no such procedure exists; there is no systematic way to distinguish between statements that cannot be proved even in principle and statements whose proofs we just haven't found yet.

8.1 Acceptable versus Decidable

Recall that there are three possible outcomes for a Turing machine M running on any particular input string w: acceptance, rejection, and divergence. Every Turing machine M immediately defines four different languages (over the input alphabet Σ of M):

- The *accepting* language $\textsc{Accept}(M) := \{w \in \Sigma^* \mid M \text{ accepts } w\}$
- The *rejecting* language $\textsc{Reject}(M) := \{w \in \Sigma^* \mid M \text{ rejects } w\}$
- The *halting* language $\textsc{Halt}(M) := \textsc{Accept}(M) \cup \textsc{Reject}(M)$
- The *diverging* language $\textsc{Diverge}(M) := \Sigma^* \setminus \textsc{Halt}(M)$

For any language L, the sentence "***M accepts L***" means $\textsc{Accept}(M) = L$, and the sentence "***M decides L***" means $\textsc{Accept}(M) = L$ and $\textsc{Diverge}(M) = \varnothing$.

Now let L be an arbitrary language. We say that L is ***acceptable*** (or *semi-computable*, or *semi-decidable*, or *recognizable*, or *listable*, or *recursively enumerable*) if some Turing machine accepts L, and ***unacceptable*** otherwise. Similarly, L is ***decidable*** (or *computable*, or *recursive*) if some Turing machine decides L, and ***undecidable*** otherwise.

8.2 Lo, I Have Become Death, Stealer of Pie

There is a subtlety in the definitions of "acceptable" and "decidable" that many beginners miss: A language can be decidable even if we can't exhibit a specific Turing machine decides it. As a canonical example, consider the language $\Pi = \{w \mid \mathtt{1}^{|w|} \text{ appears in the binary expansion of } \pi\}$. Despite appearances, this language *is* decidable! There are only two cases to consider:

- Suppose there is an integer N such that the binary expansion of π contains the substring $\mathtt{1}^N$ but does not contain the substring $\mathtt{1}^{N+1}$. Let M_N be the Turing machine with $N+3$ states $\{0, 1, \ldots, N, \mathsf{accept}, \mathsf{reject}\}$, start state 0, and the following transition function:

$$\delta(q, a) = \begin{cases} \mathsf{accept} & \text{if } a = \square \\ \mathsf{reject} & \text{if } a \neq \square \text{ and } q = n \\ (q+1, a, +1) & \text{otherwise} \end{cases}$$

  This machine correctly decides Π.

- Suppose the binary expansion of π contains arbitrarily long substrings of $\mathtt{1}$s. Then any Turing machine that accepts all inputs correctly decides Π.

We have no idea which of these machines correctly decides Π, but one of them does, and that's enough!

8.3 Useful Lemmas

This subsection lists several simple but useful properties of (un)decidable and (un)acceptable languages. For almost all of these properties, the proofs are straightforward; readers are strongly encouraged to try to prove each lemma themselves before reading ahead.

One might reasonably ask why we don't also define "rejectable" and "haltable" languages. The following lemma, whose proof is an easy exercise (hint, hint), implies that these are both identical to the acceptable languages.

Lemma 1. *Let M be an arbitrary Turing machine.*
(a) There is a Turing machine M^R such that $\textsc{Accept}(M^R) = \textsc{Reject}(M)$ and $\textsc{Reject}(M^R) = \textsc{Accept}(M)$.
(b) There is a Turing machine M^A such that $\textsc{Accept}(M^A) = \textsc{Accept}(M)$ and $\textsc{Reject}(M^A) = \varnothing$.
(c) There is a Turing machine M^H such that $\textsc{Accept}(M^H) = \textsc{Halt}(M)$ and $\textsc{Reject}(M^H) = \varnothing$.

The decidable languages have several fairly obvious useful properties.

Lemma 2. *If L and L' are decidable, then $L \cup L'$, $L \cap L'$, $L \setminus L'$, and $L' \setminus L$ are also decidable.*

Proof: Let M and M' be Turing machines that decide L and L', respectively. We can build a Turing machine $M_\cup$ that decides $L \cup L'$ as follows. First, $M_\cup$ copies its input string w onto a second tape. Then $M_\cup$ runs M on input w (on the first tape), and then runs M' on input w (on the second tape). If either M or M' accepts, then $M_\cup$ accepts; if both M and M' reject, then $M_\cup$ rejects.

The other three languages are similar. □

Corollary 3. *The following hold for all languages L and L'.*
(a) If $L \cap L'$ is undecidable and L' is decidable, then L is undecidable.
(b) If $L \cup L'$ is undecidable and L' is decidable, then L is undecidable.
(c) If $L \setminus L'$ is undecidable and L' is decidable, then L is undecidable.
(d) If $L' \setminus L$ is undecidable and L' is decidable, then L is undecidable.

The asymmetry between acceptance and rejection implies that merely acceptable languages are not quite as well-behaved as decidable languages.

Lemma 4. *For all acceptable languages L and L', the languages $L \cup L'$ and $L \cap L'$ are also acceptable.*

Proof: Let M and M' be Turing machines that decide L and L', respectively. We can build a Turing machine $M_\cap$ that decides $L \cap L'$ as follows. First, $M_\cap$ copies its input string w onto a second tape. Then $M_\cap$ runs M on input w using the first tape, and then runs M' on input w using the second tape. If both M and M' accept, then $M_\cap$ accepts; if either M or M' reject, then $M_\cap$ rejects; if either M or M' diverge, then $M_\cap$ diverges (automatically).

The construction for $L \cup L'$ is more subtle; instead of running M and M' in series, we must run them in parallel. Like $M_\cap$, the new machine $M_\cup$ starts by copying its input string w onto a second tape. But then $M_\cup$ runs M and M' simultaneously; with each step of $M_\cup$ simulating both one step of M on the first tape and one step of M' on the second. Ignoring the states and transitions needed for initialization, the state set of $M_\cup$ is the product of the state sets of M and M', and the transition function is

$$\delta_\cup(q,a,q',a') = \begin{cases} \textsf{accept}_\cup & \text{if } q = \textsf{accept} \text{ or } q' = \textsf{accept}' \\ \textsf{reject}_\cup & \text{if } q = \textsf{reject} \text{ and } q' = \textsf{reject}' \\ (\delta(q,a), \delta'(q',a')) & \text{otherwise} \end{cases}$$

Thus, $M_\cup$ accepts as soon as either M or M' accepts, and rejects only after both M or M' reject. □

Lemma 5. *An acceptable language L is decidable if and only if $\Sigma^* \setminus L$ is also acceptable.*

Proof: Let M and $\overline{M}$ be Turing machines that accept L and $\Sigma^* \setminus L$, respectively. Following the previous proof, we construct a new Turing machine M^* that copies its input onto a second tape, and then simulates M and M' in parallel on the two tapes. If M accepts, then M^* accepts; if $\overline{M}$ accepts, then M^* rejects. Since every string is accepted by either M or $\overline{M}$, we conclude that M^* decides L.

The other direction follows immediately from Lemma 1. □

8.4 Self-Haters Gonna Self-Hate

Let U be an arbitrary fixed universal Turing machine. Any Turing machine M can be encoded as a string $\langle M\rangle$ of symbols from U's input alphabet, so that U can simulate the execution of M on any suitably encoded input string. Different universal Turing machines require different encodings.[1]

A Turing machine encoding is just a string, and any string (over the correct alphabet) can be used as the input to a Turing machine. Thus, we can use the encoding $\langle M\rangle$ of any Turing machine M as the input to another Turing machine. We've already seen an example of this ability in our universal Turing machine U, but more significantly, we can use $\langle M\rangle$ as the input to *the same Turing machine M*. Thus, each of the following languages is well-defined:

$$\begin{aligned}
\textsc{SelfAccept} &:= \big\{\langle M\rangle \bigm| M \text{ accepts } \langle M\rangle\big\}\\
\textsc{SelfReject} &:= \big\{\langle M\rangle \bigm| M \text{ rejects } \langle M\rangle\big\}\\
\textsc{SelfHalt} &:= \big\{\langle M\rangle \bigm| M \text{ halts on } \langle M\rangle\big\}\\
\textsc{SelfDiverge} &:= \big\{\langle M\rangle \bigm| M \text{ diverges on } \langle M\rangle\big\}
\end{aligned}$$

One of Turing's key observations is that ***SelfReject* is undecidable**; Turing proved this theorem by contradiction as follows:

Suppose to the contrary that there is a Turing machine SR such that $\textsc{Accept}(SR) = \textsc{SelfReject}$ and $\textsc{Diverge}(SR) = \varnothing$. More explicitly, for ***any*** Turing machine M,

- SR accepts $\langle M\rangle \iff M$ rejects $\langle M\rangle$, and
- SR rejects $\langle M\rangle \iff M$ does not reject $\langle M\rangle$.

In particular, these equivalences must hold when M is equal to SR. Thus,

- SR accepts $\langle SR\rangle \iff SR$ rejects $\langle SR\rangle$, and
- SR rejects $\langle SR\rangle \iff SR$ does not reject $\langle SR\rangle$.

In short, SR accepts $\langle SR\rangle$ if and only if SR rejects $\langle SR\rangle$, which is impossible! The only logical conclusion is that the Turing machine SR does not exist!

8.5 Aside: Uncountable Barbers

Turing's proof by contradiction is nearly identical to the famous ***diagonalization argument*** that uncountable sets exist, published by Georg Cantor in 1891. Indeed, $\textsc{SelfReject}$ is sometimes called "the diagonal language". Recall that a function $f : A \to B$ is a ***surjection***[2] if $f(A) = \{f(a) \mid a \in A\} = B$.

Cantor's Theorem. *Let $f : X \to 2^X$ be an* **arbitrary** *function from an* **arbitrary** *set X to its power set. This function f is not a surjection.*

[1]In fact, these undecidability proofs never actually use the universal Turing machine; all we really need is an encoding function that associates a unique string $\langle M\rangle$ with every Turing machine M. However, we *do* need the encoding to be compatible with a universal Turing machine for the results in Section **??**.

[2]more commonly, flouting all reasonable standards of grammatical English, "an onto function"

Proof: Fix an arbitrary function $f : X \to 2^X$. Call an element $x \in X$ ***happy*** if $x \in f(x)$ and ***sad*** if $x \notin f(x)$. Let Y be the set of all sad elements of X; that is, for *every* element $x \in X$, we have

$$x \in Y \iff x \notin f(x).$$

For the sake of argument, suppose f is a surjection. Then (by definition of surjection) there must be an element $y \in X$ such that $f(y) = Y$. Then for *every* element $x \in X$, we have

$$x \in f(y) \iff x \notin f(x).$$

In particular, the previous equivalence must hold when $x = y$:

$$y \in f(y) \iff y \notin f(y).$$

We have a contradiction! We conclude that f is not a surjection after all. □

Now let $X = \Sigma^*$, and define the function $f : X \to 2^X$ as follows:

$$f(w) := \begin{cases} \textsc{Accept}(M) & \text{if } w = \langle M \rangle \text{ for some Turing machine } M \\ \varnothing & \text{if } w \text{ is not the encoding of a Turing machine} \end{cases}$$

Cantor's theorem immediately implies that not all languages are acceptable.

Alternatively, let X be the set of all Turing machines that halt on all inputs. For any Turing machine $M \in X$, let $f(M)$ be the set of all Turing machines $N \in X$ such that M accepts the encoding $\langle N \rangle$. Then a Turing machine M is *sad* if it rejects its own encoding $\langle M \rangle$; thus, Y is essentially the set SelfReject. Cantor's argument now immediately implies that no Turing machine decides the language SelfReject.

The core of Cantor's diagonalization argument also appears in the "barber paradox" popularized by Bertrand Russell in the 1910s. In a certain small town, every resident has a haircut on Haircut Day. Some residents cut their own hair; others have their hair cut by another resident of the same town. To obtain an official barber's license, a resident must cut the hair of all residents who don't cut their own hair, and no one else. Given these assumptions, we can immediately conclude that there are no licensed barbers. After all, who would cut the barber's hair?

To map Russell's barber paradox back to Cantor's theorem, let X be the set of residents, and let $f(x)$ be the set of residents who have their hair cut by x; then a resident is *sad* if they do not cut their own hair. To prove that SelfReject is undecidable, replace "resident" with "a Turing machine that halts on all inputs", and replace "A cuts B's hair" with "A accepts $\langle B \rangle$".

8.6 Just Don't Know What to Do with Myself

Similar diagonal arguments imply that the other three languages SelfAccept, SelfHalt, and Self-Diverge are also undecidable. The proofs are not quite as direct for these three languages as the proof for SelfReject; each fictional deciding machine requires a small modification to create the contradiction.

Theorem 6. *SelfAccept is undecidable.*

Proof: For the sake of argument, suppose there is a Turing machine SA such that $\textsc{Accept}(SA) = \textsc{SelfAccept}$ and $\textsc{Diverge}(M) = \varnothing$. Let SA^R be the Turing machine obtained from SA by swapping its accept and reject states (as in the proof of Lemma 1). Then $\textsc{Reject}(SA^R) = \textsc{SelfAccept}$ and $\textsc{Diverge}(SA^R) = \varnothing$. It follows that SA^R rejects $\langle SA^R \rangle$ if and only if SA^R accepts $\langle SA^R \rangle$, which is impossible. □

Theorem 7. *SelfHalt is undecidable.*

Proof: Suppose to the contrary that there is a Turing machine SH such that $\text{Accept}(SH) = \text{SelfHalt}$ and $\text{Diverge}(SH) = \varnothing$. Let SH^X be the Turing machine obtained from SH by redirecting every transition to accept to a new hanging state hang, and then redirecting every transition to reject to accept. Then $\text{Accept}(SH^X) = \Sigma^* \setminus \text{SelfHalt}$ and $\text{Reject}(SH^X) = \varnothing$. It follows that SH^X accepts $\langle SH^X \rangle$ if and only if SH^X does not halt on $\langle SH^X \rangle$, and we have a contradiction. □

Theorem 8. *SelfDiverge is* ***unacceptable and therefore*** *undecidable.*

Proof: Suppose to the contrary that there is a Turing machine SD such that $\text{Accept}(M) = \text{SelfDiverge}$. Let SD^A be the Turing machine obtained from M by redirecting every transition to reject to a new hanging state hang such that $\delta(\text{hang}, a) = (\text{hang}, a, +1)$ for every symbol a. Then $\text{Accept}(SD^A) = \text{SelfDiverge}$ and $\text{Reject}(SD^A) = \varnothing$. It follows that SD^A accepts $\langle SD^A \rangle$ if and only if SD^A does not halt on $\langle SD^A \rangle$, which is impossible. □

*8.7 Nevertheless, Acceptable

Our undecidability argument for SelfDiverge actually implies the stronger result that SelfDiverge is unacceptable; we never assumed that the hypothetical accepting machine SD halts on all inputs. However, we can use or modify our universal Turing machine to accept the other three languages.

Theorem 9. *SelfAccept is acceptable.*

Proof: We describe a Turing machine SA that accepts the language SelfAccept. Given any string w as input, SA first verifies that w is the encoding of a Turing machine. If w is not the encoding of a Turing machine, then SA diverges. Otherwise, $w = \langle M \rangle$ for some Turing machine M; in this case, SA writes the string $ww = \langle M \rangle \langle M \rangle$ onto its tape and passes control to the universal Turing machine U. U then simulates M (the machine encoded by the first half of its input) on the string $\langle M \rangle$ (the second half of its input).[3] In particular, U accepts $\langle M, M \rangle$ if and only if M accepts $\langle M \rangle$. We conclude that SR accepts $\langle M \rangle$ if and only if M accepts $\langle M \rangle$. □

Theorem 10. *SelfReject is acceptable.*

Proof: Let U^R be the Turing machine obtained from our universal machine U by swapping the accept and reject states. We describe a Turing machine SR that accepts the language SelfReject as follows. SR first verifies that its input string w is the encoding of a Turing machine and diverges if not. Otherwise, SR writes the string $ww = \langle M, M \rangle$ onto its tape and passes control to the reversed universal Turing machine U^R. Then U^R accepts $\langle M, M \rangle$ if and only if M rejects $\langle M \rangle$. We conclude that SR accepts $\langle M \rangle$ if and only if M rejects $\langle M \rangle$. □

Finally, because SelfHalt is the union of two acceptable languages, SelfHalt is also acceptable.

[3] To simplify the presentation, I am implicitly assuming here that $\langle M \rangle = \langle \langle M \rangle \rangle$. Without this assumption, we need a Turing machine that transforms an arbitrary string $w \in \Sigma_M^*$ into its encoding $\langle w \rangle$ for U; building such a Turing machine is straightforward.

8.8 The Halting Problem via Reduction

Consider the following related languages:[4]

$$\begin{aligned}
\textsc{Accept} &:= \big\{\langle M, w\rangle \bigm| M \text{ accepts } w\big\}\\
\textsc{Reject} &:= \big\{\langle M, w\rangle \bigm| M \text{ rejects } w\big\}\\
\textsc{Halt} &:= \big\{\langle M, w\rangle \bigm| M \text{ halts on } w\big\}\\
\textsc{Diverge} &:= \big\{\langle M, w\rangle \bigm| M \text{ diverges on } w\big\}
\end{aligned}$$

Deciding the language Halt is what is usually meant by the *halting problem*: Given a program M and an input w to that program, does the program halt? This problem may seem trivial; why not just run the program and see? More formally, why not just pass the input string $\langle M, x\rangle$ to our universal Turing machine U? That strategy works perfectly if we just want to *accept* Halt, but we actually want to *decide* Halt; if M is not going to halt on w, we still want an answer in a finite amount of time. Sadly, we can't always get what we want.

Theorem 11. *Halt is undecidable.*

Proof: Suppose to the contrary that there is a Turing machine H that decides Halt. Then we can use H to build another Turing machine SH that decides the language SelfHalt. Given any string w, the machine SH first verifies that $w = \langle M\rangle$ for some Turing machine M (rejecting if not), then writes the string $ww = \langle M, M\rangle$ onto the tape, and finally passes control to H. But SelfHalt is undecidable, so no such machine SH exists. We conclude that H does not exist either. □

Nearly identical arguments imply that the languages Accept, Reject, and Diverge are undecidable.

Here we have our first example of an undecidability proof by ***reduction***. Specifically, we ***reduced*** the language SelfHalt to the language Halt. More generally, to reduce one language X to another language Y, we assume (for the sake of argument) that there is a program P_Y that decides Y, and we write another program that decides X, using P_Y as a black-box subroutine. If later we discover that Y is decidable, we can immediately conclude that X is decidable. Equivalently, if we later discover that X is undecidable, we can immediately conclude that Y is undecidable.

To prove that a language *L* is undecidable,
reduce a known undecidable language to *L*.

Perhaps the most confusing aspect of reduction arguments is that the *languages* we want to prove undecidable nearly (but not quite) always involve encodings of Turing machines, while at the same time, the *programs* that we build to prove them undecidable are also Turing machines. Our proof that Halt is undecidable involved three different machines:

- The hypothetical Turing machine H that decides Halt.
- The new Turing machine SH that decides SelfHalt, using H as a subroutine.

[4]Sipser uses the shorter name A_{TM} instead of Accept, but uses $HALT_{TM}$ instead of Halt. I have no idea why he thought four-letter names are okay, but six-letter names are not. His subscript TM is just a reminder that these are languages of *Turing machine* encodings, as opposed to encodings of DFAs or some other machine model.

- The Turing machine M whose encoding is the input to H.

It is *incredibly* easy to get confused about which machines are playing each in the proof. Therefore, it is absolutely vital that we give each machine in a reduction proof a unique and mnemonic name, and then ***always*** refer to each machine ***by name***. Never write, say, or even *think* "the machine" or "that machine" or (gods forbid) "it". You also may find it useful to think of the working ***programs*** we are trying to construct (H and SH in this proof) as being written in a different language than the arbitrary ***source code*** that we want those programs to analyze ($\langle M \rangle$ in this proof).

8.9 One Million Years Dungeon!

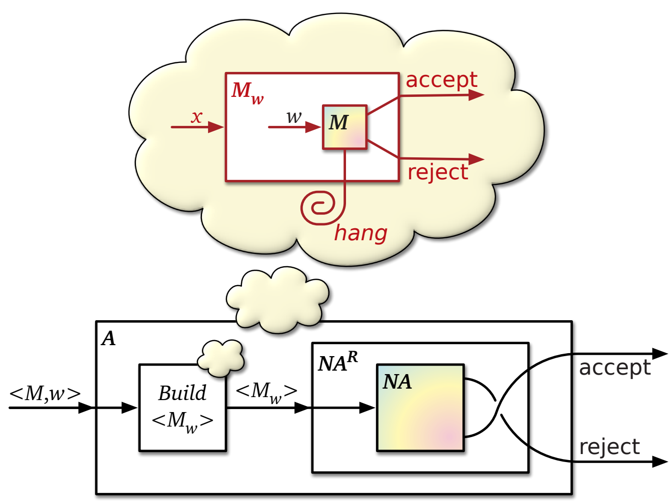
As a more complex set of examples, consider the following languages:

$$\begin{aligned}
\text{NeverAccept} &:= \left\{\langle M \rangle \mid \text{Accept}(M) = \varnothing\right\} \\
\text{NeverReject} &:= \left\{\langle M \rangle \mid \text{Reject}(M) = \varnothing\right\} \\
\text{NeverHalt} &:= \left\{\langle M \rangle \mid \text{Halt}(M) = \varnothing\right\} \\
\text{NeverDiverge} &:= \left\{\langle M \rangle \mid \text{Diverge}(M) = \varnothing\right\}
\end{aligned}$$

Theorem 12. *NeverAccept is undecidable.*

Proof: Suppose to the contrary that there is a Turing machine NA that decides NeverAccept. Then by swapping the accept and reject states, we obtain a Turing machine NA^R that decides the complementary language $\Sigma^* \setminus$ NeverAccept.

To reach a contradiction, we construct a Turing machine A that decides Accept as follows. Given the encoding $\langle M, w \rangle$ of an arbitrary machine M and an arbitrary string w as input, A writes the encoding $\langle M_w \rangle$ of a new Turing machine M_w that ignores its input, writes w onto the tape, and then passes control to M. Finally, A passes the new encoding $\langle M_w \rangle$ as input to NA^R. The following cartoon tries to illustrate the overall construction.



A reduction from from Accept to NeverAccept, which proves NeverAccept undecidable.

Before going any further, it may be helpful to list the various Turing machines that appear in this construction.

- The hypothetical Turing machine NA that decides NeverAccept.
- The Turing machine NA^R that decides $\Sigma^* \setminus$ NeverAccept, which we constructed by modifying NA.
- The Turing machine A that we are building, which decides Accept using NA^R as a black-box subroutine.
- The Turing machine M, whose encoding is part of the input to A.
- The Turing machine M_w whose encoding A constructs from $\langle M, w\rangle$ and then passes to NA^R as input.

Now let M be an arbitrary Turing machine and w be an arbitrary string, and suppose we run our new Turing machine A on the encoding $\langle M, w\rangle$. To complete the proof, we need to consider two cases: Either M accepts w or M does not accept w.

- First, suppose M accepts w.
  - Then for all strings x, the machine M_w accepts x.
  - So $\text{Accept}(M_w) = \Sigma^*$, by the definition of $\text{Accept}(M_w)$.
  - So $\langle M_w\rangle \notin$ NeverAccept, by definition of NeverAccept.
  - So NA rejects $\langle M_w\rangle$, because NA decides NeverAccept.
  - So NA^R accepts $\langle M_w\rangle$, buy construction of NA^R.
  - We conclude that A accepts $\langle M, w\rangle$, by construction of A.
- On the other hand, suppose M does not accept w, either rejecting or diverging instead.
  - Then for all strings x, the machine M_w does not accept x.
  - So $\text{Accept}(M_w) = \varnothing$, by the definition of $\text{Accept}(M_w)$.
  - So $\langle M_w\rangle \in$ NeverAccept, by definition of NeverAccept.
  - So NA accepts $\langle M_w\rangle$, because NA decides NeverAccept.
  - So NA^R rejects $\langle M_w\rangle$, buy construction of NA^R.
  - We conclude that A rejects $\langle M, w\rangle$, by construction of A.

In short, A decides the language Accept, which is impossible. We conclude that NA does not exist. □

Again, similar arguments imply that the languages NeverReject, NeverHalt, and NeverDiverge are undecidable. In each case, the core of the argument is describing how to transform the incoming machine-and-input encoding $\langle M, w\rangle$ into the encoding of an appropriate new Turing machine $\langle M_w\rangle$.

Now that we know that NeverAccept and its relatives are undecidable, we can use them as the basis of further reduction proofs. Here is a typical example:

Theorem 13. *The language DivergeSame* $:= \left\{\langle M_1\rangle\langle M_2\rangle \;\middle|\; \text{Diverge}(M_1) = \text{Diverge}(M_2)\right\}$ *is undecidable.*

Proof: Suppose for the sake of argument that there is a Turing machine DS that decides DivergeSame. Then we can build a Turing machine ND that decides NeverDiverge as follows. Fix a Turing machine Y that accepts Σ^* (for example, by defining $\delta(\text{start}, a) = (\text{accept}, \cdot, \cdot)$ for all $a \in \Gamma$). Given an arbitrary Turing machine encoding $\langle M\rangle$ as input, ND writes the string $\langle M\rangle\langle Y\rangle$ onto the tape and then passes control to DS. There are two cases to consider:

- If DS accepts $\langle M\rangle\langle Y\rangle$, then $\textsc{Diverge}(M) = \textsc{Diverge}(Y) = \varnothing$, so $\langle M\rangle \in \textsc{NeverDiverge}$.
- If DS rejects $\langle M\rangle\langle Y\rangle$, then $\textsc{Diverge}(M) \neq \textsc{Diverge}(Y) = \varnothing$, so $\langle M\rangle \notin \textsc{NeverDiverge}$.

In short, ND accepts $\langle M\rangle$ if and only if $\langle M\rangle \in \textsc{NeverDiverge}$, which is impossible. We conclude that DS does not exist. □

8.10 Rice's Theorem

In 1953, Henry Rice proved the following extremely powerful theorem, which essentially states that ***every*** interesting question about the language accepted by a Turing machine is undecidable.

Rice's Theorem. *Let $\mathcal{L}$ be any set of languages that satisfies the following conditions:*
- *There is a Turing machine Y such that $\textsc{Accept}(Y) \in \mathcal{L}$.*
- *There is a Turing machine N such that $\textsc{Accept}(N) \notin \mathcal{L}$.*

The language $\textsc{AcceptIn}(\mathcal{L}) := \left\{\langle M\rangle \mid \textsc{Accept}(M) \in \mathcal{L}\right\}$ is undecidable.

Proof: Without loss of generality, suppose $\varnothing \notin \mathcal{L}$. (A symmetric argument establishes the theorem in the opposite case $\varnothing \in \mathcal{L}$.) Fix an arbitrary Turing machine Y such that $\textsc{Accept}(Y) \in \mathcal{L}$.

Suppose to the contrary that there is a Turing machine $A_{\mathcal{L}}$ that decides $\textsc{AcceptIn}(\mathcal{L})$. To derive a contradiction, we describe a Turing machine H that decides the halting language $\textsc{Halt}$, using $A_{\mathcal{L}}$ as a black-box subroutine. Given the encoding $\langle M, w\rangle$ of an arbitrary Turing machine M and an arbitrary string w as input, H writes the encoding $\langle WTF\rangle$ of a new Turing machine WTF that executes the following algorithm:

```
WTF(x):
  run M on input w (and discard the result)
  run Y on input x
```

H then passes the new encoding $\langle WTF\rangle$ to $A_{\mathcal{L}}$.

Now let M be an arbitrary Turing machine and w be an arbitrary string, and suppose we run our new Turing machine H on the encoding $\langle M, w\rangle$. There are two cases to consider.

- Suppose M halts on input w.
  - Then for all strings x, the machine WTF accepts x if and only if Y accepts x.
  - So $\textsc{Accept}(WTF) = \textsc{Accept}(Y)$, by definition of $\textsc{Accept}(\cdot)$.
  - So $\textsc{Accept}(WTF) \in \mathcal{L}$, by definition of Y.
  - So $A_{\mathcal{L}}$ accepts $\langle WTF\rangle$, because $A_{\mathcal{L}}$ decides $\textsc{AcceptIn}(\mathcal{L})$.
  - So H accepts $\langle M, w\rangle$, by definition of H.
- Suppose M does not halt on input w.
  - Then for all strings x, the machine WTF does not halt on input x, and therefore does not accept x.
  - So $\textsc{Accept}(WTF) = \varnothing$, by definition of $\textsc{Accept}(WTF)$.
  - So $\textsc{Accept}(WTF) \notin \mathcal{L}$, by our assumption that $\varnothing \notin \mathcal{L}$.
  - So $A_{\mathcal{L}}$ rejects $\langle WTF\rangle$, because $A_{\mathcal{L}}$ decides $\textsc{AcceptIn}(\mathcal{L})$.
  - So H rejects $\langle M, w\rangle$, by definition of H.

In short, H decides the language HALT, which is impossible. We conclude that $A_{\mathcal{L}}$ does not exist. □

The set $\mathcal{L}$ in the statement of Rice's Theorem is often called a ***property*** of languages, rather than a *set*, to avoid the inevitable confusion about sets of sets. We can also think of $\mathcal{L}$ as a ***decision problem*** about languages, where the languages are represented by Turing machines that accept or decide them. Rice's theorem states that the ***only*** properties of languages that are decidable are the trivial properties "Does this Turing machine accept an acceptable language?" (Answer: Yes, by definition.) and "Does this Turing machine accept Discover?" (Answer: No, because Discover is a credit card, not a language.)

Rice's Theorem makes it incredibly easy to prove that language properties are undecidable; we only need to exhibit one acceptable language that has the property and another acceptable language that does not. In fact, most proofs using Rice's theorem can use at least one of the following Turing machines:

- M_{ACCEPT} accepts every string, by defining $\delta(\mathsf{start}, a) = \mathsf{accept}$ for every tape symbol a.
- M_{REJECT} rejects every string, by defining $\delta(\mathsf{start}, a) = \mathsf{reject}$ for every tape symbol a.
- M_{DIVERGE} diverges on every string, by defining $\delta(\mathsf{start}, a) = (\mathsf{start}, a, +1)$ for every tape symbol a.

Corollary 14. *Each of the following languages is undecidable.*
(a) $\{\langle M\rangle \mid M$ *accepts given an empty initial tape*$\}$
(b) $\{\langle M\rangle \mid M$ *accepts the string* `UIUC`$\}$
(c) $\{\langle M\rangle \mid M$ *accepts exactly three strings*$\}$
(d) $\{\langle M\rangle \mid M$ *accepts all palindromes*$\}$
(e) $\{\langle M\rangle \mid \textit{Accept}(M)$ *is regular*$\}$
(f) $\{\langle M\rangle \mid \textit{Accept}(M)$ *is not regular*$\}$
(g) $\{\langle M\rangle \mid \textit{Accept}(M)$ *is undecidable*$\}$
(h) $\{\langle M\rangle \mid \textit{Accept}(M) = \textit{Accept}(N)\}$, *for some arbitrary fixed Turing machine* N.

Proof: In all cases, undecidability follows from Rice's theorem.

(a) Let $\mathcal{L}$ be the set of all languages that contain the empty string. Then $\text{ACCEPTIN}(\mathcal{L}) = \{\langle M\rangle \mid M$ accepts given an empty initial tape$\}$.

  - Given an empty initial tape, M_{ACCEPT} accepts, so $\text{HALT}(M_{\text{ACCEPT}}) \in \mathcal{L}$.
  - Given an empty initial tape, M_{DIVERGE} does not accept, so $\text{HALT}(M_{\text{DIVERGE}}) \notin \mathcal{L}$.

  Therefore, Rice's Theorem implies that $\text{ACCEPTIN}(\mathcal{L})$ is undecidable.

(b) Let $\mathcal{L}$ be the set of all languages that contain the string `UIUC`.

  - M_{ACCEPT} accepts `UIUC`, so $\text{HALT}(M_{\text{ACCEPT}}) \in \mathcal{L}$.
  - M_{DIVERGE} does not accept `UIUC`, so $\text{HALT}(M_{\text{DIVERGE}}) \notin \mathcal{L}$.

  Therefore, $\text{ACCEPTIN}(\mathcal{L}) = \{\langle M\rangle \mid M$ accepts the string `UIUC`$\}$ is undecidable by Rice's Theorem.

(c) There is a Turing machine that accepts the language $\{$`larry`, `curly`, `moe`$\}$. On the other hand, M_{REJECT} does not accept exactly three strings.

(d) M_{ACCEPT} accepts all palindromes, and M_{REJECT} does not accept all palindromes.

(e) M_{REJECT} accepts the regular language $\varnothing$, and there is a Turing machine $M_{0^n1^n}$ that accepts the non-regular language $\{0^n1^n \mid n \geq 0\}$.

(f) M_{REJECT} accepts the regular language $\varnothing$, and there is a Turing machine $M_{0^n1^n}$ that accepts the non-regular language $\{0^n1^n \mid n \geq 0\}$.[5]

(g) M_{REJECT} accepts the decidable language $\varnothing$, and there is a Turing machine that accepts the undecidable language SELFREJECT.

(h) The Turing machine N accepts $\text{ACCEPT}(N)$ by definition. The Turing machine N^R, obtained by swapping the accept and reject states of N, accepts the language $\text{HALT}(L) \setminus \text{ACCEPT}(N) \neq \text{ACCEPT}(N)$. □

We can also use Rice's theorem as a component in more complex undecidability proofs, where the target language consists of more than just a single Turing machine encoding.

Theorem 15. *The language $L := \{\langle M, w\rangle \mid M \text{ accepts } w^k \text{ for every integer } k \geq 0\}$ is undecidable.*

Proof: Fix an arbitrary string w, and let $\mathcal{L}$ be the set of all languages that contain w^k for all k. Then $\text{ACCEPT}(M_{\text{ACCEPT}}) = \Sigma^* \in \mathcal{L}$ and $\text{ACCEPT}(M_{\text{REJECT}}) = \varnothing \notin \mathcal{L}$. Thus, *even if the string w is fixed in advance*, no Turing machine can decide L. □

Nearly identical reduction arguments imply the following variants of Rice's theorem. (The names of these theorems are not standard.)

Rice's Rejection Theorem. *Let $\mathcal{L}$ be any set of languages that satisfies the following conditions:*

- *There is a Turing machine Y such that $\text{REJECT}(Y) \in \mathcal{L}$*
- *There is a Turing machine N such that $\text{REJECT}(N) \notin \mathcal{L}$.*

The language $\text{REJECTIN}(\mathcal{L}) := \{\langle M\rangle \mid \text{REJECT}(M) \in \mathcal{L}\}$ is undecidable.

Rice's Halting Theorem. *Let $\mathcal{L}$ be any set of languages that satisfies the following conditions:*

- *There is a Turing machine Y such that $\text{HALT}(Y) \in \mathcal{L}$*
- *There is a Turing machine N such that $\text{HALT}(N) \notin \mathcal{L}$.*

The language $\text{HALTIN}(\mathcal{L}) := \{\langle M\rangle \mid \text{HALT}(M) \in \mathcal{L}\}$ is undecidable.

Rice's Divergence Theorem. *Let $\mathcal{L}$ be any set of languages that satisfies the following conditions:*

- *There is a Turing machine Y such that $\text{DIVERGE}(Y) \in \mathcal{L}$*
- *There is a Turing machine N such that $\text{DIVERGE}(N) \notin \mathcal{L}$.*

The language $\text{DIVERGEIN}(\mathcal{L}) := \{\langle M\rangle \mid \text{DIVERGE}(M) \in \mathcal{L}\}$ is undecidable.

Rice's Decision Theorem. *Let $\mathcal{L}$ be any set of languages that satisfies the following conditions:*

- *There is a Turing machine Y such that **decides** an language in $\mathcal{L}$.*
- *There is a Turing machine N such that **decides** an language not in $\mathcal{L}$.*

The language $\text{DECIDEIN}(\mathcal{L}) := \{\langle M\rangle \mid M \textbf{ decides} \text{ a language in } \mathcal{L}\}$ is undecidable.

[5] Yes, parts (e) and (f) have exactly the same proof.

As a final sanity check, always be careful to distinguish the following objects:

- The string ε
- The language $\varnothing$
- The language $\{\varepsilon\}$
- The language property $\varnothing$
- The language property $\{\varnothing\}$
- The language property $\{\{\varepsilon\}\}$
- The Turing machine M_{REJECT} that rejects every string and therefore ***decides*** the language $\varnothing$.
- The Turing machine M_{DIVERGE} that diverges on every string and therefore ***accepts*** the language $\varnothing$.

*8.11 The Rice-McNaughton-Myhill-Shapiro Theorem

The following subtle generalization of Rice's theorem precisely characterizes which properties of acceptable languages are *acceptable*. This result was partially proved by Henry Rice in 1953, in the same paper that proved Rice's Theorem; Robert McNaughton, John Myhill, and Norman Shapiro completed the proof a few years later, each independently from the other two.[6]

The Rice-McNaughton-Myhill-Shapiro Theorem. *Let $\mathcal{L}$ be an arbitrary set of acceptable languages. The language* $\textsc{AcceptIn}(\mathcal{L}) := \{\langle M\rangle \mid \textsc{Accept}(M) \in \mathcal{L}\}$ *is **acceptable** if and only if $\mathcal{L}$ satisfies the following conditions:*

*(a) $\mathcal{L}$ is **monotone**: For any language $L \in \mathcal{L}$, every superset of L is also in $\mathcal{L}$.*

*(b) $\mathcal{L}$ is **compact**: Every language in $\mathcal{L}$ has a finite subset that is also in $\mathcal{L}$.*

*(c) $\mathcal{L}$ is **finitely acceptable**: The language $\big\{\langle L\rangle \mid L \in \mathcal{L} \text{ and } L \text{ is finite}\big\}$ is acceptable.*[7]

I won't give a complete proof of this theorem (in part because it requires techniques I haven't introduced), but the following lemma is arguably the most interesting component:

Lemma 16. *Let $\mathcal{L}$ be a set of acceptable languages. If $\mathcal{L}$ is not monotone, then $\textsc{AcceptIn}(\mathcal{L})$ is unacceptable.*

Proof: Suppose to the contrary that there is a Turing machine $AI_{\mathcal{L}}$ that accepts $\textsc{AcceptIn}(\mathcal{L})$. Using this Turing machine as a black box, we describe a Turing machine SD that accepts the unacceptable language $\textsc{SelfDiverge}$. Fix two Turing machines Y and N such that

$$\begin{aligned} \textsc{Accept}(Y) &\in \mathcal{L},\\ \textsc{Accept}(N) &\notin \mathcal{L},\\ \text{and}\quad \textsc{Accept}(Y) &\subseteq \textsc{Accept}(N). \end{aligned}$$

Let w be the input to SD. After verifying that $w = \langle M\rangle$ for some Turing machine M (and rejecting otherwise), SD writes the encoding $\langle WTF\rangle$ or a new Turing machine WTF that implements the following algorithm:

---

[6]McNaughton never published his proof (although he did announce the result); consequently, this theorem is sometimes called "The Rice-Myhill-Shapiro Theorem". Even more confusingly, Myhill published his proof twice, once in a paper with John Shepherdson and again in a later paper with Jacob Dekker. So maybe it should be called the Rice–Dekker-Myhill–McNaughton–Myhill-Shepherdson–Shapiro Theorem.

[7]Here the encoding $\langle L\rangle$ of a finite language $L \subseteq \Sigma^*$ is exactly the string that you would write down to explicitly describe L. Formally, $\langle L\rangle$ is the unique string over the alphabet $\Sigma \cup \{\texttt{\{}, \texttt{,}, \texttt{\}}, \texttt{ε}\}$ that contains the strings in L in lexicographic order, separated by commas `,` and surrounded by braces `{}`, with `ε` representing the empty string. For example, $\big\langle\{\varepsilon, \texttt{0}, \texttt{01}, \texttt{0110}, \texttt{01101001}\}\big\rangle = \texttt{\{ε,0,01,0110,01101001\}}$.

```
WTF(x):
  write x to second tape
  write ⟨M⟩ to third tape
  in parallel:
      run Y on the first tape
      run N on the second tape
      run M on the third tape
  if Y accepts x
      accept
  if N accepts x and M halts on ⟨M⟩
      accept
```

Finally, SD passes the new encoding $\langle WTF\rangle$ to $AI_{\mathcal{L}}$. There are two cases to consider:

- If M halts on $\langle M\rangle$, then $\textsc{Accept}(WTF) = \textsc{Accept}(N) \notin \mathcal{L}$, and therefore $AI_{\mathcal{L}}$ does not accept $\langle WTF\rangle$.
- If M does not halt on $\langle M\rangle$, then $\textsc{Accept}(WTF) = \textsc{Accept}(Y) \in \mathcal{L}$, and therefore $AI_{\mathcal{L}}$ accepts $\langle WTF\rangle$.

In short, SD accepts SelfDiverge, which is impossible. We conclude that SD does not exist. □

Corollary 17. *Each of the following languages is unacceptable.*
(a) $\{\langle M\rangle \mid \textsc{Accept}(M) \text{ is finite}\}$
(b) $\{\langle M\rangle \mid \textsc{Accept}(M) \text{ is infinite}\}$
(c) $\{\langle M\rangle \mid \textsc{Accept}(M) \text{ is regular}\}$
(d) $\{\langle M\rangle \mid \textsc{Accept}(M) \text{ is not regular}\}$
(e) $\{\langle M\rangle \mid \textsc{Accept}(M) \text{ is decidable}\}$
(f) $\{\langle M\rangle \mid \textsc{Accept}(M) \text{ is undecidable}\}$
(g) $\{\langle M\rangle \mid M \text{ accepts at least one string in } \textsc{SelfDiverge}\}$
(h) $\{\langle M\rangle \mid \textsc{Accept}(M) = \textsc{Accept}(N)\}$*, for some arbitrary fixed Turing machine* N.

Proof: (a) The set of finite languages is not monotone: $\varnothing$ is finite; Σ^* is not finite; both $\varnothing$ and Σ^* are acceptable (in fact decidable); and $\varnothing \subset \Sigma^*$.

(b) The set of infinite acceptable languages is not compact: No finite subset of the infinite acceptable language Σ^* is infinite!

(c) The set of regular languages is not monotone: Consider the languages $\varnothing$ and $\{0^n1^n \mid n \geq 0\}$.

(d) The set of non-regular acceptable languages is not monotone: Consider the languages $\{0^n1^n \mid n \geq 0\}$ and Σ^*.

(e) The set of decidable languages is not monotone: Consider the languages $\varnothing$ and SelfReject.

(f) The set of undecidable acceptable languages is not monotone: Consider the languages SelfReject and Σ^*.

(g) The set $\mathcal{L} = \{L \mid L \cap \textsc{SelfDiverge} \neq \varnothing\}$ is not finitely acceptable. For any string w, deciding whether $\{w\} \in \mathcal{L}$ is equivalent to deciding whether $w \in \textsc{SelfDiverge}$, which is impossible.

(h) If $\textsc{Accept}(N) \neq \Sigma^*$, then the set $\{\textsc{Accept}(N)\}$ is not monotone. On the other hand, if $\textsc{Accept}(N) = \Sigma^*$, then the set $\{\textsc{Accept}(N)\}$ is not compact: No finite subset of Σ^* is equal to Σ^*! □

8.12 Turing Machine Behavior: It's Complicated

Rice's theorems imply that every interesting question about the language that a Turing machine accepts—or more generally, the function that a program computes—is undecidable. A more subtle question is whether we can recognize Turing machines that exhibit certain *internal behavior*. Some behaviors we can recognize; others we can't.

Theorem 18. *The language* NEVERLEFT $:= \{\langle M, w\rangle \mid$ *Given w as input, M never moves left*$\}$ *is decidable.*

Proof: Given the encoding $\langle M, w\rangle$, we simulate M with input w using our universal Turing machine U, but with the following termination conditions. If M ever moves its head to the left, then we reject. If M halts without moving its head to the left, then we accept. Finally, if M reads more than $|Q|$ blanks, where Q is the state set of M, then we accept. If the first two cases do not apply, M only moves to the right; moreover, after reading the entire input string, M only reads blanks. Thus, after reading $|Q|$ blanks, it must repeat some state, and therefore loop forever without moving to the left. The three cases are exhaustive. □

Theorem 19. *The language* LEFTTHREE $:= \{\langle M, w\rangle \mid$ *Given w as input, M eventually moves left three times in a row*$\}$ *is undecidable.*

Proof: Given $\langle M\rangle$, we build a new Turing machine M' that accepts the same language as M and moves left three times in a row if and only if it accepts, as follows. For each non-accepting state p of M, the new machine M' has three states p_1, p_2, p_3, with the following transitions:

$$\begin{aligned} \delta'(p_1, a) &= (q_2, b, \Delta), && \text{where } (q, b, \Delta) = \delta(p, a) \text{ and } q \neq \text{accept} \\ \delta'(p_2, a) &= (p_3, a, +1) \\ \delta'(p_3, a) &= (p_1, a, -1) \end{aligned}$$

In other words, after each non-accepting transition, M' moves once to the right and then once to the left. For each transition to accept, M' has a sequence of seven transitions: three steps to the right, then three steps to the left, and then finally accept', all without modifying the tape. (The three steps to the right ensure that M' does not fall off the left end of the tape.)

Finally, M' moves left three times in a row if and only if M accepts w. Thus, if we could decide LEFTTHREE, we could also decide ACCEPT, which is impossible. □

There is no hard and fast rule like Rice's theorem to distinguish decidable behaviors from undecidable behaviors, but I can offer two rules of thumb.

- If it is possible to simulate an arbitrary Turing machine while avoiding the target behavior, then the behavior is not decidable. For example: there is no algorithm to determine whether a given Turing machine reenters its start state, or revisits the left end of the tape, or writes a blank.
- If a Turing machine with the target behavior is limited to a finite number of configurations, or is guaranteed to force an infinite loop after a finite number of transitions, then the behavior is likely to be decidable. For example, there *are* algorithms to determine whether a given Turing machine ever leaves its start state, or reads its entire input string, or writes a non-blank symbol over a blank.

Exercises

1. Let M be an arbitrary Turing machine.
   (a) Describe a Turing machine M^R such that
   $$\textsc{Accept}(M^R) = \textsc{Reject}(M) \quad \text{and} \quad \textsc{Reject}(M^R) = \textsc{Accept}(M).$$
   (b) Describe a Turing machine M^A such that
   $$\textsc{Accept}(M^A) = \textsc{Accept}(M) \quad \text{and} \quad \textsc{Reject}(M^A) = \varnothing.$$
   (c) Describe a Turing machine M^H such that
   $$\textsc{Accept}(M^H) = \textsc{Halt}(M) \quad \text{and} \quad \textsc{Reject}(M^H) = \varnothing.$$

2. (a) Prove that Accept is undecidable.
   (b) Prove that Reject is undecidable.
   (c) Prove that Diverge is undecidable.

3. (a) Prove that NeverReject is undecidable.
   (b) Prove that NeverHalt is undecidable.
   (c) Prove that NeverDiverge is undecidable.

4. Prove that each of the following languages is undecidable.
   (a) $\textsc{AlwaysAccept} := \{\langle M\rangle \mid \textsc{Accept}(M) = \Sigma^*\}$
   (b) $\textsc{AlwaysReject} := \{\langle M\rangle \mid \textsc{Reject}(M) = \Sigma^*\}$
   (c) $\textsc{AlwaysHalt} := \{\langle M\rangle \mid \textsc{Halt}(M) = \Sigma^*\}$
   (d) $\textsc{AlwaysDiverge} := \{\langle M\rangle \mid \textsc{Diverge}(M) = \Sigma^*\}$

5. Let $\mathcal{L}$ be a non-empty proper subset of the set of acceptable languages. Prove that the following languages are undecidable:
   (a) $\textsc{RejectIn}(\mathcal{L}) := \big\{\langle M\rangle \bigm| \textsc{Reject}(M) \in \mathcal{L}\big\}$
   (b) $\textsc{HaltIn}(\mathcal{L}) := \big\{\langle M\rangle \bigm| \textsc{Halt}(M) \in \mathcal{L}\big\}$
   (c) $\textsc{DivergeIn}(\mathcal{L}) := \big\{\langle M\rangle \bigm| \textsc{Diverge}(M) \in \mathcal{L}\big\}$

6. For each of the following decision problems, either *sketch* an algorithm or prove that the problem is undecidable. Recall that w^R denotes the reversal of string w. For each problem, the input is the encoding $\langle M\rangle$ of a Turing machine M.
   (a) Does M accept $\langle M\rangle^R$?
   (b) Does M reject any palindrome?
   (c) Does M accept all palindromes?

(d) Does M diverge only on palindromes?

(e) Is there an input string that forces M to move left?

(f) Is there an input string that forces M to move left three times in a row?

(g) Does M accept the encoding of any Turing machine N such that $\text{Accept}(N) = \text{SelfDiverge}$?

7. For each of the following decision problems, either *sketch* an algorithm or prove that the problem is undecidable. Recall that w^R denotes the reversal of string w. For each problem, the input is an encoding $\langle M, w\rangle$ of a Turing machine M and its input string w.

(a) Does M accept the string ww^R?

(b) Does M accept either w or w^R?

(c) Does M either accept w or reject w^R?

(d) Does M accept the string w^k for some integer k?

(e) Does M accept w in at most $2^{|w|}$ steps?

(f) If we run M on input w, does M ever change a symbol on its tape?

(g) If we run M on input w, does M ever move to the right?

(h) If we run M on input w, does M ever move to the right twice in a row?

(i) If we run M on input w, does M move its head to the right more than $2^{|w|}$ times (not necessarily consecutively)?

(j) If we run M with input w, does M ever change a $\square$ on the tape to any other symbol?

(k) If we run M with input w, does M ever change a $\square$ on the tape to 1?

(l) If we run M with input w, does M ever write a $\square$?

(m) If we run M with input w, does M ever leave its start state?

(n) If we run M with input w, does M ever reenter its start state?

(o) If we run M with input w, does M ever reenter a state that it previously left? That is, are there states $p \neq q$ such that M moves from state p to state q and then later moves back to state p?

8. Let M be a Turing machine, let w be an arbitrary input string, and let s and t be positive integers integer. We say that M accepts w ***in space s*** if M accepts w after accessing at most the first s cells on the tape, and M accepts w ***in time t*** if M accepts w after at most t transitions.

(a) Prove that the following languages are decidable:

i. $\left\{\langle M, w\rangle \;\middle|\; M \text{ accepts } w \text{ in time } |w|^2\right\}$

ii. $\left\{\langle M, w\rangle \;\middle|\; M \text{ accepts } w \text{ in space } |w|^2\right\}$

(b) Prove that the following languages are undecidable:

i. $\left\{\langle M\rangle \;\middle|\; M \text{ accepts at least one string } w \text{ in time } |w|^2\right\}$

ii. $\left\{\langle M\rangle \;\middle|\; M \text{ accepts at least one string } w \text{ in space } |w|^2\right\}$

9. Let L_0 be an arbitrary language. For any integer $i > 0$, define the language

$$L_i := \left\{ \langle M \rangle \mid M \text{ decides } L_{i-1} \right\}.$$

For which integers $i > 0$ is L_i decidable? Obviously the answer depends on the initial language L_0; give a complete characterization of all possible cases. Prove your answer is correct. *[Hint: This question is a lot easier than it looks!]*

10. Argue that each of the following decision problems about programs in your favorite programming language are undecidable.

    (a) Does this program correctly compute Fibonacci numbers?

    (b) Can this program fall into an infinite loop?

    (c) Will the value of this variable ever change?

    (d) Will this program every attempt to deference a null pointer?

    (e) Does this program free every block of memory that it dynamically allocates?

    (f) Is any statement in this program unreachable?

    (g) Do these two programs compute the same function?

$\star$11. Call a Turing machine ***conservative*** if it never writes over its input string. More formally, a Turing machine is conservative if for every transition $\delta(p,a) = (q,b,\Delta)$ where $a \in \Sigma$, we have $b = a$; and for every transition $\delta(p,a) = (q,b,\Delta)$ where $a \notin \Sigma$, we have $b \neq \Sigma$.

   (a) Prove that if M is a conservative Turing machine, then $\text{A}\small{\text{CCEPT}}(M)$ is a regular language.

   (b) Prove that the language $\{\langle M \rangle \mid M \text{ is conservative and } M \text{ accepts } \varepsilon\}$ is undecidable.

   Together, these two results imply that every conservative Turing machine accepts the same language as some DFA, but it is impossible to determine *which* DFA.

$\bigstar$12. (a) Prove that it is undecidable whether a given C++ program is syntactically correct. *[Hint: Use templates!]*

   (b) Prove that it is undecidable whether a given ANSI C program is syntactically correct. *[Hint: Use the preprocessor!]*

   (c) Prove that it is undecidable whether a given Perl program is syntactically correct. *[Hint: Does that slash character / delimit a regular expression or represent division?]*

Caveat lector: This is the zeroth (draft) edition of this lecture note. In particular, some topics still need to be written. Please send bug reports and suggestions to jeffe@illinois.edu.

If first you don't succeed, then try and try again.
And if you don't succeed again, just try and try and try.

— Marc Blitzstein, "Useless Song", *The Three Penny Opera* (1954)
Adaptation of Bertold Brecht, "Das Lied von der Unzulänglichkeit menschlichen Strebens" *Die Dreigroschenoper* (1928)

Children need encouragement.
If a kid gets an answer right, tell him it was a lucky guess.
That way he develops a good, lucky feeling.

— Jack Handey, "Deep Thoughts", *Saturday Night Live* (March 21, 1992)

9 Nondeterministic Turing Machines

9.1 Definitions

In his seminal 1936 paper, Turing also defined an extension of his "automatic machines" that he called ***choice machines***, which are now more commonly known as ***nondeterministic Turing machines***. The execution of a nondeterministic Turing machine is *not determined* entirely by its input and its transition function; rather, at each step of its execution, the machine can *choose* from a set of possible transitions. The distinction between deterministic and nondeterministic Turing machines exactly parallels the distinction between deterministic and nondeterministic finite-state automata.

Formally, a nondeterministic Turing machine has all the components of a standard deterministic Turing machine—a finite tape alphabet Γ that contains the input alphabet Σ and a blank symbol $\square$; a finite set Q of internal states with special start, accept, and reject states; and a transition function δ. However, the transition function now has the signature

$$\delta : Q \times \Gamma \to 2^{Q \times \Gamma \times \{-1,+1\}}.$$

That is, for each state p and tape symbol a, the output $\delta(p,a)$ of the transition function is a *set* of triples of the form $(q,b,\Delta) \in Q \times \Gamma \times \{-1,+1\}$. Whenever the machine finds itself in state p reading symbol a, the machine *chooses* an arbitrary triple $(q,b,\Delta) \in \delta(p,a)$, and then changes its state to q, writes b to the tape, and moves the head by Δ. If the set $\delta(p,a)$ is empty, the machine moves to the reject state and halts.

The set of all possible transition sequences of a nondeterministic Turing machine N on a given input string w define a rooted tree, called a ***computation tree***. The initial configuration $(\text{start}, w, 0)$ is the root of the computation tree, and the children of any configuration (q,x,i) are the configurations that can be reached from (q,x,i) in one transition. In particular, any configuration whose state is accept or reject is a leaf. For deterministic Turing machines, this computation tree is just a single path, since there is at most one valid transition from every configuration.

9.2 Acceptance and Rejection

Unlike deterministic Turing machines, there is a fundamental asymmetry between the acceptance and rejection criteria for nondeterministic Turing machines. Let N be any nondeterministic Turing machine, and let w be any string.

- ***N accepts w*** if and only if there is *at least one* sequence of valid transitions from the initial configuration $(\text{start}, w, 0)$ that leads to the accept state. Equivalently, N accepts w if the computation tree contains at least one accept leaf.
- ***N rejects w*** if and only if *every* sequence of valid transitions from the initial configuration $(\text{start}, w, 0)$ leads to the reject state. Equivalently, N rejects w if every path through the computation tree ends with a reject leaf.

In particular, N can accept w even when there are choices that allow the machine to run forever, but rejection requires N to halt after only a finite number of transitions, no matter what choices it makes along the way. Just as for deterministic Turing machines, it is possible that N neither accepts nor rejects w.

Acceptance and rejection of *languages* are defined exactly as they are for deterministic machines. A non-deterministic Turing machine N ***accepts*** a language $L \subseteq \Sigma^*$ if M accepts all strings in L and nothing else; N ***rejects*** L if M rejects every string in L and nothing else; and finally, N ***decides*** L if M accepts L and rejects $\Sigma^* \setminus L$.

9.3 Time and Space Complexity

★★★

- Define "time" and "space".
- TIME$(f(n))$ is the class of languages that can be decided by a deterministic multi-tape Turing machine in $O(f(n))$ time.
- NTIME$(f(n))$ is the class of languages that can be decided by a nondeterministic multi-tape Turing machine in $O(f(n))$ time.
- SPACE$(f(n))$ is the class of languages that can be decided by deterministic multi-tape Turing machine in $O(f(n))$ space.
- NSPACE$(f(n))$ is the class of languages that can be decided by a nondeterministic multi-tape Turing machine in $O(f(n))$ space.
- Why multi-tape TMs? Because t steps on any k-tape Turing machine can be simulated in $O(t \log t)$ steps on a two-tape machine [Hennie and Stearns 1966, essentially using lazy counters and amortization], and in $O(t^2)$ steps on a single-tape machine [Hartmanis and Stearns 1965; realign multiple tracks at every simulation step]. Moreover, the latter quadratic bound is tight [Hennie 1965 (palindromes, via communication complexity)].

9.4 Deterministic Simulation

Theorem 1. *For any nondeterministic Turing machine N, there is a deterministic Turing machine M that accepts exactly the same strings and N and rejects exactly the same strings as N. Moreover, if every computation path of N on input x halts after at most t steps, then M halts on input x after at most $O(t^2 r^{2t})$ steps, where r is the maximum size of any transition set in N.*

Proof: I'll describe a deterministic machine M that performs a breadth-first search of the computation tree of N. (The depth-first search performed by a standard recursive backtracking algorithm won't work here. If N's computation tree contains an infinite path, a depth-first search would get stuck in that path without exploring the rest of the tree.)

At the beginning of each simulation round, M's tape contains a string of the form

$$\square\square\cdots\square\bullet\bullet y_1q_1z_1\bullet y_2q_2z_2\bullet\cdots\bullet y_kq_kz_k\bullet\bullet$$

where each substring $y_iq_iz_i$ encodes a configuration $(q_i, y_iz_i, |y_i|)$ of some computation path of N, and $\bullet$ is a new symbol not in the tape alphabet of N. The machine M interprets this sequence of encoded configurations as a queue, with new configurations inserted on the right and old configurations removed from the left. The double-separators $\bullet\bullet$ uniquely identify the start and end of this queue; outside this queue, the tape is entirely blank.

Specifically, in each round, first M appends the encodings of all configurations than N can reach in one transition from the first encoded configuration $(q_1, y_1z_1, |y_1|)$; then M erases the first encoded configuration.

$$\cdots\square\square\underline{\bullet\bullet y_1q_1z_1}\bullet y_2q_2z_2\bullet\cdots\bullet y_rq_rz_r\bullet\underline{\bullet\square\square\cdots}$$

$$\Downarrow \qquad\qquad\qquad\qquad \Downarrow$$

$$\cdots\square\square\underline{\square\square\cdots\square\bullet}\bullet y_2q_2z_2\bullet\cdots\bullet y_kq_kz_k\bullet\underline{\tilde{y}_1\tilde{q}_1\tilde{z}_1\bullet\tilde{y}_2\tilde{q}_2\tilde{z}_2\bullet\cdots\bullet\tilde{y}_r\tilde{q}_r\tilde{z}_r\bullet\bullet\square\square\cdots}$$

Suppose each transition set $\delta_N(q,a)$ has size at most r. Then after simulating t steps of N, the tape string of M encoding $O(r^t)$ different configurations of N and therefore has length $L = O(tr^t)$ (not counting the initial blanks). If M begins each simulation phase by moving the initial configuration from the beginning to the end of the tape string, which takes $O(t^2r^t)$ time, the time for the rest of the the simulation phase is negligible. Altogether, simulating all r^t possibilities for the the tth step of N requires $O(t^2r^{2t})$ time. We conclude that M can simulate the first t steps of every computation path of N in $\boldsymbol{O(t^2r^{2t})}$ ***time***, as claimed. $\square$

The running time of this simulation is dominated by the time spent reading from one end of the tape string and writing to the other. It is fairly easy to reduce the running time to $O(tr^t)$ by using either two tapes (a "read tape" containing N-configurations at time t and a "write tape" containing N-configurations at time $t+1$) or two independent heads on the same tape (one at each end of the queue).

9.5 Nondeterminism as Advice

Any nondeterministic Turing machine N can also be simulated by a *deterministic* machine M with *two* inputs: the user input string $w \in \Sigma^*$, and a so-called ***advice*** string $x \in \Omega^*$, where Ω is another finite alphabet. Only the first input string w is actually given by the user. At least for now, we assume that the advice string x is given on a separate read-only tape.

The deterministic machine M simulates N step-by-step, but whenever N has a choice of how to transition, M reads a new symbol from the advice string, and that symbol determines the choice. In fact, without loss of generality, we can assume that M reads a new symbol from the advice string and moves the advice-tape's head to the right on *every* transition. Thus, M's transition function has the form $\delta_M : Q\times\Gamma\times\Omega \to Q\times\Gamma\times\{-1,+1\}$, and we require that

$$\delta_N(q,a) = \{\delta_M(q,a,\omega) \mid \omega\in\Omega\}$$

For example, if N has a binary choice

$$\delta_N(\mathsf{branch}, ?) = \big\{(\mathsf{left}, \mathsf{L}, -1), (\mathsf{right}, \mathsf{R}, +1)\big\},$$

then M might determine this choice by defining

$$\delta_M(\texttt{branch}, \texttt{?}, \texttt{0}) = (\texttt{left}, \texttt{L}, -1) \qquad \text{and} \qquad \delta_M(\texttt{branch}, \texttt{?}, \texttt{1}) = (\texttt{right}, \texttt{R}, +1)$$

More generally, if every set $\delta_N(p,a)$ has size r, then we let $\Omega = \{1,2,\ldots,r\}$ and define $\delta_M(q,a,i)$ to be the ith element of $\delta_N(q,a)$ in some canonical order.

Now observe that N accepts a string w if and only if M accepts the pair (w,x) for *some* string $x \in \Omega^*$, and N rejects w if and only if M rejects the pair (w,x) for *all* strings $x \in \Omega^*$.

The "advice" formulation of nondeterminism allows a different strategy for simulation by a standard deterministic Turing machine, which is often called ***dovetailing***. Consider all possible advice strings x, in increasing order of length; listing these advice strings is equivalent to repeatedly incrementing a base-r counter. For each advice string x, simulate M on input (w,x) for exactly $|x|$ transitions.

```
DOVETAIL_M(w):
  for t ← 1 to ∞
      done ← TRUE
      for all strings x ∈ Ω^t
          if M accepts (w, x) in at most t steps
              accept
          if M(w, x) does not halt in at most t steps
              done ← FALSE
      if done
          reject
```

The most straightforward Turing-machine implementation of this algorithm requires three tapes: A read-only input tape containing w, an advice tape containing x (which is also used as a timer for the simulation), and the work tape. This simulation requires $O(tr^t)$ time to simulate all possibilities for t steps of the original non-deterministic machine N.

If we insist on using a standard Turing machine with a single tape and a single head, the simulation becomes slightly more complex, but (unlike our earlier queue-based strategy) not significantly slower. This standard machine S maintains a string of the form $\bullet w \bullet x \bullet z$, where z is the current work-tape string of M (or equivalently, of N), with marks (on a second track) indicating the current positions of the heads on M's work tape and M's advice tape. Simulating a single transition of M now requires $O(|x|)$ steps, because S needs to shuttle its single head between these two marks. Thus, S requires $\boldsymbol{O(t^2 r^t)}$ ***time*** to simulate all possibilities for t steps of the original non-deterministic machine N. This is significantly faster than the queue-based simulation, because we don't record (and therefore don't have to repeatedly scan over) intermediate configurations; recomputing everything from scratch is actually cheaper!

9.6 The Cook-Levin Theorem

★★★ Define SAT and CIRCUITSAT. Non-determinism is fundamentally different from other Turing machine extensions, in that it seems to provide an exponential speedup for some problems, just like NFAs can use exponentially fewer states than DFAs for the same language.

The Cook-Levin Theorem. *If SAT $\in$ P, then P=NP.*

Proof: Let $L \subseteq \Sigma^*$ be an arbitrary language in NP, over some fixed alphabet Σ. There must be an integer k and Turing machine M that satisfies the following conditions:

- For all strings $w \in L$, there is at least one string $x \in \Sigma^*$ such that M accepts the string $w\square x$.
- For all strings $w \notin L$ and $x \in \Sigma^*$, M rejects the string $w\square x$.
- For all strings $w, x \in \Sigma^*$, M halts on input $w\square x$ after at most $\max\{1, |w|^k\}$ steps.

Now suppose we are given a string $w \in \Sigma^*$. Let $n = |w|$ and let $N = \max\{1, |w|^k\}$. We construct a boolean formula Φ_w that is satisfiable if and only if $w \in L$, by following the execution of M on input $w\square x$ for some unknown advice string x. Without loss of generality, we can assume that $|x| = N - n - 1$ (since we can extend any shorter string x with blanks.) Our formula Φ_w uses the following boolean variables for all symbols $a \in \Gamma$, all states $q \in Q$, and all integers $0 \leq t \leq N$ and $0 \leq i \leq N + 1$.

- $Q_{t,i,q}$ — M is in state q with its head at position i after t transitions.
- $T_{t,i,a}$ — The kth cell of M's work tape contains a after t transitions.

The formula Φ_w is the conjunction of the following constraints:

- **Boundaries:** To simplify later constraints, we include artificial boundary variables just past both ends of the tape:

$$Q_{t,i,q} = Q_{t,N+1,q} = \text{FALSE} \qquad \text{for all } 0 \leq t \leq N \text{ and } q \in Q$$
$$T_{t,0,a} = T_{t,N+1,a} = \text{FALSE} \qquad \text{for all } 0 \leq t \leq N \text{ and } a \in \Gamma$$

- **Initialization:** We have the following values for variables with $t = 0$:

$$\begin{aligned}
Q_{0,1,\text{start}} &= \text{TRUE} & & \\
Q_{0,1,q} &= \text{FALSE} & & \text{for all } q \neq \text{start} \\
H_{0,i,q} &= \text{FALSE} & & \text{for all } i \neq 1 \text{ and } q \in Q \\
T_{0,i,w_i} &= \text{TRUE} & & \text{for all } 1 \leq i \leq n \\
T_{0,i,a} &= \text{FALSE} & & \text{for all } 1 \leq i \leq n \text{ and } a \neq w_i \\
T_{0,n+1,\square} &= \text{TRUE} & & \\
T_{0,i,a} &= \text{FALSE} & & \text{for all } a \neq \square
\end{aligned}$$

- **Uniqueness:** The variables $T_{0,i,a}$ with $n + 2 \leq i \leq N$ represent the unknown advice string x; these are the "inputs" to Φ_w. We need some additional constraints ensure that for each i, *exactly one* of these variables is TRUE:

$$\left(\bigvee_{a \in \Gamma} T_{0,j,a}\right) \wedge \bigwedge_{a \neq b} \left(\overline{T_{0,j,a}} \vee \overline{T_{0,j,b}}\right)$$

- **Transitions:** For all $1 \leq t \leq N$ and $1 \leq i \leq N$, the following constraints simulate the transition from time $t - 1$ to time t.

$$Q_{t,i,q} = \bigvee_{\delta(p,a)=(q,\cdot,+1)} \left(Q_{t-1,i-1,p} \wedge T_{t,i-1,a}\right) \vee \bigvee_{\delta(p,a)=(q,\cdot,-1)} \left(Q_{t-i,i+1,p} \wedge T_{t,i+1,a}\right)$$

$$T_{t,i,b} = \bigvee_{\delta(p,a)=(\cdot,b,\cdot)} \left(Q_{t-1,i,p} \wedge T_{t-1,i,a}\right) \vee \left(\bigwedge_{q \in Q} \overline{Q_{t-1,i,q}} \wedge T_{t-1,i,b}\right)$$

- **Output:** We have one final constraint that indicates acceptance.

$$z = \bigvee_{t=0}^{N} \bigvee_{i=1}^{N} Q_{t,i,\text{accept}}$$

A straightforward induction argument implies that *without the acceptance constraint*, any assignment of values to the unknown variables $T_{0,i,a}$ that satisfies the uniqueness constraints determines *unique* values for the other variables in Φ_w, which consistently describe the execution of M. Thus, Φ_w is satisfiable if and only if for some input values $T_{0,i,a}$, the resulting , including acceptance. In other words, Φ_w is satisfiable if and only if there is a string $x \in \Gamma^*$ such that M accepts the input $w\square x$. We conclude that Φ_w is satisfiable if and only if $w \in L$.

For any any string w of length n, the formula Φ_w has $O(N^2)$ variables and $O(N^2)$ constraints (where the hidden constants depend on the machine M). Every constraint except acceptance has constant length, so altogether Φ_w has length $O(N^2)$. Moreover, we can construct Φ_w in $O(N^2) = O(n^{2k})$ time.

In conclusion, if we could decide SAT for formulas of size n in (say) $O(n^c)$ time, then we could decide membership in L in $O(n^{2kc})$ time, which implies that $L \in \text{P}$. □

Exercises

1. Prove that the following problem is NP-hard, *without* using the Cook-Levin Theorem. Given a string $\langle M, w\rangle$ that encodes a non-deterministic Turing machine M and a string w, does M accept w in at most $|w|$ transitions?

★★★ More excerises!