Lecture Notes on Foundations of Computer Science

DANIEL J. DOUGHERTY

Worcester Polytechnic Institute

August 25, 2021 – 09 : 46

Contents

Intro

Introduction

The notes are designed for teaching various courses in the foundations of computer science. We hot the canonical set of topics that appears reliably in every such course: finite automata, context-free grammars, and undecidability.

Depending on your background you might be put off by the (necessarily) mathematical presentation. But being able to read mathematics and write mathematically is a crucial skill for a computer scientist. One of our goals here is to help you acquire those skills, so we are extra supportive and discursive *around* the math. If you come out of this course with sharpened ability to communicate mathematically, this will probably be the most useful outcome of the course.

How to Succeed

There is only one rule: **Do the problems at the end of the sections.**

The way to learn mathematical material is to work through problems. Period. The text exists only to provide guidance in working the problems. Class lectures—or Youtube videos—exist only to provide guidance in working the problems. Really.

The biggest mistake you can make is to read the text, nod your head, say, "ok that makes sense", and turn the page to the next topic. It is super-easy to fool yourself into believing that you understand something just because you read it. But ask yourself: can you learn to play soccer by reading about it? Can you learn to play the piano by reading about? No. You have to actively engage something if you want to be able to do it.

Solutions to many of the problems will be made available. But in the spirit of the previous paragraph, I advise you in the strongest possible terms not to look at the solution to a problem before you have arrived at your own answer. It is very tempting, I know. And it seems like it will save you time, I know. But please trust me when I tell you that grappling with problems and looking for solutions is where your learning happens.

Acknowledgments Some problems and examples have been borrowed, as indicated, from the textbooks by John Hopcroft, Rajeev Motwani and Jeffrey Ullman [HMU06], Dexter Kozen [Koz97], Michael Sipser [Sip96], and Thomas Sudkamp [Sud97].

Part I

Mathematics Background

Chapter 1

Functions and Relations

1.1 Notation for Common Sets

The following standard sets arise everywhere:

- $\mathbb{N}$, the natural numbers: $\{0, 1, 2, \ldots\}$.
- $\mathbb{Z}$, the integers: $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$.
- $\mathbb{Q}$, the rational numbers: the set of numbers that can be written as quotients p/q of integers (with $q \neq 0$).
- $\mathbb{R}$, the real numbers: the set of numbers that can be written as—possibly infinite—decimals.

1.2 Functions and Composition

You are probably used to thinking of a function as something like, "a formula for taking inputs and defining outputs." A key point for this class is to get rid of that idea! One important reason is that function need not be defined by formulas; we'll explore that in depth in the Computability part of this text. But even before we get to that idea, we can see already that a function can't be the same thing as a formula.

Consider these two formulas for defining functions (over the real numbers)

$$f(x, y) = \sin(x + y)$$

$$g(x, y) = \sin(x)\cos(y) + \cos(x)\sin(y)$$

Surely those are two different formulas. But as you learned in calculas class, f and g are the same function. So functions can't be the same as formulas.

So what does that mean, exactly, "the same function"? It means that for every input f and g have the same output. Thus it is the collection of (input, output) pairs that truly says what a function *is*. Any formula or algorithm that you might use to *calculate* these (input, output) pairs is a secondary notion.

Here, then is our definition of what a function is.

1.1 Definition (Functions)**.** A *function* $f : A \to B$ is a subset $A \times B$, that is a set of ordered pairs, that satisfies the following property: for each $a \in A$ there is exactly one $b \in B$ with $(a, b) \in f$.

Common usage is to write $f(a) = b$ instead of $(a, b) \in f$. We say that f has *domain* A and *codomain* B. The *image* of f is $\{b \in B \mid \exists a \in A,\ f(a) = b\}$, the elements of the codomain that actually arise as images. [1]

1.2.1 Partial Functions

Sometimes (especially when studying computability) it is convenient to relax the definition of function to allow "partial functions", which are like functions except that they need not yield an answer for each input.

1.2 Definition (Partial Functions)**.** A *partial function* $f : A \to B$ is a subset $A \times B$, that is a set of ordered pairs, that satisfies the following property: for each $a \in A$ there is at most one $b \in B$ with $(a, b) \in f$.

It is rare in ordinary mathematics to work with partial functions. It's not unheard of: consider the partial function defined by the expression $1/x$, which is undefined for $x = 0$ But in computability theory, partial functions are the norm.

Caution! When we refer to a certain f as being a "partial function" we do *not* mean that its domain fails to be all of Σ^*: we are just leaving open that possibility. That is to say, an ordinary function is *also* a "partial function." If f is a partial function that happens to be an ordinary function on A, that is, if the domain of f is known to be all of A, then we sometimes emphasize this by referring to f as a *total function.*

1.2.2 Function Composition

[1] Unfortunately some authors use the word "range" for what we have called the "codomain." You will have to be careful if you read other books in conjunction with this one.

1.3 Definition (Function Composition)**.** Let f be a function from A to B, and let g be a binary relation from B to C. The *composition* $g \circ f$ is the function from A to C given by: $(g \circ f)(x) = g(f(x))$.

Note that $(g \circ f)$ means "f first, then g"! Composition of functions is not always defined, of course, the domain and codomains have to match up properly.

The identity function on set A is denoted id_A.

- Function composition is associative: $((h \circ g) \circ f) = (h \circ (g \circ f))$.
- The identity function is an identity element for composition: $(f \circ id_A) = f$ and $(id_A \circ f) = f$ when $f : A \to A$.

Composition is *not* commutative.

1.4 Check Your Reading. *Give an example to show that function composition is not commutative.*

An easy way out is to note that if f and g are functions with different domains and codomains then $f \circ g \neq g \circ f$, for a dumb reason (what is it?)

So do something more interesting: find a set A and functions $f : A \to A$ and $g : A \to A$ where $f \circ g \neq g \circ f$. (This is easy. Just try some random functions and they will probably work!)

1.3 Subsets, Characteristic Functions, and Bitstrings

There is a very important connection between the idea of *subset* of a given set and the idea of *function* defined over that set. We will use this connection constantly throughout the course.

1.5 Notation. Let U be a set. Then $Pow(U)$ is the set of all subsets of U.

Think of it intuitively as "the universe."

Now imagine a set $S \subseteq U$.

Based on S we can define a function $\mathrm{ch}_S : U \to \{0, 1\}$ by

1.6 Definition.

$$\mathrm{ch}_S(x) = \begin{cases} 0 & \text{if } x \notin S \\ 1 & \text{if } x \in S \end{cases}$$

This function is called the *characteristic function* of the set S.

Think of this as a "data structure" for representing S.

1.7 Example. Let U be the set $\{0,1,\ldots,9\}$. Let S be the set of even numbers in U. Then ch_S is (remember that, officially, a function is a set of ordered pairs):

$$\{(0,1),(1,0),(2,1),(3,0),(4,1),(5,0),(6,1),(7,0),(8,1),(9,0),$$

1.8 Check Your Reading. *1. Let U be the set $\{0,1,\ldots,9\}$ For each $S \subseteq U$ below, write down ch_S as a set of ordered pairs.*

(a) S is the set of numbers in U less than 5.

(b) S is the set of prime numbers in U.

(c) S is the empty set.

(d) S is U itself.

2. Let U be $\mathbb{N}$, the set of natural numbers. For each $S \subseteq U$ below, describe ch_S by writing down enough ordered pairs so that your reader gets the idea (you can't write down all of ch_S of course since it is an infinite set of pairs).

(a) S is the set of natural numbers less than 5.

(b) S is the set of prime numbers.

(c) S is the empty set.

(d) S is $\mathbb{N}$ itself.

1.9 Check Your Reading (Very Important). *Fix U.*

1. *Show that if $c : U \to \{0,1\}$ is any function from U to $\{0,1\}$ then there is a subset S of U of which c is the characteristic function.*
2. *Show that if S and T are two different subsets of U then ch_S and ch_T are different functions.*
3. *Show that if $c : U \to \{0,1\}$ and $d : U \to \{0,1\}$ are two different functions then they are the characteristic functions of two different sets.*

*Summarizing: Every subset has a characteristic function: that's Definition 1.6. Every function from U to $\{0,1\}$ **is** a characteristic function for some set: that's what 1 says. Different subsets give different characteristic functions: that's what 2 says. Different characteristic functions give different subsets: that's what 3 says.*

The takeaway message from all of this is the idea that subsets and characteristic functions are the same thing, just in different notation.

Putting this a bit more formally::

> *For any set U there is a natural one-to-one correspondence between $Pow(U)$ and the set of all functions $f : U \to \{0,1\}$*

Bitstrings

A bitstring is just a sequence of 0s and 1s. A bitstring can be finite or infinite. Bitstrings are yet another data structure to capture subsets. Suppose U is a set, and further suppose that **we have agreed on some linear ordering on** U. For example, suppose U is the set $\{0, 1, \ldots, 9\}$, and we agree on the natural numerical order on U. Then to represent, for example, the set of even numbers in U, we can just write the string

$$1\,0\,1\,0\,1\,0\,1\,0\,1\,0.$$

Think of a 0-or-1 entry in the ith place of the string as the answer to the question: is element i in my set?

As another example, the set of numbers in U less than 5 can be represented by the bitstring

$$1\,1\,1\,1\,1\,0\,0\,0\,0\,0.$$

1.10 Check Your Reading. *Explain clearly* why *we need to have established an ordering on U before we use bitstrings to capture subsets.*

When the universe U is finite, bitstrings comprise a common representation of sets in programming; you may have used it yourself. But mathematically the idea even applies to subsets of infinite sets U; all that is required is that we first establish a linear ordering on U.

1.11 Check Your Reading. *Refer to the "Check Your Reading" problem 1.9. Write down a paraphrase of that exercise that is about bitstrings rather than characteristic functions.*

1.12 Check Your Reading. *What's the relationship between characteristic functions and bitstrings: can we retrieve one from the other, for a given subset?*

1.4 Injective and Surjective

Here are some properties that a given function may or may not posses. Let $f : A \to B$. We say that f is

- *one-to-one,* or *injective,* if whenever $a \neq a'$ then $f(a) \neq f(a')$,
- *onto,* or *surjective,* if for every $b \in B$ there is at least one $a \in A$ such that $f(a) = b$.
- *bijective* if it is both injective and surjective.

A bijection is sometimes called a *one-to-one correspondence,* but this latter name invites confusion with the simple term "one-to-one" so we will stick to "bijection," especially since it is more fun to say.

1.13 Examples. The goal of these examples is to show that the properties of being injective and surjective are independent of each other.

1. Give an example of a function $f : \mathbb{N} \to \mathbb{N}$ that is injective and surjective. Now give a second example.
2. Give an example of a function $f : \mathbb{N} \to \mathbb{N}$ that is injective but not surjective. Now give a second example.
3. Give an example of a function $f : \mathbb{N} \to \mathbb{N}$ that is surjective but not injective. Now give a second example.
4. Give an example of a function $f : \mathbb{N} \to \mathbb{N}$ that is neither injective nor surjective. Now give a second example.

The properties of being injective and of being surjective are nicely preserved by function composition.

1.14 Lemma. *Let $f : A \to B$ and $g : B \to C$. Then*

1. *if f and g are injective then so is $g \circ f : A \to C$*
2. *if f and g are surjective then so is $g \circ f : A \to C$*

Proof.

1. To show that $g \circ f$ is injective, suppose that for some a_1 and a_2 we have $(g \circ f)(a_1) = (g \circ f)(a_2)$; we want to show that we must have $a_1 = a_2$. To say $(g \circ f)(a_1) = (g \circ f)(a_2)$ is to say $g(f(a_1)) = g(f(a_2))$. Since g is injective we conclude $f(a_1) = f(a_2)$. But then since f is injective we conclude $a_1 = a_2$.
2. To show that $g \circ f$ is surjective, suppose that c is an arbitrary element of C; we wish to show that there is some $a \in A$ such that $(g \circ f)(a) = c$. Since g is surjective there is some $b \in B$ with $g(b) = c$. Since f is surjective there is some $a \in A$ with $f(a) = b$. Putting all this together we see that $g(b) = g(f(a)) = c$ so indeed this a is the element we seek.

///

The empty set and singleton sets.

The empty set and singleton sets have some distinctive features. Here are some elementary facts to keep in mind, make sure they are clear to you.

Let S be a set with exactly one element. For any set A there is exactly one function from A to S. This function is surjective. For any set A there are precisely as many functions from S to A as there are elements of A. Each of those functions is injective.

Consider the empty set $\emptyset$. For any set B there is exactly one function from $\emptyset$ to B. This function is injective. For any set B there are *no* functions from B to $\emptyset$, except when B is also the empty set, in which case there is exactly one (bijective) function.

1.5 Inverses

The question of when a function $f : A \to B$ has a inverse function is interesting. Indeed it turns out to be very useful to be picky, and make a distinction between *left* and *right* inverses.

1.15 Definition. (Function Inverses) Let f be a function, $f : A \to B$.

- $g : B \to A$ is a *left inverse* for f if $g \circ f$ is the identity on A.
- $g : B \to A$ is a *right inverse* for f if $f \circ g$ is the identity on B.
- $g : B \to A$ is a *two-sided inverse* for f if it is both a left inverse and a right inverse.

  It is common practice (though potentially confusing) to simply say that "g is the inverse of f" when one really means that "g is a two-sided inverse of f.")

1.16 Check Your Reading. *Left inverses and right inverses are (at least, can be) different:*

- *Give an example of a function f such that there is some g with $f \circ g$ the identity yet $g \circ f$ is not the identity.*
- *Give an example of a function f such that there is some g with $g \circ f$ the identity yet $f \circ g$ is not the identity.*

1.6 Relating Injections, Surjections, and Inverses

Does every function have a two-sided inverse? Certainly not. It is useful to explore this carefully. So suppose $f : A \to B$. We want to build a function $g : B \to A$ that "undoes" f. The obvious thing to try, intuitively, is,

> *take some element b in B, we want to define what $g(b)$ is ... since g is supposed to undo f, we just take $g(b)$ to be the thing from A such that $f(a) = b$.*

This is the right thinking, but there are two things that can wrong in trying to make it happen.

1. For the b in question, there might not be *any* a from A such that $f(a) = b$.

   This problem can happen if f is not surjective.

2. Even if there is such an a, there might not be a unique one, so that we don't have a natural choice about what to choose as our $g(b)$.

   This problem can happen if f is not injective.

The discussion above suggests that if f is both injective and surjective then we can always build an inverse function g. What's even more interesting is that if f is injective, but not necessarily surjective, then we can get halfway, namely we can define a "one-sided" inverse. Similarly, f is surjective, but not necessarily injective, then we can get halfway, and define a one-sided inverse on the other side.

1.17 Theorem. *Let $f : A \to B$.*

1. *f is surjective if and only if f has a right inverse.*
2. *f is injective if and only if $A = \emptyset$ or f has a left inverse.*

Proof.

1. Suppose f is surjective. Define $g : B \to A$ as follows. For any $a = b \in B$, choose an arbitrary $a \in A$ such that $f(a) = b$. Such an a exists since f is surjective. Then g is a right inverse for f since $f(g(b)) = b$ by definition.

   Conversely, let g be a right inverse for f, so that $f \circ g = id_B$. Let $b \in B$; we seek $a \in A$ such that $f(a) = b$. The element $g(b)$ works: $f(g(b)) = id_B(b) = b$.

2. Suppose f is injective. If $A \neq \emptyset$, we define $g : B \to A$ as follows. First choose some arbitrary $a_0 \in A$. We can do that because we've assumed $A \neq \emptyset$. Now we define g as follows. If b is in the image of A, choose $g(b)$ to be any a such that $f(a) = b$. If b is not in the image of A, choose $g(b)$ to be a_0. It is easy to check that $g \circ f = id_A$.

   Conversely, let g be a left inverse for f, so that $g \circ f = id_A$. To show f injective, suppose we have $f(a_1) = f(a_2)$, we want to show $a_1 = a_2$. Since $f(a_1) = f(a_2)$, we have $g(f(a_1)) = g(f(a_2))$. Since $g \circ f = id_A$ this equation reduces to $a_1 = a_2$.

///

1.7 "Pointless" Reasoning

It can be efficient and satisfying to reason about functions as algebraic objects rather than working with "points" in sets. Theorem 1.17 is a great tool for this.

Here are a couple of examples.

Recall Lemma 1.14 about injectivity and surjectivity being preserved by composition. We proved that by chasing elements around, which is fine, but we can give proofs with a more "algebraic" flavor, as follows.

Lemma 1.14 (again). *Let $f : A \to B$ and $g : B \to C$. Then*

1. *if f and g are injective then so is $g \circ f : A \to C$*
2. *if f and g are surjective then so is $g \circ f : A \to C$*

Proof. Before reading, draw a little picture of f and g mapping between A and B and B and C, and follow along.

1. To show that $g \circ f$ is injective, we will show that it has a left inverse. Since f is injective, it has a left inverse f'. That is, $f' \circ f = id$. Since g is injective, it has a left inverse g'. That is, $g' \circ g = id$. Then the function $f' \circ g'$ is a left inverse for $g \circ f$, as we can see by calculation. Since composition is associative we can suppress parentheses and group things however it is convenient:

$$\begin{aligned}(f' \circ g') \circ (g \circ f) &= f' \circ g' \circ g \circ f \\ &= f' \circ id \circ f \\ &= f' \circ f \\ &= id\end{aligned}$$

2. To show that $g \circ f$ is surjective, we will show that it has a right inverse. In the same spirit as the previous part, using the fact that f has a right inverse f' and g has a right inverse g' , we show that the function $f' \circ g'$ is a right inverse for $g \circ f$,

$$\begin{aligned}(g \circ f) \circ (f' \circ g') &= g \circ f \circ f' \circ g' \\ &= g \circ id \circ g' \\ &= g \circ g' \\ &= id\end{aligned}$$

///

For another example of the power of pointless arguments, let's return to Theorem 1.17. If you think hard about the construction we did there you will see that we actually proved a bit more than we claimed. Namely, when f is a surjective function the right inverse for f that we built in Theorem 1.17 happened to be an *injective* function. And, when f is an injective function the left inverse for f that we built in Theorem 1.17 happened to be an *surjective* function. This information is buried in the middle of the proof. But somewhat amazingly we can recover these stronger claims just from the statement of the original theorem.

The trick is to notice that if we have any two functions, say h_1 and h_2 with the property

$$h_1 \circ h_2 = id$$

then, as we showed in Theorem 1.17, h_1 is necessarily surjective, and h_2 is necessarily injective.

This leads us to the following fact, which is often quite useful.

1.18 Lemma. *Let A and B be sets, with A not empty. There is an injective function from A to B if and only if there is an surjective function from B to A.*

Proof. Suppose $f : A \to B$ is injective. Since f is injective it has a left inverse, $g : B \to A$ with $g \circ f = id_A$. This g is the function we seek: since g has a right inverse (namely f) it is surjective.

Conversely, suppose $g : B \to A$ is surjective. Since g is surjective it has a right inverse, $f : A \to B$ with $g \circ f = id_A$. This f is the function we seek: since f has a left inverse (namely g) it is injective. ///

It is easy enough to prove the above results directly, by chasing points around. But the algebraic proofs are much easier.

Another good example of the power of pointless reasoning is Problem 9.

1.8 Relations

An *n-ary relation* is simply is subset of $A_1 \times A_2 \cdots \times A_n$ where $A_1, A_2, \ldots, A_n$ are sets. The most common case is that of a *binary relation*, where $n = 2$. And the most common case of all is when $R \subseteq A \times A$ is a binary relation from a set to the same set; in this case we speak of a binary relation *on* A.

These two expressions mean the same thing: $(a,b) \in A$ and $R(a,b)$. People tend to use the first when thinking "set-theoretically" and tend to use the second when thinking "logically."

Notice that a binary relation on A is the same thing as a directed graph with A as the set of nodes. Sometimes it is helpful to draw pictures of relations in this way.

1.19 Examples. Here are some binary relations.

1. On the integers: $R(a,b)$ if $a-b$ is divisible by 2.
2. On the integers: $R(a,b)$ if a divides b.
3. On the real numbers: $R(a,b)$ if $a^2+b^2 \leq 1$.
4. On the real numbers: $R(a,b)$ if $a-b$ is an integer
5. On the real numbers: $R(a,b)$ if $|a-b| \leq 1$
6. Let A be the set of *pairs* (n,d) of integers with $d \neq 0$. Define $R((n,d),(n',d'))$ if $n*d' = n'*d$
7. On the integers: $R(a,b)$ if ab is a perfect square.

1.9 Composition and Inverse on Relations

1.20 Definition (Relation Composition)**.** Let R be a binary relation from A to B, and let P be a binary relation from B to C. The *composition* $P \circ R$ is the binary relation from A to C given by: $(x,z) \in P \circ R$ if there is some $y \in B$ such that $(x,y) \in P$ and $(y,z) \in R$.

1.21 Definition (Relation Inverse)**.** Let R be a binary relation from A to B. The *inverse* R^{-1} of R is binary relation from B to A given by: $(x,y) \in R^{-1}$ if and only if $(y,x) \in R$.

The topic of inverses is much more interesting when talking about functions, as opposed to relations.

If R is (just) a binary relation from A to B then we can *always* define the inverse of R (we just did so, in Definition 1.21). We just swap all the ordered pairs in R. Why, then was the topic of inverses of functions so fussy? After all, a function is just a certain kind of relation. Here's what's going on. Suppose f is a *function* from A to B. Since f is a (special kind of) relation, then sure, there is an inverse of f. *But the inverse of f, in the relational sense, might not be a function!*

1.22 Check Your Reading. *We already defined composition of functions, and any function is a also a relation. So it would be embarrassing if those definitions didn't match up. Convince yourself that if $f : A \to B$ and $g : B \to C$ are functions, then the composition of f and g considered as relations (that is, via Definition 1.20) really is the same the composition as functions (that is, via Definition 1.3).*

1.10 Properties of Relations

Here is a list of properties that a given binary relation may or may not posses. Let $R \subseteq A \times A$. We say that R is

- *reflexive* if for every $a \in A$: $R(a,a)$.
- *symmetric* if for every $a,b \in A$: $R(a,b)$ implies $R(b,a)$
- *transitive* if for every $a,b,c \in A$: $R(a,b)$ and $R(b,c)$ imply $R(a,c)$.
- *antisymmetric* if for every $a,b \in A$: $R(a,b)$ and $R(b,a)$ imply $a = b$.
- *complete* if for every $a,b \in A$: $R(a,b)$ or $R(b,a)$.

Try to describe what each of the properties above corresponds to in terms of graphs. (For example, to say that R is reflexive is to say that at, as graph, the relation has a self-loop at each node.)

1.23 Check Your Reading. *For each subset S of the properties { reflexive, symmetric, transitive, antisymmetric } try to define a set A and a binary relation R on A that has those properties in S but not the others; try to make A and R as small as possible. (Finding really examples is a good way to make sure you have isolated the* crucial *components of the point you are making.)*

Note that there are 16 parts to this question! Feel free to draw your answers as directed graphs.

1.10.1 Equivalence relations and partitions

A relation R is an *equivalence relation* if it is reflexive, symmetric, and transitive.

A *partition* of a set A is a set $\mathcal{P} = \{A_1, A_2, \dots\}$ of subsets of A, with the two properties that (i) $A_i \cap A_j = \emptyset$ if $i \neq j$, and (ii) $\bigcup_i A_i = A$. Equivalence relations and partitions are different ways of expressing the same situation. If R is an equivalence relation on A, then we obtain a partition of A by putting two elements in the same partition element precisely when they are R-related. And if $\mathcal{P}$ is a partition, then we obtain an equivalence relation from $\mathcal{P}$ by defining two items to be related precisely when they are in the same element of the partition.

1.24 Check Your Reading. *For each of the relations earlier that you determined to be equivalence relations, describe the associated partition.*

1.10.2 Order relations

Let R be a binary relation on a set A, in other words let R be a subset of $A \times A$.

- R is a *preorder* if it is reflexive, and transitive.
- R is a *partial order* if it is reflexive, antisymmetric, and transitive.
- R is a *total order* if it is reflexive, antisymmetric, transitive, and complete.

1.11 Problems

Problem 1.

For this problem, let A be a finite set with a elements, let B be a finite set with b elements, and let S be some arbitrary singleton set (*i.e.* a set with one element).

1. How many functions are there from A to S?
2. How many functions are there from S to A?
3. How many functions are there from A to $\emptyset$?
4. How many functions are there from $\emptyset$ to A?
5. How many functions are there from A to B?

Problem 2.

a) For each function below, decide whether it is injective or not. If it is injective, just say so. If it is not injective, give a concrete reason why not.

1. $f : \mathbb{R} \to \mathbb{R}$, defined by $f(x) = x^2$
2. $g : \mathbb{R} \to \mathbb{R}$, defined by $g(x) = 2^x$

b) For each function below, decide whether it is surjective or not. If it is surjective, just say so. If it is not surjective, give a concrete reason why not.

1. $f : \mathbb{R} \to \mathbb{R}$, defined by $f(x) = x^2$
2. $g : \mathbb{R} \to \mathbb{R}$, defined by $g(x) = 2^x$

Problem 3.

Suppose we re-do the examples in Example 1.13 but instead of considering functions to and from the infinite set $\mathbb{N}$, we try to find examples that are functions from a *finite* set to itself. Things are different then. Let's explore the following question. For which of the four parts of that problem could we have asked for a function $f : A \to A$, with A finite, and for which would it be impossible? Give counterexamples for the claims which are (now) false.

Problem 4.

Suppose $f_1 : A_1 \to B_1$ and $f_2 : A_2 \to B_2$ are each injective. Consider the natural function $f : (A_1 \times A_2) \to (B_1 \times B_2)$ defined by

$$f(a_1, a_2) = (f_1(a_1), f_2(a_2))$$

Prove that this f is injective.

Problem 5.

Suppose $f_1 : A_1 \to B_1$ and $f_2 : A_2 \to B_2$ are each surjective. Consider the natural function $f : (A_1 \times A_2) \to (B_1 \times B_2)$ defined by

$$f(a_1, a_2) = (f_1(a_1), f_2(a_2))$$

Prove that this f is surjective.

Problem 6.

Suppose $f : A \to B$ and $g : B \to C$ are functions.

Here we ask the question(s): can the results in Lemma 1.14 can somehow be "reversed?" That is, based on knowing something about the injectivity or surjectivity of a function $(g \circ f)$, can we conclude anything about the injectivity or surjectivity of g, or of f?

For each of the following claims, (i) decide whether it is always true, then (ii) if it is true, prove it, and if it is false, find a counterexample.

For your proofs, use the facts in Theorem 1.17 to give a nice "calculational" proof.

For your counterexamples, give a specific pair of functions f and g making the statement false (don't forget to say what A, B, and C are).

a) If $(g \circ f)$ is injective then g is injective.

b) If $(g \circ f)$ is surjective then g is surjective.

c) If $(g \circ f)$ is injective then f is injective

d) If $(g \circ f)$ is surjective then f is surjective

Problem 7.

Here we explore the question, "can a function have more than one right inverse, or are right-inverses unique?"

Decide what you think and then either **prove** the following assertion or **disprove** it by giving a specific counterexample.

$$\textit{if } f : A \to B, \textit{ and } g_1 \textit{ and } g_2 \textit{ satisfy } f \circ g_1 = id_A \textit{ and } f \circ g_2 = id_A, \textit{ then } g_1 = g_2.$$

Problem 8.

Here we explore the question, "can a function have more than one left inverse, or are left inverses unique?"

Decide what you think and then either **prove** the following assertion or **disprove** it by giving a specific counterexample.

$$\textit{if } f : A \to B, \textit{ and } g_1 \textit{ and } g_2 \textit{ satisfy } g_1 \circ f = id_B \textit{ and } g_2 \circ f = id_B, \textit{ then } g_1 = g_2.$$

Problem 9.

Recall that in order for g to be an "inverse" of a function f we required that g serve both as a left inverse for f and also as a right inverse for f. This suggests a question: suppose we have a function f that has a right inverse g_1 and a left inverse g_2. Will f necessarily have an inverse, that is, a single function g that simultaneously does the job of being a left- and right inverse? The answer is yes:

Prove that if g_1 is a right inverse for f and g_2 is a left inverse for f, then in fact $g_1 = g_2$. (And so, this one function serves as both a left- and a right-inverse of f, that is, a two-sided inverse of f.)

Hint. This is an excellent example of the use of pointless reasoning. Start with the equation $(f \circ g_1) = id_B$ and do a little algebraic reasoning do derive $g_1 = g_2$.

Problem 10.

Fix a set U, and subsets S_1, S_2 if U. Let s_1 and s_2 be the respective bitstrings representing S_1 S_2 (under some ordering of U that we don't need to know about).

How would you calculate the bitstring representing $S_1 \cap S_2$ from s_1 and s_2? (It is a simple operation).

Problem 11.

Same question as Problem 10, for $S_1 \cup S_2$, $S_1 - S_2$, and $\overline{S_1}$.

Problem 12.

True or false? *If R is a relation that is a function, then R^{-1} is a function.*

If true, give a proof, if false, give a *specific* counterexample.

Problem 13.

1. List all the binary relations on the set $\{1,2\}$. *Hint:* There are 16 of them.
2. How many binary relations are there on a set with n elements?

Problem 14.

Give an example to show that relation composition is not commutative.

Problem 15.

For each relation R from Examples 1.19 determine which of the properties (reflexive, symmetric, transitive, antisymmetric, complete) it enjoys.

Problem 16.

An important example of an equivalence relation is the following. Let $f : A \to B$. Define the relation R on A by: $R(a_1, a_2)$ if $f(a_1) = f(a_2)$. Convince yourself that this is indeed an equivalence relation. What is the associated partition of the set A?

Consider some specific functions (you choose which ones). For each function, write down (draw pictures of) the equivalence relation generated by each function

Problem 17.

Let A be the set $\{a, b, c\}$. List all the equivalence relations on A. It will be easiest to present all the partitions. Hint: there are 5 equivalence relations.

How many equivalence relations are there on the set $\{a, b, c, d\}$?

If I asked you how many equivalence relations there were on the set $\{a, b, c, d, e\}$, you'd have a right to be mad at me: there are 52.

Problem 18.

Find natural examples of relations over the integers that are

1. preorders but not partial orders,
2. partial orders but not total orders,

Now, do these two problems again, but give examples where R is a relation over a finite set A, and where R and A are as small as you can make them.

Problem 19.

Consider the "divides" relation $a \mid b$, which holds a divides b evenly, that is, if there exists c such that $ac = b$. (Note that according to this definition, 0 divides 0....)

1. Suppose a, b, and c are taken to range over the natural numbers (that is, the non-negative integers). Is divides a preorder? Is it a partial order? Is it an equivalence relation?
2. Same questions, except now suppose a, b, and c are taken to range over the integers.
3. Same questions, except now suppose a, b, and c are taken to range over the rational numbers. (Be careful, this is a little tricky.)

Problem 20.

Suppose R is a preorder on A. Decide whether the following are true or false. If true, give a proof, if false, give a *specific* counterexample.

1. If R is an equivalence relation then R^{-1} is an equivalence relation.
2. If R is a partial order then R^{-1} is a partial order.
3. If R is a total order then R^{-1} a total order.

Chapter 2

Cardinality

2.1 Introduction

When A and B are finite sets, everyone knows what it means to say that A and B "have the same size" or that A is "bigger than" B. The big idea in this section is to develop a useful way of talking about these ideas when A and B are allowed to be *arbitrary* sets.

The fundamental question is this: is there a sensible mathematical way to capture when one set X is "bigger than" another set Y? One obvious idea is to say simply say that X is bigger than Y if Y is a proper subset of X. That's precise, and useful, sure. But it's limited. For example, if that is all we mean by "bigger" then the set $X = \{a, b, c\}$ would not be bigger than the set $Y = \{0, 1\}$. What we really want is something which captures the intuition of "X has more elements than Y" in such a way that it applied even when the sets are not necessarily finite.

Mathematicians have indeed defined and explored such a notion, called "cardinality." The study of cardinality is a highly complex subject. Most computer scientists only need to be familiar with the basic ideas. We will cover the basic ideas in sections 2.2 through 2.7. After that we will give a glimpse of the general theory.

2.2 Countable and Uncountable

To motivate the next definition, think about what is really going on when you count the elements of a finite set X. You point to each element and say, in turn, "0", "1", "2", and so on[1] until you stop, at, say "99." What you are doing there is constructing a function from X to the set $\{0, 1, \ldots, 99\}$. It's a function because you use up all of X

[1] Maybe you start counting with "1", ... but you get the idea.

and you only say one number when pointing any given object. And that function is injective, because you don't repeat any numbers as you count.

So to count a finite set is to build an injective function from that set into an initial segment of $\mathbb{N}$. It is natural, then, to say that to "count" an infinite set X is to build an injective function from X into all of $\mathbb{N}$. We are allowed to use all of $\mathbb{N}$ for our answers if we care to, though we don't have to. But an essential point is that in order to really make a function from X, we have to give a value for every point in X. What will be fascinating to see (soon) is that for some sets X this is not possible.

When it is possible, we will say that our set X is *countable*. You might object to using that term, if the word "countable" suggests to your intuition that we have to *finish* counting. If so, you just have to accept the fact that mathematicians have agreed to use term "countable" in this more generous way.

Let's make an official definition and explore some examples.

2.1 Definition (Countable and Uncountable)**.** A set A is *countable* if there is an injective function $f : A \to \mathbb{N}$. A is *uncountable* if A is not countable.

Note that any finite set is countable: if X can be written as $\{x_1, x_2, \ldots, x_n\}$ then we can map X injectively into $\mathbb{N}$ by simply sending x_i into i. Some authors reserve the word countable for infinite sets that can be injected into $\mathbb{N}$; in other words, for them, finite sets are not called "countable." But this is not the way we use the term: the more inclusive terminology that we use here is more typical.[2]

Note that by the results of Section 1, a set A is countable if there is a surjective function $g : \mathbb{N} \to A$. So that gives two ways to prove that a set X is countable: (i) exhibit a function $f : X \to \mathbb{N}$ that is injective, or (i) exhibit a function $f : \mathbb{N} \to X$ that is surjective.

What is not obvious at this point is whether there are *uncountable* sets. That is, you might very well be thinking that no matter what set X is, we can always inject it into $\mathbb{N}$. We will see soon enough that that is *not* true. For example, the real numbers $\mathbb{R}$ is not countable. In fact, if you prefer, you may want to look ahead to Theorem 2.4 to see an example of an uncountable set together with a proof of that.

But here we'll first do some concrete examples of countable sets.

2.3 First Examples of Countability

2.2 Examples. The following sets are countable.

[2]Sometimes, if we want to emphasize that a set A is countable, but not finite, we may use the expression *countably infinite*.

1. Very easy examples: $\{a,b,c\}$ is countable via the function $\{(a,0),(b,1),(c,2)\}$

2. The set $E = \{0,2,4,\ldots\}$ of even natural is countable.

   The inclusion function, mapping each (even) number to itself, is an injection from E to $\mathbb{N}$.

3. The set $\mathbb{Z}$ of integers is countable.

   The function $f : \mathbb{Z} \to \mathbb{N}$ defined by $f(z) = \begin{cases} 2z & z \geq 0 \\ -(2z+1) & z < 0 \end{cases}$ is injective.

4. The set $\mathbb{N}^2 = \mathbb{N} \times \mathbb{N}$ of ordered pairs (a,b) of natural numbers is countable.

   Use the function $g : \mathbb{N}^2 \to \mathbb{N}$ defined by $g(x,y) = 2^x 3^y$. Check for yourself that g is indeed injective; the crucial fact is the unique factorization property of the integers.

5. We can generalize the previous example. The set $\mathbb{N}^k = \mathbb{N} \times \mathbb{N} \times \cdots \times \mathbb{N}$ of ordered k-tuples of natural numbers is countable.

   To show $\mathbb{N}^k \preceq \mathbb{N}$, we define a function $g : \mathbb{N}^k \to \mathbb{N}$ as follows. Let p_i denote the ith prime number, so that $p_0 = 2, p_1 = 3, p_5 = 13$ and so on. Then define $g(x_0, x_1, \ldots, x_{k-1}) = 2^{x_0} 3^{x_1} \ldots p_{(k-1)}^{x_{k-1}}$.

   Thus: $\mathbb{N}^k$ is countable.

6. In fact we can generalize even further. The set $\mathbb{N}^*$ of all finite sequences drawn from the natural numbers is countable.

   Here is a 1-1 function $f : \mathbb{N}^* \to \mathbb{N}$. We use the prime-number-exponentiation trick once again. If x is a finite string of natural numbers, $x = x_1 \ldots x_k$ we take $f(x)$ to be the number $2^{x_1+1} 3^{x_2+1} \ldots p_k^{x_k+1}$, where p_k is the kth prime number. As usual, this is 1-1 because of the uniqueness of prime factorization. (Do you see why we could not use the formula $f(x) = 2^{x_1} 3^{x_2} \ldots p_x^{x_k}$?)

   As a corner case, note that what we have said so far does not say what f should do on the empty string λ. But since no non-empty string will map to 0, we can define $f(\lambda) = 0$ and we are then done.

2.4 An Uncountable Set

It's time to show that all those countability results above were not redundant; that is, it is time to see that some sets are *not* countable.

Let $\mathcal{B} = \{0,1\}^{\mathbb{N}}$ denote the set of all **infinite** bitstrings, that is, the set of all infinite sequences of 0s and 1s. The formal definition of a bitstring is that it is a function $\beta : \mathbb{N} \to \{0,1\}$. So actually a bitstring is a characteristic function over $\mathbb{N}$.

Remember our work with characteristic functions and their relationship with subsets. Now we see that each bitstring represents a subset of $\mathbb{N}$ and each subset of $\mathbb{N}$ is captured by a bitstring.

2.3 Theorem. *The set $\mathcal{B}$ of infinite bitstrings is uncountable.*

Proof. We prove that any function $g : \mathbb{N} \to \mathcal{B}$ must fail to be surjective. So let an arbitrary $g : \mathbb{N} \to \mathcal{B}$ be given. That is, for any $i \in \mathbb{N}$, $g(i)$ is an infinite bitstring. Now we claim that the following infinite bitstring β is not in the image of g:

$$\beta(i) = \begin{cases} 0 & \text{the } i\text{th entry of } g(i) \text{ is } 1 \\ 1 & \text{the } i\text{th entry of } g(i) \text{ is } 0 \end{cases}$$

An equivalent, more compact, definition way of expressing the above is just

$$\beta(i) = 1 - g(i)(i)$$

Now we show that β is not in the image of g. That is, we show that for each i, β cannot be $g(i)$. And to do that we only need to show that the bitstring β differs from each bitstring $g(i)$ in at least one entry. But our β differs from $g(i)$ in the ith place: β was built expressly to satisfy this condition. ///

This proof technique is called "diagonalization;" the next section shows why.

2.4.1 Uncountability in Pictures

Here is the proof of the uncountability of $\mathcal{B}$ presented visually. We stress that we are showing **the same proof** as above, just drawing a picture of it.

Imagine a table, extending infinitely down and to the right, whose rows are the bitstrings in the image of g.

$g(0)$	0	0	0	0	0	…
$g(1)$	1	1	1	1	1	…
$g(2)$	1	0	1	0	1	…
$g(3)$	0	1	0	1	0	…
$g(4)$	0	0	1	1	0	…
⋮						

The picture above suggests the situation where

- $g(0)$ is the constant-0 bitstring;
- $g(1)$ is the constant-1 bitstring;
- $g(2)$ alternates 0s and 1s starting with 1;
- $g(3)$ alternates 0s and 1s starting with 0;
- $g(4)$ has a 1 in all the places corresponding to prime numbers; ...

Then β is the bitstring that goes down the diagonal and flips all the entries there. In the concrete example g above, β would start out

$$1\,0\,0\,0\,1\,\ldots$$

And we can see that this β can't be equal to any of the rows. So g isn't surjective (it misses β).

2.5 $Pow(\mathbb{N})$ is Uncountable

Reminder: the set $Pow(\mathbb{N})$ is the set of all subsets of $\mathbb{N}$.

Since we have observed that the infinite bitstrings are in one-to-one correspondence with the subsets of $\mathbb{N}$, the fact that the set $\mathcal{B}$ is uncountable already proves that $Pow(\mathbb{N})$ is uncountable.

But it is worth presenting an uncountability argument for $Pow(\mathbb{N})$ directly, even though it is, under the hood, the same argument as we saw in the last section. The advantage of phrasing the proof as we do here is that it is not tied to sets—like $\mathbb{N}$—that can be laid out naturally in a line. We will see that later, when we explore what Theorem 2.4 tells us when we lift our eyes from $\mathbb{N}$ and $Pow(\mathbb{N})$.

2.4 Theorem. *The set $Pow(\mathbb{N})$ is uncountable.*

Proof. We show that there can be no function $g : \mathbb{N} \to Pow(\mathbb{N})$ that is surjective. So let $g : \mathbb{N} \to Pow(\mathbb{N})$ be arbitrary; we claim that there is some element of $Pow(\mathbb{N})$, that is, some subset of $\mathbb{N}$, that is not in the image of g. Here is the definition of one such set, call it D

$$x \in D \quad \text{if and only if} \quad x \notin g(x)$$

To justify that this works, we must argue that for any $i \in \mathbb{N}$, the set $g(i)$ is not D. But the argument for that is simply this: for every element i, the sets $g(i)$ and D differ about whether i is in them:

- If $i \in g(i)$ then by the definition of D given above, $i \notin D$. So $g(i)$ and D differ as to whether i is a member, so $g(i)$ is not D.
- If $i \notin g(i)$ then by the definition of D given above, $i \in D$. So $g(i)$ and D differ as to whether i is a member, so $g(i)$ is not D.

So no matter what i is, $g(i)$ is not D. Thus D is not in the image of g, that is, g is not surjective. ///

Try to be clear in your mind that the two proofs we have seen, in the language of subsets and in the language of bitstrings have the same strategy and content.

2.6 New Examples from Old

Suppose C is some set that we have *already* proven to be countable. Now suppose we can exhibit a function $f : X \to C$ that is injective. Then that is good enough to prove X to countable. Why? Well, having said that C is countable, we know there is some function $f' : C \to \mathbb{N}$ that is injective. If we do succeed in building an injective $f : X \to C$, then the composition $f' \circ f$ maps X into $\mathbb{N}$, and it is injective since it is the composition of two injectve functions.

The reason this is useful is that sometimes, for some set X that you care about, it will be easier to map X into some set C that is related to X in some way rather than try to map X into $\mathbb{N}$. As long as you have already done the work to show C countable, mapping X injectively into C is good enough.

Using very similar reasoning, we can see that if C is some set known to be countable, defining a *surjective* function g from C to X is sufficient for proving X to be countable. Work through the justification of this for yourself; the key point is that the composition of surjective functions is surjective.

Summarizing:

2.5 Lemma. *In order to prove a set X countable, it is sufficient to do either of the following things:*

1. *Show that there is an injective $f : X \to C$ where C is known to be countable.*
2. *Show that there is an surjective $f : C \to X$ where C is known to be countable.*

2.6 Examples.

1. A subset of a countable set is countable.

   Let $A \subseteq B$ and suppose B is countable. The inclusion function $i : A \to B$ is obviously injective, so by part 1, A is countable.

2. The set $\mathbb{Z}^2$ of ordered pairs of integers is countable.

   Since: by part 1 it suffices to find an injective $f : \mathbb{Z}^2 \to \mathbb{N}^2$. But since we know that there is an injective $f_1 : \mathbb{Z} \to \mathbb{N}$. we have the function we want using Problem 4 .

3. The set $\mathbb{Q}$ of rational numbers is countable.

   It will be easiest to exhibit a *surjective* function g from $\mathbb{Z}^2$ to $\mathbb{Q}$, and then use part 2. Suppose we have a pair $(p,q) \in \mathbb{Z}^2$. We map this to the rational number p/q. Clearly every rational number is in the image of this function. We do have to be careful when $q = 0$ since this won't make a legal fraction; let's just return something random like $1/2$. So our function is:

$$g(p,q) = \begin{cases} p/q & \text{if } q \neq 0 \\ 1/2 & \text{otherwise} \end{cases}$$

   Note that two different fractions a/b and c/d can denote the same rational number. This means that the function g is not injective. But that doesn't matter.

We can use ideas similar to those in Lemma 2.5 to prove sets uncountable as well. Let's record those idea here.

2.7 Lemma. *In order to prove a set X uncountable, it is sufficient to do either of the following things:*

1. *Show that there is an injective $f : U \to X$ where U is known to be uncountable.*
2. *Show that there is an surjective $f : X \to U$ where U is known to be uncountable.*

Proof. These assertions are really just the contrapositives of the assertions in Lemma 2.5. For instance suppose $f : U \to X$ is injective. The first result in Lemma 2.5 says that if X is countable then U is countable; this is equivalent to saying that if U is not countable then X is not countable.

The same reasoning applies to our second assertion: it is just the contrapositive of the second result in Lemma 2.5. ///

2.8 Examples.

1. A superset of an uncountable set is uncountable.

   Let $A \subseteq B$ and suppose A is uncountable. The inclusion function $i : A \to B$ is obviously injective, so by part 1, B is uncountable.

2. The set [0, 1], the closed unit interval of real numbers, is uncountable.

   Since: by part 1 it suffices to exhibit an injective $f : \mathcal{B} \to [0,1]$. The idea is to take a bitstring β, put a decimal point in from of it, and view it as the decimal representation of a real number in $[0,1]$.

   By the way, it is tempting to be cute and read the representation in binary notation, since it is after all composed only of 0s and 1s. But that introduces the issue that there can be more than one string representing a real number. For example, in binary notation these both represent $1/2$[3]

   $$.0111111111... \quad \text{and} \quad .1000000000\ldots$$

   So our function wouldn't be injective. If we interpret things in base-10 then we dodge that annoyance.

3. The set $\mathbb{R}$ of real numbers is uncountable.

   Having shown [0,1] uncountable, this is just an instance of the general fact that a superset of an uncountable set is uncountable. No surprise, but it is worth noting that this is an instance of the first part of Lemma 2.7, using inclusion as our injective function.

2.7 Countability, Unions and Product

Countability and Union

If A_0 and A_1 are countable sets then the union $(A_0 \cup A_1)$ is countable. Indeed this fact extends to any finite union $(A_0 \cup A_1 \cup A_2 \cup \cdots \cup A_k)$ of countable sets.

Perhaps surprisingly, it even extends to an *infinite* union of countable sets, as long as the number of sets we take the union of is only countable. We prove that now. The result about a finite union $(A_0 \cup A_1 \cup A_2 \cup \cdots \cup A_k)$ will follow from the fact about the infinite union, as we show after the proof.

2.9 Lemma. *If $A_0, A_1, A_2, \ldots$ is a family of countable sets indexed by the natural numbers then the union $(A_0 \cup A_1 \cup A_2 \cup \ldots)$ is countable.*

Proof. Since each A_i is countable, for each i there is an injection $f_i : A_i \to \mathbb{N}$. We want to glue these functions together to make a single function mapping the union of the A_i into $\mathbb{N}$. We need to take care of the fact that a given a in the union might be in more than one of the A_i. That's not such a problem: for any a in the union there is a

[3] Remember from your calculus class! The .01111... represents a geometric series, which sums to 1/2.

least i such that $a \in A_i$, so we can imagine our new function doing the same thing to a as f_i does. If we just do that we will have defined a function from the union to $\mathbb{N}$. But that would not be good enough, because that function is unlikely to be injective: we can certainly imagine some $a \in A_{17}$ and $a' \in A_{23}$ such that $f_{17}(a)$ and $f_{23}(a')$ are the same value.

So instead, we define different functions $h_i : A_i \to \mathbb{N}$, based on the f_i, namely $h_i(a) = p_i^{f_i(a)+1}$ where p_i is the ith prime number. That is, take the function f_i and spread it out over $\mathbb{N}$ by ensuring that all the values lie on the powers of the ith prime number. Certainly each h_i is an injection. But even better, for different i and j, the values h_i takes on never intersect with the values that h_j takes on.[4]

So now we can define $h : (A_0 \cup A_1 \cup A_2 \cup \dots) \to \mathbb{N}$ by $h(a) = h_i(a)$ where i is least such that $a \in A_i$. This is an injection, so $(A_0 \cup A_1 \cup A_2 \cup \dots)$ is countable. ///

We get the finite-union result easily now.

2.10 Corollary. *If $A_0, A_1, A_2, \dots A_k$ is a finite family of countable sets then the union $(A_0 \cup A_1 \cup A_2 \cup \dots A_k)$ is countable.*

Proof. Just use Lemma 2.9 by taking each of the $A_{k+1}, A_{k+2}, \dots$ in that statement to be the empty set. ///

2.11 Examples.

1. The set $\mathbb{N}^*$ of finite sequences of natural numbers is countable. We have already proved this but the Lemma 2.9 gives another proof.

   Letting λ denote the empty sequence , we have

   $$\mathbb{N}^* = (\{\lambda\} \cup \mathbb{N} \cup \mathbb{N}^2 \cup \mathbb{N}^3 \cup \dots).$$

   This is a union of countably many sets, each one of which is countable.

2. Let I denote the set of *irrational* numbers. Then I is uncountable. Since: every real number is either rational or irrational, that is

   $$\mathbb{R} = I \cup \mathbb{Q}$$

   We know that $\mathbb{Q}$ is countable. If, for sake of contradiction, I were countable, then $\mathbb{R}$ would be the union of two countable sets, hence countable. But we know $\mathbb{R}$ is uncountable.

[4] By the way, do you see why we set $h_i(a) = p_i^{f_i(a)+1}$ rather than simply $h_i(a) = p_i^{f_i(a)}$?

Countability and Cartesian Product

2.12 Lemma. *If $A_0, A_1, A_2, \ldots A_k$ is a* finite *family of countable sets then the Cartesian product $(A_0 \times A_1 \times A_2 \times \cdots \times A_k)$ is countable.*

Proof. To show $(A_0 \times A_1 \times A_2 \times \cdots \times A_k)$ countable we use the result of part 5 of Examples 2.2. That is, we know that $\mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \cdots \times \mathbb{N}$ (k times) is countable, so all we have to do is find an injection g from $(A_0 \times A_1 \times A_2 \times \cdots \times A_k)$ into $\mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \cdots \times \mathbb{N}$.

Since we are assuming that each A_i is countable we know that for each i there is an injection $f_i : A_i \to \mathbb{N}$. So we just take the product of those functions, that is we define $f : (A_1 \times A_2 \times \cdots \times A_k) \to (\mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \cdots \times \mathbb{N})$ by

$$f(a_1, \ldots, a_k) = (f_1(a_1), \ldots f_k(a_k))$$

It's easy to see that this is injective, based on the fact that each f_i is injective. ///

Caution. In contrast to the situation with union, we cannot extend Lemma 2.12 to infinitely many sets. Even if each of the sets we are taking the product of is pretty small!

2.13 Example. The infinite product

$$\{0,1\} \times \{0,1\} \times \{0,1\} \times \ldots$$

is uncountable.

This product is precisely our set $\mathcal{B}$ of infinite bitstrings.

We'll stop here in our basic discussion of countable and uncountable sets. There are several questions that we have not addressed than might have occurred to you.

- Are there sets that are bigger than $Pow(\mathbb{N})$?

  Answer: yes. In fact for any set X there are always bigger sets. We prove that below as Theorem 2.16.

- Can there be sets that can't be compared by our methods, or is it true that for any sets A and B there must be either an injection from A to B or an injection from B to A?

  Answer: Sets can always be compared by $\preceq$, but that is hard to prove. See a textbook on set theory.

- Suppose we have injections back and forth between A and B. Can we always build a bijection between A and B?

  Answer: yes. We prove that below as Theorem 2.18.

- Are there sets that are smaller than $\mathcal{B}$ yet still bigger than $\mathbb{N}$?

  Answer: yes and no. Literally! One can adopt either a "yes" answer or adopt a "no" answer, add it to the usual axioms of mathematics, and the result will still be consistent. We say a tiny bit about that below.

2.8 Cardinality in General

Here we lay a foundation for a general study of "cardinality".

The original ideas are due to the mathematician Georg Cantor, in the late 19th century. Do yourself a favor and have a look at the first few paragraphs of the Wikipedia article on Cantor. It's interesting and poignant to read about the reactions of Cantor's contemporaries to his work.

We use the technical term *cardinality* to remind ourselves that we are working with a precise mathematical concept; hopefully we can avoid getting overly influenced by naive intuitions about the "size" of infinite sets.

The big idea is to say that the cardinality of A is less than or equal to the cardinality of B just in case we can map A into B in a one-to-one fashion. For this we will write $A \preceq B$. That looks like an ordering relation, and indeed we will see that behaves a lot like an order, but one way it is different from the orderings you may be familiar with is the fact that $A \preceq B$ and $B \preceq A$ certainly won't imply that $A = B$. We can certainly have injective functions back-and-forth between two sets without them being the same set! Still, it is worth being able to record that A and B can be injected into each other, so we introduce the notation $A \approx B$ for this.

2.9 Key Definitions

2.14 Definition (Cardinality)**.** Let A and B be arbitrary sets.

- We define $A \preceq B$, pronounced *the cardinality of A is less than or equal to the cardinality of B*, to mean there is an injective function from A to B.
- We define $A \approx B$, pronounced *A and B have the same cardinality*, to mean that both $A \preceq B$ and $B \preceq A$ hold.
- We define $A \prec B$, pronounced *the cardinality of A is less than the cardinality of B*, to mean $A \preceq B$ but *not* $B \preceq A$.

Note that to say $A \prec B$ is to say that there is an injective function from A to B, but there can be no injective function from B to A. When A is not empty, it is also is equivalent to saying that there is an surjective function from B to A, but there can be no surjective function from A to B.

The set $\mathbb{N}$ of natural numbers is a useful benchmark set. Notice that the set A is infinite precisely if $\mathbb{N} \preceq A$. Earlier in this chapter we gave the name "countable" to the property $A \preceq \mathbb{N}$.

Some authors define $A \approx B$ differently, namely they say that $A \approx B$ if there is a bijection between A and B (you may have seen this definition if you came across cardinality somewhere before). That definition is intuitively satisfying. And it is mathematically equivalent to the one we've given here. That is, there is a bijection between A and B if and only if there are injections in each direction. But *that* theorem is pretty hard to prove; we give it later on, as Theorem 2.18. And the definition we have given above is actually much more convenient to work with in practice: it is often much easier to define two injections between a couple of sets than it is to define one bijection.

The case of finite sets As a warm-up, let's see what the definition of cardinality yields when we restrict attention to finite sets. As you can easily see, when A and B are finite, to say that there is an injective function from A to B, that is, to say $A \preceq B$, is equivalent to saying that the number of elements in A is no more than the number of elements in B. So to say that there is an injective function from A to B and there is an injective function from B to A, that is, to say $A \approx B$, is equivalent to saying that A and B have the same number of elements.

The reason to define cardinality the way did is that it allows us to compare sets without having to use numbers. And the reason for *that* is precisely that we can talk about functions between infinite sets even though we can't count them using numbers.

2.10 Cardinality Makes an Ordering

The notations for $\preceq$ and $\prec$ give the impression that these relation behave like orderings. But just using an "ordering-like" notation doesn't prove anything! Let's explore, and see which properties that orderings typically have are enjoyed by $\preceq$.

Certainly the relation $\preceq$ is reflexive. (Why?) Next, let's show that the relation $\preceq$ is transitive.

2.15 Lemma. *If $A \preceq B$ and $B \preceq C$ then $A \preceq C$.*

Proof. The result follows readily from the fact that the composition of injective functions is injective. Since $A \preceq B$ then there is an injective $f : A \to B$. Since $B \preceq C$ then there is an injective $g : B \to C$. The function $(g \circ f) : A \to C$ is injective, which means $A \preceq C$, as desired. ///

How about antisymmetry? Is it the case that if $A \preceq B$ and $B \preceq A$ then $A = B$? No, certainly not. That what the relation $\approx$ is all about.

So far we have said that $\preceq$ is a *preorder*. Is it a total order? That is, do we have that for any two sets A and B, either $A \preceq B$ or $B \preceq A$ (or both)? The answer is yes, but it is quite a deep result. We won't need to discuss this further, but it is worth knowing.

How about the relation $\prec$? It is not reflexive, by definition. It is transitive, as you can prove as an exercise (Problem 34).

2.11 An Endless Hierarchy

So finite sets have smaller cardinality than $\mathbb{N}$, and we have already seen that $\mathbb{N}$ have smaller cardinality than $Pow(\mathbb{N})$. Are there any sets with larger cardinality than $Pow(\mathbb{N})$?

Yes. In fact, no matter what set A you start with, you can always construct a larger one: namely, $Pow(A)$. The next proof shows this. As a matter of fact it is really the *same proof* as the proof of Theorem 2.4. We mean that in a strong sense: the core of the proof below is exactly the same text as the proof there, simply changing occurrences of "$\mathbb{N}$" there to a generic "A" here! Cantor's idea of diagonalization is uniform across all sets A.

2.16 Theorem (Cantor). *Let A be any set. Then $A \prec Pow(A)$.*

Proof. We first observe that $A \preceq Pow(A)$ by virtue of the injective function $f : A \to Pow(A)$ defined by $f(a) = \{a\}$.

To complete the proof that $A \prec Pow(A)$ we show that $Pow(A) \preceq A$ is false. To do this, we show that there can be no function $g : A \to Pow(A)$ that is surjective. So let $g : A \to Pow(A)$ be arbitrary; we claim that there is some element of $Pow(A)$, that is, some subset of A, that is not in the image of g. Here is the definition of one such set, call it D

$$x \in D \quad \text{if and only if} \quad x \notin g(x)$$

To justify that this works, we must argue that for any $a \in A$, the set $g(a)$ is not D. But the argument for that is simply this: for every element a, the sets $g(a)$ and D differ about whether a is in them:

- If $a \in g(a)$ then by the definition of D given above, $a \notin D$. So $g(a)$ and D differ as to whether a is a member, so $g(a)$ is not D.
- If $a \notin g(a)$ then by the definition of D given above, $a \in D$. So $g(a)$ and D differ as to whether a is a member, so $g(a)$ is not D.

So no matter what a is, $g(a)$ is not D. Thus D is not in the image of g, that is, g is not surjective. ///

It is important to note that the above proof works uniformly for any set A whatsoever, in particular whether A is finite or infinite.

2.17 Check Your Reading. *Try Cantor's diagonalization in a concrete example: let A be the set* $\{0,1,2\}$. *Now invent any function* $g : A \to Pow(A)$ *by drawing arrows. Trace through the proof of Cantor's Theorem for the g you invented, and see for yourself that you are building a subset of A not in the image of g.*

2.12 The Schröder-Bernstein Theorem

Suppose there is a bijection f between A and B. Clearly this means that there are injections back-and-forth between A and B, namely, f and f^{-1}. So having a bijection between A and B certainly yields $A \approx B$ according to our definition. But does the converse hold as well? If we have injections back and forth between A and B can we always build a bijection?

To see that this is not so obvious, consider the open and closed intervals (0,1) and [0,1] on the real number line. Clearly there is an injection $i : (0,1) \to [0,1]$, the inclusion function. It's also easy to construct an injective function $f : [0,1] \to (0,1)$, by halving numbers and shifting a bit to avoid 0:

$$f(x) = (.5)x + .25$$

But to come up with a bijection between [0,1] and (0,1) seems tricky. (For one thing, such a function could not be continuous, right?

But in fact we can always build a bijection if we have two injections. It is a tricky thing to prove, and it's a little complicated to decide who to give credit to for the result: see the Wikipedia article "Schröder-Berstein Theorem" for (one) history. At any rate the name "Schröder-Berstein Theorem" is standard.

2.18 Theorem (Schröder-Bernstein)**.** *If there is an injection from A to B and there is an injection from B to A then there is a bijection from A to B.*

Sketch. Let $f : A \to B$ and $g : B \to A$ be injective.

Define

$$A_0 = A - \text{image}(g) \qquad A_{n+1} = g[f[A_n]]$$

Now define $h : A \to B$ by

$$\begin{cases} f(x) & \text{if } x \in A_n \text{ for some } n \\ g^{-1}(x) & \text{otherwise} \end{cases}$$

Have to argue that g^{-1} makes sense for x not in any A_n.

Then argue that h is a bijection.

///

So now we know can say that

If $A \approx B$ then there is a bijection from A to B.

For many authors this is the *definition* of $\approx$. But as we have mentioned, it is much easier to work with the "injections both ways" definition of $\approx$ than with the "bijection" definition.

For instance, it's easy to define injections each way between the closed interval $[0,1]$ and the open interval $(0,1)$. Can you directly define a bijection between them?

2.13 The Continuum Hypothesis

Cantor's Theorem says that $A \prec Pow(A)$. When A is a finite set this expresses something familiar: if A has n elements then the power set of A has 2^n elements, and $n < 2^n$.

Are there sets strictly "in between" A and $Pow(A)$ in the sense of cardinality? For finite sets, sure: whenever A has more than one element, there are sets B with $A \prec B \prec Pow(A)$, simply because there are numbers strictly in between n and 2^n whenever n is bigger than 1.

But now consider an infinite set, say $\mathbb{N}$ to be specific. Question: are there any sets B strictly "in between" $\mathbb{N}$ and $Pow(\mathbb{N})$ in the sense of cardinality? That is, is there a set B whose cardinality is greater than that of the natural numbers and less than that of $Pow(\mathbb{N})$: $\mathbb{N} \prec B$ and $B \prec Pow(\mathbb{N})$? Since we know that $Pow(\mathbb{N}) \approx \mathbb{R}$, an equivalent way to ask this question is: is there a set B whose cardinality is greater than that of the natural numbers and less than that of the real numbers?

The assertion that there are no such sets is called *The Continuum Hypothesis* (abreviated CH). That name stems from the fact that the real number line is sometimes called the "continuum."

Cantor (and others) tried hard to prove Continuum Hypothesis. He couldn't prove it, though, and for good reason: it was eventually shown that the Continuum Hypothesis *can neither be proven nor disproven* from the usual axioms of mathematics. In logic jargon, the Continuum Hypothesis is *independent* of the axioms of mathematics.

This means that you can choose to do math under the assumption that Continuum Hypothesis is true or you can choose to do math under the assumption that it is false, and neither of these assumptions, once you pick one, leads to a contradiction. And in fact there are certain statements P that are proven in mathematical journals to be true under the assumption of Continuum Hypothesis, while $\neg P$ is proven under the assumption that Continuum Hypothesis is false. So you get to choose whether to believe P or to believe $\neg P$!

2.14 Problems

Problem 21.

Prove that the set of all *finite* subsets of $\mathbb{N}$ is countable.

Problem 22.

Prove that the set of all *infinite* subsets of $\mathbb{N}$ is uncountable.

Problem 23.

Prove that the set of all *co-finite* subsets of $\mathbb{N}$ is countable.

(A set X is co-finite if its complement $\overline{X}$ is finite.)

Hint. Use Problem 21.

Problem 24.

For each of the following sets, decide if they are countable or uncountable. Then prove your answer.

a) $\{0,1\}^*$, the set of all finite strings over the alphabet $\{0,1\}$.

b) $\mathbb{N}^*$, the set of all finite sequences drawn from the natural numbers.

c) The set $\mathbb{N}^\infty$, the set of all infinite sequences drawn from the natural numbers.

d) $\mathbb{Z}^*$, the set of all finite sequences drawn from the integers.

Problem 25.

Let $f : \mathcal{B} \to [0,1]$ be the function used in Example 2.8 part 2 to show [0, 1] to be uncountable. What is the image (the set of values returned) of f?

Problem 26.

Prove that if there is a bijection between X and Y and X is countable, then Y is countable.

Hint. This is easy, but give the argument.

Problem 27.

Prove that if there is a bijection between X and Y and X is uncountable, then Y is uncountable.

Hint. This is easy, but give the argument.

Problem 28.

Let P be the set of one-variable polynomials with integer coefficients. Show that P is countable.

Hint. A polynomial such as $3x^2 - 5$ can be represented by the sequence of its coefficients $\langle 3, 0, -5 \rangle$.

Problem 29.

Show that the set $\mathbb{N}^{[2]}$ of *unordered* pairs of natural numbers is countable.

Hint. You need to take a little care since $\{x, y\}$ is the same unordered pair as $\{y, x\}$.

Problem 30.

Show that the set $\mathbb{R}^{[2]}$ of *unordered* pairs of real numbers is uncountable.

Hint. You need to take a little care since $\{x, y\}$ is the same unordered pair as $\{y, x\}$.

Problem 31.

Show that the set of all right triangles is uncountable. (Consider two triangles to be the same if they are congruent).

Problem 32.

Show that the set of all right triangles with integer-length sides is countable. (Consider two triangles to be the same if they are congruent).

Problem 33.

Let $\mathcal{S}$ be the set of all infinite sequences of natural numbers. More formally, $\mathcal{S}$ is just $\{f \mid f : \mathbb{N} \to \mathbb{N}\}$. Certainly $\mathcal{S}$ is uncountable, since it is a superset of $\mathcal{B}$. In this problem we consider certain subsets of $\mathcal{S}$. For each set below tell whether it is countable or uncountable.

a) The set of all infinite sequences of $\mathbb{N}$ that are eventually 0.

b) The set of all infinite sequence of $\mathbb{N}$ that are eventually constant

c) The set of all infinite sequence of $\mathbb{N}$ that are always strictly increasing

d) The set of all infinite sequence of $\mathbb{N}$ that are always strictly decreasing $b(i) > \beta(i+1)$ (a trick question! Can you write down any examples?)

e) The set of all infinite sequences β of $\mathbb{N}$ such that for all i, $b(i) \geq \beta(i+1)$

Problem 34.

Prove that if $A \prec B$ and $B \prec C$ then $A \prec C$.

Hint. This isn't trivial: you can use Lemma 2.15 to make some progress but note that part of what you need to establish is that $C \preceq A$ fails!

Problem 35.

What is $Pow(\emptyset)$? What does Cantor's Theorem say when $A = \emptyset$?

Problem 36.

Prove that if A is infinite then $Pow(A)$ is uncountable.

Chapter 3

Trees

Since things below are subtle and can get confusing, we will strive for a high level of formality, to avoid any ambiguity. In particular we will need to agree on a definition of when a set is finite and when a set is infinite. We take for granted that for any given natural number n we understand what it means for a set to have n elements. Consequently we say that set S is *finite* when there exists an $n \in \mathbb{N}$ such that S has n elements. Set S is is *infinite* when there is no $n \in \mathbb{N}$ such that S has n elements, or equivalently, for every $n \in \mathbb{N}$, we can find more than n elements in S.

3.1 Definition. A *tree* is a set of *nodes*, partially ordered by a relation "ancestor of" with a unique least element, called the root, with the property that the ancestors of each node are well-ordered (that is, there is no infinite chain going *toward* the root).

The *size* of a tree is the number of nodes.

The 0th *level* of a tree T consists of the root; the $(n+1)$st level is the set of immediate successors of nodes at the nth level, if there are any. If there are no such nodes we say that the $(n+1)$st level is empty.

The *depth* of a tree T is the maximum level that is not empty, if there is such a maximum. If level n is non-empty for each natural number n, we say that T has infinite depth.

A *path* or *branch* is a (finite or infinite) sequence of node $\langle x_0, x_1, \ldots, x_i, \ldots \rangle$ such that

- x_0 is the root,
- for all i, x_{i+1} is a child of x_i, and
- if the sequence is finite, then the last node is a leaf.

Strictly speaking the above definition only covers *countable* trees. It is rare for uncountable trees to occur anywhere but in set theory research, so we will blissfully ignore uncountable trees.

There is alternative definition of (countable) tree that is more concrete and so sometimes useful. To motivate it, suppose you have a tree in the sense above. We can assign to each node an "address" which is a sequence of natural numbers by the inductive rule: the root has address $\langle\rangle$; if a node has address α and has $n+1$ children then they have the addresses obtained by extending α in the obvious way: $\alpha 0, \alpha 1, \ldots \alpha n$. So each tree generates a set of addresses. Observe that the address of a node at level l is a sequence of length l.

3.2 Check Your Reading. *Draw some trees and label the nodes with addressees.*

Having assigned addresses to tree nodes, we might as well just say that the tree *is* the set of addresses it generates. That will be the alternate definition of tree.

Now, not just any set of sequences will do to be considered as a set of addresses for a tree. They have to "hang together" in the sense we describe below in Definition 3.3.

Let's fix some notation. Let $\mathbb{N}$ be the set of natural numbers $\{0, 1, 2, \ldots\}$. Let $\mathbb{N}^*$ be the set of finite sequences of natural numbers. If α and β are elements of $\mathbb{N}^*$ say that α is a *prefix* of β, written $\alpha < \beta$, if β is obtained by adding some elements to the end of α.

3.3 Definition (Trees, version 2)**.** A *tree* T is a subset of $\mathbb{N}^*$ satisfying the following two properties

- If $\alpha < \beta$ and $\beta \in T$ then $\alpha \in T$, and
- If $\alpha k \in T$ then for every $k' \in \mathbb{N}$ with $k' < k$, $\alpha k' \in T$. (Here $k' < k$ is taken in the ordinary sense of natural-number ordering).

There's a theorem in the background here saying that indeed the two definitions are equivalent but we won't bother to be formal here. Still:

3.4 Check Your Reading. *Convince yourself that the second definition really does capture your intuitions about what a tree is.*

3.5 Check Your Reading. $\mathbb{N}^*$ *is a tree. Draw it.*

3.6 Definition (Branching)**.** Let T be a tree.

- If a given node x has at most n successors then we say that x is *n-branching*; if x has infinitely many successors we say x is *infinite branching*.
- If T satisfies

  for all nodes x there exists an n such that x is n-branching

  then we say that T is *finite branching*.
- If T satisfies

  there exists an n such that for all nodes x, x is n-branching

  then we say that T is *n-branching*.

Note that being n-branching for a particular n is a much stronger condition than being finite-branching. To be n-branching means that this same n has to simultaneously bound the numbers of children for *all* nodes. If T has infinitely many nodes there is no reason to think that such a bound must exist, even if each node x has its own bound. Another way to put this point is to observe that to say that T is finite-branching is merely to say that no single node has infinitely many children: this is not the same as saying some single number bounds all the branchings.

3.1 König's Lemma

Before reading this section you should do Problem 41 at the end of this section.

When trees are known to be finite-branching, life is very different! Each of the statements in Problem 41 is *true* under the hypothesis of finite-branching. It all comes down the following famous and important theorem:

3.7 Theorem (König's Lemma)**.** *Let T be a finite-branching tree. If T has infinitely many nodes then T has an infinite branch.*

Proof. We build our infinite branch $\langle x_0, x_1, \ldots, x_i, \ldots \rangle$ as follows. Start with the x_0 being the root. Now, there are finitely many immediate subtrees below the root. Since there are infinitely many nodes in T, there must be infinitely many nodes in at least one of these subtrees. Let x_1 be the root of any one of these infinite subtrees. Now since there are infinitely many nodes in the subtree rooted at x_1, and since x_1 has only finitely many immediate subtrees, there must be infinitely many nodes in

at least one of these subtrees. Choose x_2 to be the root of any one of these infinite subtrees. Continue in this way, maintaining the invariant that the current node x_i has infinitely many nodes below it, and choosing x_{i+1} to be some child of x_i which itself has infinitely many nodes below it.

We can maintain the invariant *precisely* because each node has finitely many children. ///

Note that in order for the above proof to go through we did *not* require that the tree be n-branching for any particular n. All that mattered was that we never found ourselves looking at a node with infinitely many children.

3.2 Application: multiset induction

A *multiset* is, informally, a set with repetitions allowed. This won't do as a proper definition of course, so formally we say that a multiset S over a set X is a function $S : X \to \mathbb{N}$. Intuitively, $S(x)$ is the number of copies of x in S. A multiset S is *finite* if there are only finitely many x such that $S(x) > 0$.

We will continue to speak and write informally, and for example refer to the multiset $\{3, 17, 17, 17, 0, 0, 17\}$. This is the same multiset as $\{0, 17, 17, 17, 17, 0, 3\}$, different from the multiset $\{0, 17, 3\}$, and is formally the function that maps $0 \mapsto 2$, $3 \mapsto 1$, $17 \mapsto 4$ and maps every other number to 0.

3.8 Definition. The *multiset game* is as follows. Start with a finite multiset S_0 of natural numbers. At stage t of the game we will have a multiset S_t and we may proceed to build S_{t+1} by (i) removing some occurrence of an element n, and (ii) replacing it by *any finite number* of occurrences of elements strictly less than n.

For example, we might have

$$\begin{aligned} S_0 &\equiv \{0, 1, 17, 100\} \\ &\Rightarrow \{0, 1, 16, 16, 16, 16, 16, 16, 100\} \\ &\Rightarrow \{0, 1, 16, 16, 16, 16, 16, 16, 99, 99, 99\} \\ &\Rightarrow \{\quad 1, 16, 16, 16, 16, 16, 16, 99, 99, 99\} \ldots \end{aligned}$$

3.9 Lemma. *The multiset game always terminates after a finite number of moves.*

Proof. We use König's Lemma. Associated with any play of the game we construct a tree T whose nodes (other than the root) are labeled with natural numbers. If S_0 is the multiset $\{a_1, \ldots, a_k\}$ then initially T has a root and k children labeled, respectively, $a_1, \ldots a_k$. At stage t of the game the leaves of tree T will coincide exactly with the occurrences of elements of S_t. And when element n is replaced in S_t by new elements $n_1, \ldots n_p$, we add $n_1, \ldots n_p$, as new leaves of T, children of node n.

Note that at each stage the non-leaf nodes of T are in one-to-one correspondence with the moves of the game so far. So it suffices to show that we can never build an infinite tree by playing the game. But any such tree is finite branching. And each branch in the tree finds its labels to be strictly decreasing as we move down from the root. So each branch must be finite. So König's Lemma immediately implies that T must be finite. ///

This Lemma has obvious generalizations to multisets over well-founded orders other than the natural numbers. It is a powerful tool for showing that certain processes must terminate. We will see an example when we argue that certain tableaux constructions must terminate.

3.3 Problems

Problem 37.

The set of all bit strings, *i.e.*, the set of all finite sequences of 0s and 1s is a tree. Draw it.

Problem 38.

A *binary* tree is a tree such that each node has either 0 or 2 children.

Suppose T is a non-empty finite binary tree. Find a formula relating the number of leaf nodes and the total number of nodes in a finite binary tree T. Prove your answer by induction based on the inductive definition above.

Problem 39.

Give a specific example of a tree that is finite-branching yet is not n-branching for any n.

It may help to see these two assertions written in a more formal notation.

$$T \text{ is } n\text{-branching:} \qquad \forall x,\ x \text{ is } n\text{-branching}$$

$$T \text{ is finite-branching:} \qquad \forall x\ \exists n,\ x \text{ is } n\text{-branching}$$

Problem 40.

Since the issue in Problem 39 is sometimes confusing when you first meet it, here is the same distinction in another setting.

Give an example of a simple loop, say in a C-program, whose number of iterations depend on an integer variable x, that (i) never goes into an infinite loop, but (ii) there is no single integer n that bounds the number of iterations uniformly across all values of x.

Problem 41.

Here are some problems designed to drive home the subtle distinctions about branching we've been exploring.

Note: I'm aware of the fact that certain pairs of statements below are *logically equivalent* in the sense that they are contrapositive of each other. Sometimes such pairs have different intuitive force, though, so I presented them both.

a) Find a counterexample to the assertion: If T has finite depth then it has finite size.

We might write this more formally as *If there exists a $d \in \mathbb{N}$ such that T has depth d then there exists an $n \in \mathbb{N}$ such that T has size n.*

b) Find a counterexample to the assertion: If every branch in T has finite length then T has finite depth.

We might write this more formally as *If for all branchs p there is an $l \in \mathbb{N}$ such that p has length l then there exists a $d \in \mathbb{N}$ such that all nodes are at level $\leq d$.*

c) Find a counterexample to the assertion: If every branch in T has finite length then T has finite size.

More formally: *If for all branchs p there is an $l \in \mathbb{N}$ such that p has length l then there exists a $d \in \mathbb{N}$ such that all nodes are at level $\leq d$.*

d) Find a counterexample to the assertion: If T has infinite size then it has infinite depth.

More formally: *If there is no $n \in \mathbb{N}$ such that T has size n, then there is no $d \in \mathbb{N}$ such that T has depth d.*

e) Find a counterexample to the assertion: If T has infinite depth then it has an infinite branch.

More formally: *If for every $n \in \mathbb{N}$ level n of T is non-empty then there is an infinite branch in T, i.e., a branch p such that for each $k \in Nat$ there are more then k nodes in p.*

f) Find a counterexample to the assertion: If T has infinite size then there is a branch through T with infinite length.

More formally: *If for every $n \in \mathbb{N}$ we can find more than n nodes in T then there is an infinite branch in T, i.e., a branch p such that for each $k \in Nat$ there are more then k nodes in p.*

g) Find a counterexample to the assertion: If each branch in T is finite then there is a maximum branch length in T.

More formally: *If for all branchs p there is an $l \in \mathbb{N}$ such that p has length l then there exists a single number $n \in \mathbb{N}$ such that all branchs have length $\leq n$.*

h) Find a counterexample to the assertion: If T has branchs of arbitrarily long length then it has an infinite branch.

More formally: *If for every $l \in \mathbb{N}$ there is a branch in T of length at least l then there is an infinite branch in T.*

i) Find a counterexample to the assertion: If each branch in T has finite length then there is a maximum branching factor.

More formally: *If for all branchs p there is an $l \in \mathbb{N}$ such that p has length l then there exists a $b \in \mathbb{N}$ such that every node has fewer than b children.*

j) Find a counterexample to the assertion: If there is no maximum branching factor in T then every branch is finite.

Problem 42.

Prove that for any multiset S over $\mathbb{N}$ with at least one positive entry, there are *arbitrarily long* plays of the game starting with S. That is, for any S, given any number k, there is a play of the game starting with S that makes at least k moves. (This is easy.)

Explain why this does not contradict Lemma 3.9.

Chapter 4

Induction: Defining and Proving

Induction is the crucial technique for defining functions over the natural numbers and also for proving things about natural numbers. Perhaps even more important for computer science is the fact that data types such as like lists and trees are defined inductively. Once such an "inductive data type" has been defined, induction can be used to prove things about the data. And finally we can use induction to define programs over that data: programs defined by induction are more usually called "recursive." But it's all the same idea.

We assume in these notes that you seen induction before (though you might not feel experienced with it). So, this section is organized as a sequence of problems.

4.1 Defining

Problem 43.

Give an inductive definition of the function $e(n) = 2^n$ for $n \geq 0$.

Problem 44.

Give an inductive definition of the function $fact(n) = n!$ for $n \geq 1$

Problem 45.

The the first few Fibonacci Numbers are:

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, ...$$

Each is the sum of the previous two.

Give an inductive definition of the function $fib(n)$ for $n \geq 0$

Problem 46.

Let D_n denote the number of ways to cover the squares of a $2 \times n$ board using plain dominos. Here, a $2 \times n$ board has 2 rows and n columns, and "domino" is a figure comprising 2 squares, which can be oriented either horizontally or vertically.

Check for yourself that $D_1 = 1$, $D_2 = 2$, and $D_3 = 3$. (For the last case, you could have the 3 dominoes all be horizontal, or the first row covered by a horizontal domino and the latter two by two vertical ones, or finally have the last row covered by a horizontal domino and the upper two by two vertical ones ... draw pictures!).

What's D_4? What's D_5? What's D_n in general, based on previous values on D_k for $k < n$?

Problem 47.

A *binary tree* is a rooted tree in which every node has 0 or 2 children. These are computer scientist trees, not graph-theorists trees, which is to say that "left and right matter." So there are *two* trees with 3 leaves, even though they are mirror-images of each other, and isomorphic as graphs.

Draw a few pictures of small binary trees. There's only one tree that has 1 node; and only one tree with 2 nodes. As we said, there are two trees with 3 nodes. How many can you make with 4 nodes?

Let $t(n)$ count the number of binary trees with n leaves. Thus $t(1) = 1$, $t(2) = 1$, $t(3) = 2$. Convince yourself that $t(4) = 5$. Now given an inductive definition of $t(n)$.

Problem 48.

Suppose we draw n lines in the plane, with no two of them parallel, and no three of them intersecting in the same point. Let $l(n)$ count the number of regions in the plane obtained when we do this. So $l(1) = 2$, $l(2) = 4$, $= 7$. Give an inductive definition of $l(n)$

Problem 49.

Given a list n items to be combined by a binary operator $\oplus$, let $b(n)$ be the number of different ways we can group them. For example $x \oplus y \oplus z$ can be grouped as $((x \oplus y) \oplus z)$ or as $(x \oplus (y \oplus z))$ so $b(3) = 2$.

Given an inductive definition of $b(n)$. You should start with "$b(2) = 1$ and $b(3) = 2$."

4.2 Proving

Problem 50.

Prove that $1 + 2 + 3 + \cdots + n = \frac{1}{2}n(n+1)$

Problem 51.

Prove that $2+4+6+\ldots 2n = n(n+1)$

Problem 52.

Prove that $fib(0)+fib(1)+\cdots+fib(n) = fib(n+2)-1$

Problem 53.

Prove that for all $n \geq 4$, $2^n < n!$.

Problem 54.

Refer to part 47 for the definition of *binary tree*.

For the small examples you drew there count the number of leaves and count the total number of nodes. You will be moved to make the conjecture: *A binary tree with k leaves has* $2k-1$ *nodes.* Prove that.

Problem 55.

Consider a square chessboard, with one square missing; let us call such a board "defective." A "trimino" is made up of 3 squares in an L-shape. We are interested in when a defective board can be tiled (completely covered without overlapping) by triminos.

Clearly a 2×2 board can. Clearly a 3×3 board cannot (there are only 8 squares to be covered, not a multiple of 3). Experiment with some defective 4 boards and convince yourself that they can all be covered. Now prove that fact. *Try very hard in your argument to make use of the fact that the* 4×4 *board (before removing a square) can broken into 4 boards of size* 2×2 *each . . .*

Now prove carefully by induction on n: *any* $2^n \times 2^n$ *defective chessboard can be tiled with L-shaped trominos.*

Hint: For the inductive case, with a board size of $2^{n+1} \times 2^{n+1}$, divide the board into four quadrants. Note that each quadrant has size 2^n. The tricky part is that the induction hypothesis only applies to *one* of them, the quadrant where the original missing square lives. Find a way to place a trimino so that you can then use the induction hypothesis to all quadrants and thus tile the whole board.

Problem 56.

Some time ago (like the 20th century) postage stamps existed in small denominations, such as 3 or 5 cents. Prove that any amount of postage greater than 4 cents can be made with a sufficiently large supply of 3 and 5 cent stamps.

Formally, prove that for all $n \geq 8$ the equation

$$3x + 5y = n$$

can be solved with non-negative x and y.

Problem 57.

Do you see that in problem 55 you proved, as a side-effect, that *for every n, 3 divides* $2^n \times 2^n - 1$? Go ahead and prove that arithmetic fact directly, inspired by your proof there (but don't actually quote the chessboard result).

Problem 58.

Suppose you can get fried chicken in buckets of size 4 or 7. Find a number n such that any chicken order of size at least n can be filled exactly.

Do this with some other choices of numbers instead of 4 and 7. Can you do it with, say, 4 and 6? Can you say something in general about which pairs of numbers will work?

Problem 59.

An example of needing a "strong" induction hypothesis. Try to prove the following statement by induction on n:

For every n, the sum 1 + 3 + 5 + 7 + . . . (2n-1) is a perfect square

You will fail.

Now try to prove the following *stronger* statement by induction on n:

For every n, the sum 1 + 3 + 5 + 7 + . . . (2n-1) is in fact n^2 *itself*

Note that this statement is indeed stronger then the original, since it implies the original, but says more. And: this time you will succeed in your proof, precisely because you have *a stronger induction hypothesis.*

Problem 60.

Another example of the power of a "strong" induction hypothesis:

Prove that *for* $n \geq 0$, $fib(3n+2)$ *is even.*

Hint. Prove the following stronger statement (by induction)

for $n \geq 0$, $fib(3n)$ *is odd and* $fib(3n+1)$ *is odd and* $fib(3n+2)$ *is even.*

Problem 61.

Let a and b be distinct alphabet symbols. Prove that there is no string x such that $ax = xb$.

Problem 62.

You might think that proof by induction is just making obvious things hard, in the sense that once you verify a statement for lots of cases, it is obviously true and shouldn't require a fussy proof.

But consider the quantity $n^2 - n + 41$. Suppose someone makes the claim that for every n, $n^2 - n + 41$ is a prime number.

Plug in $n = 1, 2, 3, 4...$ until you get tired. Notice that each of these results is a prime number. In fact for every n up to $n = 40$ you will get a prime. But what happens for $n = 41$?

Moral: verifying finitely many cases prove nothing!

Chapter 5

Strings and Languages

5.1 Strings

5.1 Definition. An **alphabet** is any finite set. We typically use Σ (the Greek capital letter "Sigma") to denote an alphabet, and often use "symbol" to denote an element of an alphabet.

A **string** over alphabet Σ is a finite sequence of symbols from Σ. The set of strings over alphabet Σ is denoted Σ^*. Sometimes strings are called "words."

Strings vs Words Many authors use the term "word" instead of "string." Don't let that bother you, they mean the same thing.

One special string arises constantly: the empty string

5.2 Notation. The unique string with length 0, the empty string, is denoted λ.

5.3 Examples. The binary alphabet Σ_2 is just $\{0, 1\}$. Here are some strings over Σ_2: 01, 111110, λ, 10101010101010101010.

A binary file can be viewed as a string over Σ_2.

The ASCII alphabet is the set of 128 keyboard symbols you are familiar with.

A text file can be viewed as a string over the ASCII alphabet.

5.4 Definition. Let x and y be strings. We say that x is a *prefix* of y, written $x \preceq y$, if there is a string z such that $xz = y$.

We say that x is a *substring* of y if there are strings z_1 and z_2 such that $z_1 x z_2 = y$.

5.5 Check Your Reading. *Explain why every string x is a prefix of itself. (That means: show how this fits the formal definition above. What's the z?)*

In the same way, explain why every string x is a substring of itself.

5.1.1 Operations on Strings

The main operation on strings is **concatenation.** This is just string-append in programming. Just as with multiplication on numbers we typically denote concatenation by juxtaposition: if x and y are strings then xy is their concatenation.

For example if x is the string *ground* and y is the string *hog* then their concatenation xy is the string *groundhog*.

The analogy with multiplication is helpful but not exact.

- Concatenation is associative: $x(yz)$ is the same string as $(xy)z$. For this reason, we usually don't parenthesize concatenations.
- We use exponent notation: x^n stands for $xx\ldots x$, concatenating x with itself n times. (By convention, we take x^0 to be λ.)
- Concatenation has an identity element: $x\lambda$ is the same string as x, and is the same string as λx.
- But concatenation is not commutative: xy is usually not the same string as yx.

5.2 Languages

5.6 Definition. A **language** L over alphabet Σ is a set of strings over Σ.

A compact way to write "L is a language over alphabet Σ" is to write $L \subseteq \Sigma^*$.

You might wonder why we use that term "language." The reason is historical. The entire field we are studying has its origin not in computer science or even mathematics, but rather linguistics. The linguist Noam Chomsky, in the mid-1950s, revolutionized the study of human languages by using mathematical techniques. Computer scientists realized that they could adopt Chomsky's insights as tools for working with programming languages, especially writing compilers. By now there are many more applications in computer science (and elsewhere).

Note that *any* set of strings counts as a language.

One special language arises constantly: the empty language.

5.7 Notation. The unique language with no strings in it, *i.e.* the empty set, is a language, denoted $\emptyset$.

5.2.1 Empty string vs empty language?

Do not confuse the empty language $\emptyset$ with the empty string λ. They are not even the same type of object: the latter is a string that happens to have length 0, the former is a set of strings that happens to be the empty set. Also, do not confuse the language $\emptyset$ with the language $\{\emptyset\}$.

It may help your intuition to think of any set S as being like a box; it has the elements of S inside. In particular a language is like a box with some strings inside. One special case is the box with nothing in it: this is $\emptyset$. Another example is the box that has a single string inside, which happens to be the empty string λ. This is the language $\{\lambda\}$. That's a perfectly good language. But, since it is a language with one element in it, it is not the same as the empty language.

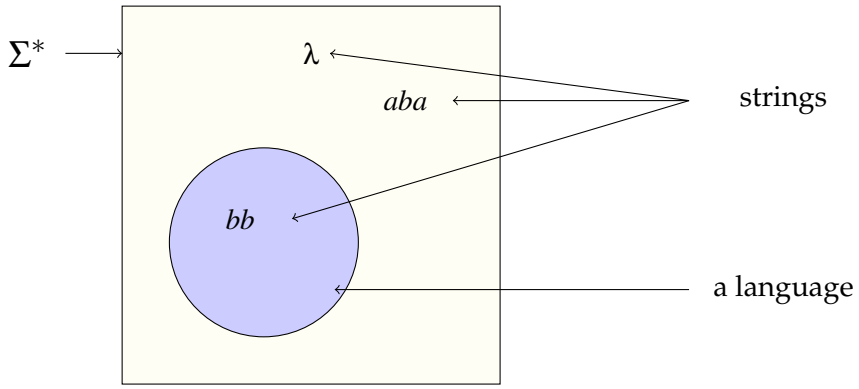


Figure 5.1: Everything this course is about!

5.3 Why Do We Care About Strings and Languages?

The notions of "string" and "language" as we have defined them sound somewhat narrow at first, not to mention dry. But no. Let's think about programs. Suppose we, for now, restrict our attention to input/output behavior of programs. Then

- All computing can be viewed as computation on strings. After all, a program reads input from *standard input*, a file,and it returns output on *standard output*, which is again a file. A file is a sequence of bits. And a sequence of bits is nothing more than a string over the alphabet $\Sigma = \{0, 1\}$.

So in other words, with respect to its input/output behvior, any program is nothing more than a device for changing bitstrings into bitstrings.

- A program itself, its source code, is a file. Thus a program is a string over $\Sigma =$ the ASCII alphabet. But there is a standard encoding of ACII symbols into bitstrings, so an ASCII string can be naturally thought of as a bitstring. That is to say, a program is just a bitstring as well.

  As will become clearer, the difference between ASCII and binary won't matter for the kinds of things we will be thinking about.

- A fundamentally important subclass of computing problems are "decision problems," in which we are presented with an input (a string) and want to answer yes or no. When we do that we are defining a language, namely the set of those strings for which the answer is "yes".

As will become clearer as we go, language membership, in the above sense, is a fundamental problem in computer science, really, *the* fundmental problem.

Examples

Here are some naturally-occurring decision problems, presented first as yes-no questions and then as the naturally associated languages.

1. *Lexical Analysis*
   INPUT: an ASCII string x
   QUESTION: is x a legal identifier for Java?

   The corresponding language:

   $$\{x \mid x \text{ is a legal identifier for Java}\}$$

2. *Parsing*
   INPUT: an ASCII string x
   QUESTION: is x a legal Java program?

   The corresponding language:

   $$\{x \mid x \text{ is a syntactically legal Java program}\}$$

3. *A Simple Correctness Property*
   INPUT: an ASCII string x
   QUESTION: is x a Java program that prints "Hello World"?

   The corresponding language:

   $$\{x \mid x \text{ is a Java program that eventually prints "Hello World" }\}$$

4. *A Uniform Correctness Property*
   INPUT: an ASCII string x
   QUESTION: is x a Java program that terminates normally on all inputs?

   The corresponding language:

$$\{x \mid x \text{ is a Java program that terminates normally on all inputs}\}$$

Each of the sets above comprises a language, specifically, a subset of the set of strings over the ASCII alphabet.

More interestingly, note that the above decision problems are given in increasing order of intuitive complexity. One of the contributions of the material in this course is to make this intuitive idea precise. We will develop a taxonomy of problems that captures their inherent "complexity," not in the sense of time or space requirements, but in terms of the complexity of the kinds of *abstract machines* required to solve them.

As an aside we note that the above perspective does make some non-trivial assumptions. For example, we are not thinking about interactive programs, that receive input and generate output continuously through the course of a computation. Also, we are not considering hybrid systems like thermostats, or robots, that communicate with and act on the physical world. There is of course much to be said about inherent complexity of computation in these richer sense, but the theory here should be mastered first.

5.3.1 Extended Example: Languages and Policies

Strings are not necessarily sequences of symbols written on a page. One important application for strings and languages is the following.

- an alphabet symbol stands for an *action* of a system
- thus a string is a sequence of actions, a *run* of the system

Sometimes we want to identify a set of possible runs of the system as being the desirable ones, those that meet some specification. Since this is a set of strings, it's a language.

Sometimes we want to identify a set of possible runs of the system as being the undesirable ones, those that violate some goal. That's a language too. A simple example might be: the set of all strings where some particular action—-like entering an unsafe state—never happens.

Let's work through a slightly more elaborate example.

Detecting Shoplifting

Let's model a store. We have with some items on shelves, with RFID tags on them. We can detect when an item is removed from a shelf; we can detect when an item is paid for at the registers; we can detect when an item leaves the store, past the sensors.

If we have k items, let's make an alphabet with $2k$ symbols, two symbols per item:

$$p_1, s_1, p_2, s_2, \ldots, p_k, s_k$$

where p_1 is "item 1 is purchased"; s_1 is "item 1 leaves the store"; p_2 is "item 2 is purchased"; s_2 is "item 2 leaves the store"; etc.

Here is a string over this alphabet:

$$p_3\ s_3\ p_1\ p_2\ s_2\ s_1$$

It models the event stream, "Item 3 was purchased, then item 3 left the store, then item 1 was purchased, then item 2 was purchased, then item 2 left the store, then item 1 left the store."

So this string models a run of the system that we are content with, it's a "good run."

Here is another string over this alphabet

$$p_1\ s_1\ s_2\ p_3\ s_3$$

This string captures a "bad" sequence of actions. Item 2 left the store without being purchased.

So articulating a policy, like

> "no item can leave the store without being paid for"

is the same thing as designating a set of runs as being "good" or "bad". And the set of good runs is a just a *language*.

The moral of the story is:

> Defining a policy about runs of a system is the same thing as defining a language.

Other examples:

- The policy: "Only superuser can read file f"

  As a language:

  - the alphabet is the set of read/write access on files
  - a string is a sequence of such reads and writes
  - the language consisting of the set of all strings which do *not* contain an alphabet symbol corresponding to a non-super-user reading f is the language defined by the policy.

- The policy "All requested print jobs are eventually printed"

  As a language:

  - the alphabet is the set of events "job j queued" and "job j printed"
  - a string is a sequence of such print requests and print executions
  - the language defined by the policy is the set of strings x such that for every j, if the symbol "job j queued" ever appears in x then at some point later the symbol "job j printed" appears somewhere later in x.

Why is this a useful point of view?

1. Recognizing that a run obeys (or doesn't obey) a policy corresponds to recognizing whether a string is a member (or not a member) of a language. This entire course is largely about building algorithms for such question. Stay tuned.

2. Complex policies can be constructed by mathematical operations on languages. To say that two different policies must be enforced just corresponds to taking the intersection of the two languages; to say that the system is allowed more than strategy to be "good" corresponds to taking the union of some languages; and so forth. We explore this idea in several places later.

We'll return to the shoplifting example in Section 5.3.1.

5.4 Ordering Strings

Let Σ be an alphabet. Let us order Σ in some arbitrary way, as $\Sigma = \langle a_1, \ldots, a_n \rangle$. Once we do that, there is a natural way to order the set Σ^* of strings. Namely, the empty string comes first, followed by all the strings of length 1, then all the strings of length 2, and so on; with each group of length k we order strings lexicographically, that is, in the dictionary ordering derived from the order on Σ.

Formally:

5.8 Definition. (Lexicographic Order on Strings) If the alphabet Σ is totally ordered, the *lexicographic order* $\prec$ on Σ^* is defined by:

- if $|x| < |y|$ then $x \prec y$
- if $|x| = |y|$ then, let i be the first position where x and y differ, let c_x and c_y be the alphabet sybmbols in x and y at position i; set $x \prec y$ if c_x is less than c_y in the ordering on Σ.

For example, when Σ is $\{a,b,c\}$ with $a < b < c$, we get the following ordered enumeration of Σ^*

$$\lambda, a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc, aaa, aab, \ldots\ldots$$

5.4.1 Enumerating the Bitstrings

Ordering strings in the case when $\Sigma = \{0,1\}$, with $0 < 1$, works out particularly nicely. Your first instinct might be to simply identify Σ_2^* with the natural numbers by treating a bitstring as a natural number in binary notation. That doesn't quite work, because of the problem of leading zeros. But a small tweak fixes this problem.

When we list the bitstrings in lexicographic order (ie ordering first by length, then by comparing earliest difference) we get this

$$\lambda, 0, 1, 00, 01, 10, 11, 000, 001, \ldots$$

When we then associate these strings to naturals numbers we get this

$$0 \mapsto \lambda,\ 1 \mapsto 0,\ 2 \mapsto 1,\ 3 \mapsto 00,\ \ldots, etc$$

Now suppose you were asked for the bitstring in the 173rd place. Do you have to start writing the strings in lexicograpic order till you have written 173 of them? Similarly, what if you had to say where 001101001 was in the scheme? Do you have to start writing the strings in lexicographic order till you get to 001101001 ? No! It's a nice and somewhat amusing fact that coding and decoding under the above scheme can be done "arithmetically," as follows.

1. Given a number n; to find the bitstring: *write* $(n+1)$ *in binary; strip off the leading 1*
2. Given a bitstring x; to find the code number: *add a 1 to the beginning of* x*, calculate the number that the string* $1x$ *encodes in binary, then subtract 1 from the answer.*

5.5 Bit Strings Are Universal

For any given problem domain, it is convenient to choose an alphabet Σ that fits the problem at hand. If we are doing program analysis, so that the objects of study are program users have written, we might chose Σ to be ASCII, or UTF-8. If we doing graph algorithms, Σ might include symbols to represent nodes, edges, and so forth.

But when we want to do *general* reasoning about problems (whether they are decidable, or what their complexity is, etc) it is better to have more uniformity. That's pretty easy to achieve: we can take $\{0,1\}$ as a canonical alphabet choice.

You might think that there is a price to pay for using only Σ_2. But in fact Σ_2 is perfectly general, in the sense that any other finite alphabet can be encoded into it. Here's one way to do it.

Suppose $\Sigma = \{a_0, \ldots, a_{n-1}\}$ is any (finite of course) alphabet. Let $k = \lceil \log_2 n \rceil$, that is, k is the least integer such that $2^{k-1} \geq n$. Then we can represent each a_i as a length-k bit string. And then any string x over Σ can be encoded as a single bitsring, just by concatenating the codes for the symbols in x. Since all the symbol-encodings have the same length, it is easy to translate back from a bitstring encoding of a string to the original string. (If complexity of computations is being considered, it is worth noting that converting from some alphabet to Σ_2 involves only a linear increase in the length of strings.)

For example, if Σ were $\{a,b,c,d,e\}$ we could translate this via

$$a \mapsto 000, \quad b \mapsto 001, \quad c \mapsto 010, \quad d \mapsto 011, \quad e \mapsto 100$$

And the string $x = abdb$ over Σ could be represented as a string over $\{0,1\}$ as 000001011001

5.6 Introduction to Combinatorics on Strings

In this section we'll explore a small corner of the—surprisingly complex—world of string concatenation, focusing mainly on a specific idea, that of one strings being *conjugate* to another.

First we give a rudimentary lemma that gets used in proving more interesting things. Suppose we have any four strings x, x', y, y' and we know that $xy = x'y'$. What else can we say? If $|x| = |x'|$ then it's easy, we must have $x = x'$ and $y = y'$. If x and x' have different lengths that obviously won't hold. But we can dig out more information.

Before diving into the lemma, note that when we analyze $xy = x'y'$ there are at first glance two cases: when $|x| \geq |x'|$ and when $|x| < |x'|$. But we only need to prove our basic lemma about the first case. Because : once we analyze that, if we ever find

ourselves looking at an equation $xy = x'y'$ where the x is shorter than the x' we just turn it around in our minds and think about the equation $x'y' = xy$ and apply our basic lemma to *that*.

5.9 Lemma (The Basic String Lemma). *Suppose $xy = x'y'$, and assume $|x| \geq |x'|$.*

Then there is a string u with

- $x = x'u$ *and*
- $uy = y'$

Reading such a lemma can be mind-numbing. Draw pictures! In this case, see Figure 5.2.

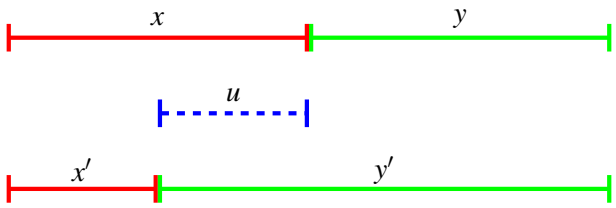


Figure 5.2: Illustrating the Basic String Lemma

But pictures are not proofs (though they guide you to write a proof). Let's write a real proof of Lemma 5.9.

Proof of Lemma 5.9. Let's first note that the in special case that the lengths of x and x' are the same it is clear that $x = x'$ and $y = y'$. And the result follows by taking u to be λ.

Next suppose $xy = x'y'$ and $|x| > |x'|$. Then clearly x' is a prefix of x; take u to be the string satisfying $x = x'u$. This u satisfies the lemma provided we show that $uy = y'$. But $xy = x'y'$ says that $x'uy = x'y'$, so we can cancel x' on the left to obtain $uy = y'$. ///

5.10 Check Your Reading. *Do Problem 72 now!*

5.6.1 Extended Example: Conjugate Strings

Recall the concatenation is not commutative: $uv \neq vu$ as a rule. But it is reasonable to wonder: is there *something* we can say about how uv is related to vu?

The answer is yes. Here is the key definition.

5.11 Definition (Conjugate Strings)**.** String x is *conjugate* to string x', written $x \sim x'$, if there exists a string w such that

$$xw = wx'$$

5.12 Examples.

- The string *abacd* is congruent to the string *bacda*. We can take w to be a. Note that *bacda* is obtained from *abacd* by a circular shift of 1 position.
- The string *abacd* is congruent to the string *cdaba*. We can take w to be *aba*. Note that *bacda* is obtained from *abacd* by a circular shift of 3 positions.

Here are some basic facts about the $\sim$ relation.

- If $x \sim x'$ then x and x' have the same length.
- Conjugacy is reflexive: Any string x is conjugate to itself: take w to be λ.
- Conjugacy is symmetric: If $x \sim y$ then $y \sim x$. Surprisingly, this is tricky to prove. (Don't be fooled by the notation "$\sim$" which tries to sneak symmetry into your subconscious!) We have to wait a little bit to prove it (Corollary 5.18).
- Conjugacy is transitive: If $x \sim y$ and $y \sim w$ then $x \sim w$. This isn't too hard to prove: we leave it as Problem 74.

Our goal is to show that for any two strings x and x', the following are equivalent:

1. x is conjugate to x'
2. x' is obtained from x by a circular permutation, that is shifting the symbols of x rightward and wrapping around
3. there exists strings u and v such that $x = uv$ and $x' = vu$

Actually the equivalence of the latter two is fairly easy to see even without doing any hard work.

5.13 Check Your Reading. *Convince yourself that to say $x = uv$ and $x' = vu$ is the same as saying that x' is obtained from x by a circular permutation.*

So we focus on proving that $x \sim x'$ if and only if there are strings u and v with $x = uv$ and $x' = vu$.

As a first step, notice that one direction of that isn't too hard. That is, two strings of the form uv and vu definitely *are* conjugate:

5.14 Lemma. *Suppose there are u and v such that $x = uv$ and $x' = vu$. Then $x \sim x'$.*

Proof. We need to show that there is a w such that $uvw = wvu$. Just take w to be u. ///

The harder thing is to show the other way around, that if $x \sim x'$ are conjugate then we can find u and v with $x = uv$ and $x' = vu$.

Here is a technical observation: we show that can always find a "short" w to witness conjugacy. This will be convenient at a key point later. It's a first use of our Basic Lemma, 5.9.

5.15 Lemma. *Suppose x and x' are conjugate. Then there exists some w such that $xw = wx'$ and $|w| \leq |x|$.*

Proof. What we will actually prove is the stronger claim that: the shortest string w such that $xw = wx'$ has the property that $|w| \leq |x|$.

If x is λ then so is x' and we may choose $w = \lambda$, which works.

So next let $x \neq \lambda$ and let w be a shortest string such that $xw = wx'$. For sake of contradiction suppose that $|w| > |x|$. Applying Lemma 5.9 to the equation $wx' = xw$, there is a string u such that

$$w = xu \quad \text{and} \quad ux' = w$$

This means that

$$xu = ux'$$

and since x is not λ, u is shorter than w. This contradicts our choice of w. ///

Now we are ready to prove our remaining goal.

5.16 Lemma. *If $x \sim x'$ then there exist strings u and v such that*

$$x = uv \quad \textit{and} \quad x' = vu.$$

Proof. We choose u to be the shortest string such that $xu = ux'$. By Lemma 5.15, that $|u| < |x|$. Now apply Lemma 5.9 to the equation $xu = ux'$. If we let v be the string produced there (the name u is already taken!) we get

- $x = uv$ and
- $vu = x'$

Which is what we want. ///

Summing up:

5.17 Theorem. *For any two strings x and x', the following are equivalent:*

1. *There exists a string w such that $xw = wx'$*
2. *There exists strings u and v such that $x = uv$ and $x' = vu$*

By the way we can—now!—establish that conjugacy is symmetric.

5.18 Corollary. *If $x \sim x'$ then $x' \sim x$.*

Proof. If $x \sim x'$ then for some u and v, $x = uv$ and $x' = vu$. But by swapping the roles of u and v, this establishes that $x' \sim x$! ///

5.6.2 An Instructive Lemma

Turning away from conjugacy, here is a fact that turns out to be useful in a few places later. It is worth studying the proof because it highlights a potentially tricky aspect of proof by induction.

5.19 Lemma. *For all natural numbers n, for all strings x and y,*

$$(xy)^n x = x(yx)^n$$

A picture (Figure 5.3) de-mystifies this. If there are $n+1$ copies of x and n copies of y, we can view the "extra" x as being at the start or at the end.

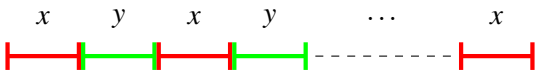


Figure 5.3: Illustrating Lemma 5.19

Proof of Lemma 5.19. By induction on n. When $n = 0$ the claim becomes simply $\lambda x = x\lambda$ which is clear.

Now let $n > 0$. Our induction hypothesis is

$$\textit{for all strings } x \textit{ and } y, \quad (xy)^{n-1}x = x(yx)^{n-1}$$

and we want to show

$$\textit{for all strings } x \textit{ and } y, \quad (xy)^n x = x(yx)^n$$

So choose x_0 and y_0. We want to show

$$(x_0y_0)(x_0y_0)^{n-1}x_0 = x_0(y_0x_0)^{n-1}(y_0x_0)$$

By canceling the leftmost x_0 then canceling the rightmost x_0—note that the parentheses in (x_0y_0) and (y_0x_0) don't matter—it suffices to show

$$y_0(x_0y_0)^{n-1} = (y_0x_0)^{n-1}y_0$$

Now this is an instance of the induction hypothesis so we are done. ///

There is an important subtlety in the above proof. The final claim $y_0(x_0y_0)^{n-1} = (y_0x_0)^n y_0$ is an instance of the induction hypothesis even though the roles of x and y seem to be reversed. The y_0 in our claim is the x in the induction hypothesis, and the x_0 in our claim is the y in the induction hypothesis. But this is ok, precisely because the induction hypothesis starts with "*for all x and y.*"

Once we said "choose x_0 and y_0 ..." we are then permitted to use the induction hypothesis with y_0 plugged in for x and x_0 plugged in for y.

In contrast, suppose we had stated the lemma this way:

Let x and y be strings. For all natural numbers n,

$$(xy)^n x = x(yx)^n$$

Then suppose we try to prove this by induction. We can do the calculations as above, but we get stuck because have chosen x and y first, and so our induction hypothesis is simply

$$(xy)^{n-1}x = x(yx)^{n-1}$$

Note the absence of the "*for all x and y*" This means we can't help ourselves to the cleverness of swapping the roles of x and y.

5.7 Operations on Languages

If A and B are languages, we can take the union $A \cup B$ the intersection $A \cap B$, and the complement $\overline{A}$ of these languages just because they are sets. But since A and B are sets of strings, there are two other operations that make sense, concatenation and Kleene-closure.

5.20 Definition. Suppose A and B are languages.

- $A \cup B$, $A \cap B$, and $\overline{A}$ (union, intersection, and complement) are defined in the standard way as for any sets.
- AB is the **concatenation** of A and B, the result of taking all possible concatenations of the strings from A then from B:

$$AB \stackrel{\text{def}}{=} \{xy \mid x \in A,\ y \in B\}$$

- A^* is the **Kleene-closure** or **Kleene-star** of A, the result of taking all possible concatenations of any finite number of strings from A:

$$\begin{aligned} A^* &\stackrel{\text{def}}{=} \{x_1 x_2 \ldots x_n \mid n \geq 0;\ \text{each } x_i \in A\} \\ &= \{\lambda\} \cup A \cup AA \cup AAA \cup \ldots \end{aligned}$$

The Kleene closure is named for Stephen Kleene, a pioneer of the subject.[1] Things to be careful about.

- Everyone uses the same name *concatenation* (and the same juxtaposition-notation) for two related, but different, things: an operation on two strings and an operation on two languages. This should not cause confusion assuming you know what type of objects are being talked about.
- Here is a very common source of confusion: in a concatenation xy there are no "markers" to say where x ends and y begins. So for example, the string aba is (i) the concatenation of a with ba, and also (ii) the concatenation of ab with a, and also (iii) the concatenation of aba with λ, and so forth.
- Suppose you are given two languages A and B, and you want to know whether some string z is in AB. In principle this means looking at *all* the different ways that z can be broken down as a concatenation xy, and then checking whether, for *any* of these ways, we have $x \in A$ and $y \in B$.
- Another overloaded notation: we write A^* for the Kleene-closure of A and we write Σ^* for the set of all strings. But this is a suggestive overloading. And anyway if you view Σ as being the language consisting of the set of symbols considered one-element strings, then the Σ^* notation does the right thing even when viewed as a Kleene-closure.

[1] Everyone pronounces his last name "cleany." That's weird, though, since Kleene himself pronounced his name "clay-nee".

- Whenever you see the "*" exponent, it means "zero or more" iterations of something. In particular we always include the empty string in any A^*.
- To poke further at the delicious possibility for confusions involving $\emptyset$ and λ, we will note that $\emptyset^*$ is precisely $\{\lambda\}$. Do you see why that is the case?

Examples

Here are some very simple examples, given just to exercise the definitions.

Let Σ be the alphabet $\{a,b,c\}$, and let A,B,C,D, and E be the following languages

$A = \{a\}$, the language whose only string is the length-1 string a
$B = \{b\}$, the language whose only string is the length-1 string b
$C = \{a,b\}$, the language consisting of the two strings a and b
$D =$ the set of all strings of odd length
$E =$ the set of all strings of even length
$\emptyset =$ the empty language.

Then

- $AB = \{ab\}$
- $AC = \{aa,ab\}$
- $CC = \{aa,ab,ba,bb\}$
- $A^* = \{\lambda,a,aa,aaa,\dots\}$, the infinite language consisting of all strings of as
- $A^* \cup B^* =$ the infinite language consisting of strings that are either all as or all bs
- $(A \cup B)^*$ = the set of all strings over $\{a,b\}$.
- $DD = E - \{\lambda\}$
- $D^* = \Sigma^*$
- $E^* = E$
- $(A\Sigma^*) =$ the set of strings starting with a
- $(A\Sigma^*) \cap E =$ the set of even-length strings starting with a
- $\Sigma^*(A\Sigma^*) =$ the set of strings in which an a appears

5.7.1 Algebraic Facts About Languages

You are already familiar with some routine facts about languages that arise just because they are sets. Some examples are the commutativity of $\cup$ and of $\cap$, the fact that $X \cap (Y \cup Z) = (X \cap Y) \cup (X \cap Z)$ and so forth.

But there are other facts that arise that only make sense because our languages are sets *of strings* with their inherent operations of concatenation and Kleene-star. For example, the fact that connection is associative: $(XY)Z = X(YZ)$.

5.21 Example. Here is a set of laws that hold between languages.

$$E \cup (F \cup G) = (E \cup F) \cup G \qquad E\{\lambda\} = \{\lambda\}E = E$$
$$E(FG) = (EF)G \qquad E\emptyset = \emptyset E = \emptyset$$
$$E \cup F = F \cup E \qquad E(F \cup G) = EF \cup EG$$
$$E \cup E = E \qquad (F \cup G)E = FE \cup GE$$
$$E \cup \emptyset = E \qquad E^* = \{\lambda\} \cup EE^* = \{\lambda\} \cup E^*E$$

and the rules

$F \cup EG \subseteq G$ implies $E^*F \subseteq G$

$F \cup GE \subseteq G$ implies $FE^* \subseteq G$

The above are all true equations about all languages. Most of them are easy to see just from the definitions of union, concatenation, etc. Others may seem obscure. But there is a reason that those particular equations and rules are presented. The reason is that *any* equation about languages involving unions, intersections, concatenations, and Kleene closure that is universally true follows—in a precise formal way—from this set of laws. A proof of that fact is beyond the scope of this course.

5.7.2 A Typical Proof About Languages

It may be that you are not experienced in writing proofs. Here's some advice. The only way to learn how to write proofs well is practice, guided by examples. One guideline that you will find very useful is: be very explicit, at each stage of a proof, about what your strategy is. That is, use lots of "here is what we now know, and here is what we are going to do next" statements. These are an aid to your reader, first of all. But also, such "scaffolding" will help *you* structure your proof and keep the logic correct.

Here we present, as an example, a proof of a simple set-theoretic statement. Pay attention to how much of the proof is devoted to those organizing assertions.

To prove:

$$A(B \cup C) = AB \cup AC$$

Proof.

We show that

1. $A(B \cup C) \subseteq AB \cup AC$ and
2. $AB \cup AC \subseteq A(B \cup C)$

Proof of 1:

Let w be an arbitrary element of $A(B \cup C)$; we want to show that $w \in AB \cup AC$. By definition of concatenation, w can be written as $w_1 w_2$ with $w_1 \in A$ and $w_2 \in (B \cup C)$. Thus $w_2 \in B$ or $w_2 \in C$. So there are two cases:

- if $w_2 \in B$: then $w_1 w_2 \in AB$. It follows that $w_1 w_2 \in AB \cup AC$
- if $w_2 \in C$: then $w_1 w_2 \in AC$. It follows that $w_1 w_2 \in AB \cup AC$

Since the result holds in each case, we are done.

Proof of 2:

For the second, let w be an arbitrary element of $AB \cup AC$; we want to show that $w \in A(B \cup C)$. There are two cases:

- if $w_2 \in AB$: then by definition of concatenation, w can be written as $w_1 w_2$ with $w_1 \in A$ and $w_2 \in B$. Then $w_2 \in (B \cup C)$. It follows that w, which is $w_1 w_2$, is in $A(B \cup C)$
- if $w_2 \in AC$: then by definition of concatenation, w can be written as $w_1 w_2$ with $w_1 \in A$ and $w_2 \in C$. Then $w_2 \in (B \cup C)$. It follows that w, which is $w_1 w_2$, is in $A(B \cup C)$.

Since the result holds in each case, we are done.

5.8 Algorithms About Languages

A key part of this course are the *algorithmic* questions about languages. Here is an example to give a hint of the flavor of this.

Membership in a Concatenation

Let A and B be languages. Suppose you have available an algorithm $\mathcal{D}_A$ which, given any string w, will answer "yes" or "no" as to whether $w \in A$. Further suppose you have an algorithm $\mathcal{D}_B$ will can test membership in B in the same way.

Then we can write an algorithm to solve the following problem.

> *Concatenation Testing*
>
> INPUT: a string w
>
> QUESTION: is $w \in AB$?

Here is pseudocode for a solution.

Algorithm 1: Concatenation Testing

on input w: let $n = |w|$; for each i with $0 \leq i \leq n$:
let x be the string defined by the first i symbols of w;
let y be the string defined by the last $n - i$ symbols of w;
call $\mathcal{D}_A$ on x; call $\mathcal{D}_B$ on y; if both of these return "yes", return "yes"
// else : next i ..
// If we get here no successful x and y were found: return "no"

Problem 84 asks you to do something slightly harder.

5.9 Cardinality and Formal Languages

In this section we apply the ideas of cardinality (*cf.* Chapter 2) in the setting of formal languages.

Recall the construction we did in Section 5.4. For any finite alphabet Σ we put Σ^* into a natural bijective correspondence with $\mathbb{N}$. This proves:

5.22 Theorem. *Let Σ be a finite alphabet. The set Σ^* is countable.*

Proof. Start with any total order on the alphabet Σ, and extend it to the lexicographic order on Σ^*. This define a surjection from $\mathbb{N}$ to Σ^*. ///

So, the set of all finite strings over a finite alphabet is countable. What if we keep the alphabet finite but allow the *strings* to be infinitely long? We addressed this in Theorem 2.3: there are uncountably many infinite strings.

This leaves one question unexplored: what about the set of finite strings over an infinite alphabet? Is this countable or uncountable?

First note that if the alphabet itself were uncountable then surely the set of strings over this alphabet in uncountable (after all there are uncountable many strings of length one!). So the interesting question is: suppose Σ is a countaby infinite alphabet. Is the set Σ^* of finite strings over Σ countable or uncountable?

To make things slightly more concrete, we might as well assume that Σ is just $\mathbb{N}$. So the question becomes: *Is the set of all finite strings of natural numbers countable or uncountable?* The answer is: countable. You are asked to prove this carefully in Problem 85.

Summarizing, we have that

- The set of all finite strings over a finite alphabet is countable.
- The set of all finite strings over a countably infinite alphabet is countable.
- The set of all infinite strings over a finite alphabet is uncountable.

The next result is really just a variation on Cantor's Theorem, but it is worth seeing the proof idea exercised again.

5.23 Theorem. *Let Σ be any non-empty alphabet. The set $\mathcal{L}$ of all languages over Σ is uncountable.*

Proof. We prove that any function $g : \mathbb{N} \to \mathcal{L}$ must fail to be surjective. Let an arbitrary $g : \mathbb{N} \to \mathcal{L}$ be given. Note that for any $i \in \mathbb{N}$, $g(i)$ is a language.

We make use of the fact that the set of all strings over Σ can be enumerated as

$$x_0, x_1, x_2, \ldots$$

This is a consequence of Theorem 5.22: formally we have a function from $\mathbb{N}$ surjective Σ^* and when we write x_i above we are referring to the string that is the image of i under this function.

Claim: the following language B is not in the image of g. B is defined by

$$x_i \in B \text{ if and only if } x_i \notin g(i)$$

To see that B is not in the image of g, we show that for each string n, B cannot be $g(n)$. But it is easy to check that the language B differs from the language $g(n)$: the string x_n will be in one of these languages but not the other. ///

Please make sure you see that the proof of Theorem 5.23 and the proof of Cantor's Theorem (Theorem 2.16) are really the same proof, just applied in a slightly different setting. Just keep in mind the association between characteristic functions and subsets.

5.10 Problems

Just About the Definitions

63. Diff

What's the difference between an alphabet and a language?

64. Ez

Let $A = \{aa, bb\}$ and $B = \{\lambda, b, ab\}$

1. List the strings in the language AB
2. List the strings in the language BB
3. Is abb in BA? How about bba? How about aa?
4. How many strings of length 6 are there in A^*?
5. List the strings in B^* of length 3 or less.
6. List the strings in A^*B^* of length 4 or less.

65. Emp

Is $\{\lambda\}$ a language? Is $\{\emptyset\}$ a language? Is $\emptyset$ a language?

66. Inf

a) Can a language be infinite?

b) Can an element of a language $A \subseteq \Sigma^*$ be infinite?

c) Does it make sense to talk about taking the union of two strings?

67. Nonsense

Suppose that: Σ is an alphabet, x and y are strings over Σ, A and B are languages over Σ.

Which of the following assertions make sense and which are nonsense? We mean "nonsense" in the syntactic sense: the assertions don't "type-check". For example "$17 \subseteq 23$" is nonsense because $\subseteq$ is a relation between sets, and 17 and 23 are not sets.

You need not justify your answers.

1. $x \cup y = y$
2. $xy \in (A \cup B)$
3. $A \subseteq B$
4. $A \in B$
5. $\emptyset \in A$
6. $\emptyset \subseteq A$
7. $\lambda \in A$
8. $\lambda \subseteq A$
9. $x \in A^*$
10. $x \subseteq A$
11. $x \subseteq A^*$
12. $x \subseteq \Sigma^*$
13. $A^* \subseteq \Sigma$
14. $x \in \Sigma^*$

Ordering Strings

68. Lex

Under the lexicographic order of the strings in $\{0,1\}^*$, what number string is 101? How about 0001?

What is the 99th string?

69. Ord

By definition, a *total ordering* $\leq$ is a relation that is

- reflexive: for all x, $x \leq x$;
- antisymmetric: for all x and y, $x \leq y$ and $y \leq x$ imply $x = y$;
- transitive: for all x, y and z, $x \leq y$ and $y \leq z$ imply $x \leq z$;
- complete: for all x and y, $x \leq y$ or $y \leq x$

The lexicographic order we defined is easily seen to be a total order.

But you might wonder why we invented the lexicographic order on strings rather than use familiar "dictionary order" on strings, which is the order you would, literally, arrange them in a dictionary.

Informally, in the dictionary we declare x to be less than y just if, in the first position where x and y differ, the x-entry is smaller, in the alphabet, than the corresponding y-entry. If there is no place where x and y differ but $x \neq y$ then one of x or y is a prefix of the other (make sure you see that), in which case we *then* compare lengths.

A bit more formally:

> Assume a total ordering on the alphabet Σ. The *dictionary order* $\leq_D$ on Σ^* is defined by:
>
> - $x \leq_D x$ for all x;
> - if $x \neq y$ and x is a prefix of y then $x \leq_D y$;

- otherwise let i be the first position where x and y differ, let c_x and c_y be the alphabet symbols in x and y at position i; set $x \leq_D y$ if c_x is less than c_y in the ordering on Σ.

This is a perfectly legal mathematical definition of an order, but it is probably not the order one wants.

Let's explore this order a little bit.

a) Is the dictionary ordering a total ordering? Prove this or give a counterexample to one of those properties.

b) Let's see why this is an inconvenient ordering to use. Suppose $\Sigma = \{0, 1\}$ where $0 < 1$.

1. Which comes first, 0 or 1?
2. Where does 1 come in the ordering?
3. Where does 01 come in the ordering?
4. Where does 001 come in the ordering?
5. Can you draw a "picture" of the ordering, indicating what things come before other things?

   If you find it hard to conceptualize a picture, do at least the following: *for the 15 0-1 strings of length 3 or less show how they are related to each other.*

70. NotLex

Show that for strings x and y, $x = y$ if and only if $x \not\prec y$ and $y \not\prec x$.

Combinatorics on strings

71. Ends

Let Σ be an alphabet with (at least) distinct symbols a and b. Prove that for no Σ-string x do we have $ax = xb$.

Hint. Suppose x is a smallest such string; get a contradiction.

72. BasicLem1

In the spirit of the remark just before the Basic Lemma 5.9, suppose $xy = x'y'$, with $|x| < |x'|$. Write down the conclusion that the basic lemma yields.

73. BasicLem2

Refer to Lemma 5.9. We said that the first claim, when $xy = x'y'$ and the lengths of x and x' are the same, is clear enough to not require proof.

But just for practice, give a careful proof by induction.

74. ConjTrans

Prove that $\sim$ is transitive: if $x \sim y$ and $y \sim w$ then $x \sim w$.

Hint. Just follow the definition: suppose there is a z_1 such that $xz_1 = z_1y$, and suppose there is a z_2 such that $yz_2 = z_2w$, then find a z_3 such that $xz = zw$.

75. ConjRev

Prove that if $x \sim y$ then $x^R \sim y^R$.

Hint. If you start with the original definition of $\sim$ then you will have to invoke the result that $\sim$ is symmetric.

76. ConjSquare

First, give an example of two different strings x and y, neither λ, such that $xy = yx$.

Next, prove that for any two strings x and y we have $xy = yx$ if and only if there is a string z such that $x^2y^2 = z^2$.

77. SquareFree

Say that a string w is "square free" if there are no non-λ strings x such that w contains xx as a substring. For example 210120 is square free, but 210102 is not square free.

Over a two-element alphabet such as $\Sigma = \{0, 1\}$ there are exactly seven square-free strings. What are they?

Over a three-element alphabet such as $\Sigma = \{0, 1, 2\}$ there are infinitely many square-free strings. Finding methods to generate them is quite hard. What's the longest one you can find?

Properties of Languages

78. LangProps

For each part decide if there can be languages A and B with given property. If so, give a specific example, if not, explain why not.

1. $AB = A$
2. $AB = \emptyset$
3. $AB = BA$
4. $A^* = A$ and $A \neq \Sigma^*$
5. $AA = A$.
6. $AA \subseteq A$ but $AA \neq A$.
7. $AA \not\subseteq A$
8. $A \subseteq AA$ but $AA \neq A$.
9. $A^* \subset A$, where $\subset$ means *proper* subset

79. LangSubset

Prove or disprove

1. forall A, B, C, $A(B \cup C) \subseteq AB \cup AC$
2. forall A, B, C, $A(B \cup C) \supseteq AB \cup AC$
3. forall A, B, C, $A(B \cap C) \subseteq AB \cap AC$
4. forall A, B, C, $A(B \cap C) \supseteq AB \cap AC$

For a proof, give a careful mathematical argument, based on the formal definitions of subset, concatenation, etc. For a disprof, give a specific, concrete, example of languages A, B, and C. *Hint.* Three of those are true and one is false.

80. LangEqns

Let A and B be arbitrary languages. Prove $(A \cup B)^* = A^*(BA^*)^*$

81. LangXYN

a) Prove that for all natural numbers n, for all languages X and Y,

$$(XY)^n X = X(YX)^n$$

Hint. Yes, this looks just like Lemma 5.19. The difference is that that was about *string* concatenation, whereas this problem is about *language* concatenation. So, look carefully at the proof of Lemma 5.19 and ask yourself: can I use that proof as a guide to the proof I want here? What has to change?

b) Prove that for all languages X and Y,

$$(XY)^*X = X(YX)^*$$

Hint. Please be clear that this is not the same statement as the previous part. But you should *use* the previous part in your answer.

82. Idempotent

A language L is said to be *idempotent* if $LL \subseteq L$. In this problem you will prove that (modulo a detail about λ) for any A, A^* is the smallest idempotent language containing A.

Specifically, prove the following hold for any language A.

1. The language A^* is idempotent.
2. If B is any idempotent language with $A \subseteq B$ and $\lambda \in B$, then $A^* \subseteq B$.

83. UsefulEquation

Prove:

$$\text{forall } A, B, C, \quad \text{if } B \cup AC \subseteq C \text{ then } A^*B \subseteq C$$

Hint.. To say that $x \in A^*B$ is to say that for some k, $x \in A^kB$. So another way to express the result is to say that if forall A, B, C, if $B \cup AC \subseteq C$, then we have forall k, $A^kB \subseteq C$. Now, prove that statement by induction on k.

Algorithms for Languages

84. StarTest

Let A be a language and suppose you have available an algorithm $\mathcal{D}_A$ which, given any string w, will answer "yes" or "no" as to whether $w \in A$.

Give pseudocode for an algorithm $\mathcal{E}$ to solve the following problem.

Kleene-star Testing

INPUT: a string w

QUESTION: is $w \in A^*$?

Cardinality of Languages

85. StarCountable

Let A be a countably infinite set. Prove that the set A^* of all finite strings over A is countable.

Hint. Use Problem 24, part b). Don't reinvent the wheel.

Part II

Regular Languages

Chapter 6

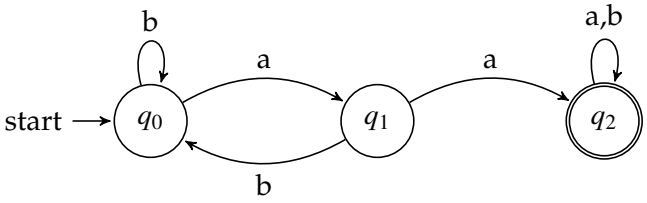
First Examples : Automata and Patterns

6.1 Finite Automata

Before we give careful definitions let's get an impression of what Finite Automata and Patterns are and why we care about them.

Text Search

6.1 Example. Automata can be used for text search. For example, the set of all strings over the alphabet $\{a, b\}$ that contain an occurrence of aa

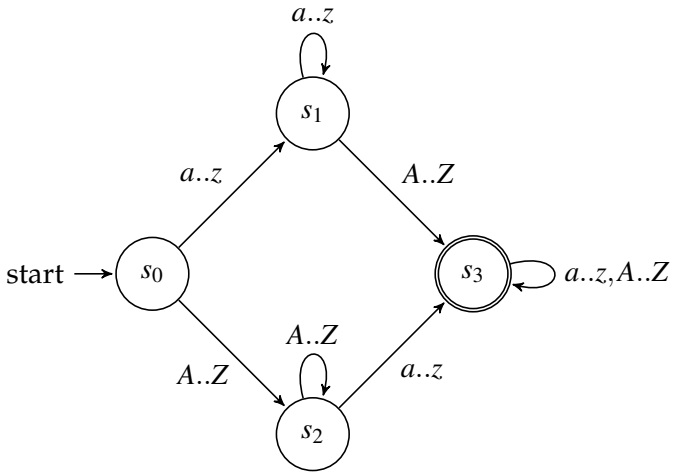


Here the arrow indicates that q_0 is the "start state" and the double circle indicates that q_2 is an "accepting state". (There doesn't have to be a *unique* accepting state.)

To see this as a machine, think of some sample strings made up of as and bs , trace through the automaton, and see where you finish. (For example on input *babaab* you will end in q_2.)

6.2 Example. Automata can recognize "acceptable" passwords.

As a baby example, let's just consider inputs of ASCII letters, and say we want just to insist that a password must have at least one lowercase letter and at least one uppercase letter



6.2 Patterns

Automata are for computing things or recognizing things. Sometimes we just want to **describe** what we want.

1. maybe we don't yet know how to compute it, and we want to have a specification for what to do, or
2. maybe we don't care how it is computed, we just want to tell someone what is being done

Patterns are expressions that *match* strings. We start with primitive patterns such as **a** and **b** that match only single-letter strings, and build up complex patterns using concatenation, union $E_1 + E_2$, and iteration $(E)^*$. These are the *regular expressions*. We may also allow ourselves operations for intersection and complement (and sometimes even more).

6.3 Examples. Fix the alphabet $\Sigma = \{a, b\}$.

- **a** matches the string a only.
- **banana** matches the string *banana* only.
- $\mathbf{a} + \mathbf{b}$ matches two strings: a and b.

- $\mathbf{a}^*$ matches any string of as (including λ)
- $\mathbf{a}^* + \mathbf{b}^*$ matches any string that is either all-a or all-b.
- $(\mathbf{a}+\mathbf{b})^*$ matches any string built using as and bs. So if the alphabet is $\{a,b\}$, that pattern matches any string in Σ^*

Patterns for Text Search

6.4 Example. Here is a pattern that matches the set of strings over the alphabet $\{a,b\}$ which contain an occurrence of aa. (We built a finite automaton recognizing this same language above in Example 6.1.)

$$(\mathbf{a}+\mathbf{b})^*\mathbf{a}\,\mathbf{a}(\mathbf{a}+\mathbf{b})^*$$

The $(\mathbf{a}+\mathbf{b})^*$ parts of the pattern matches anything (even the empty string); the **a a** part of the pattern matches only the string aa; so the whole pattern matches "anything, followed by aa, followed by anything". This is what we want.

6.5 Example. Here is a pattern that matches the set of strings over the alphabet $\{a,b\}$ which end in aba.

$$(\mathbf{a}+\mathbf{b})^*\mathbf{a}\,\mathbf{b}\,\mathbf{a}$$

The initial $(\mathbf{a}+\mathbf{b})^*$ part of the pattern matches anything; the **a b a** part of the pattern matches only the string aba; so the whole pattern matches "anything, followed by aba".

6.6 Examples. Patterns are good for describing acceptable passwords. Here are some very simple examples.

Let's add some convenient features to our pattern language:

- $[\mathbf{A}-\mathbf{Z}]$ matches any character between A and Z, and the doc
- $\cdot$ matches one occurrence of *any* character.

Must start with uppercase letter

$$[\mathbf{A}-\mathbf{Z}](\cdot)^*$$

Must contain and uppercase letter or a digit

$$(\cdot)^*([\mathbf{A}-\mathbf{Z}]+[\mathbf{0}-\mathbf{9}])(\cdot)^*$$

If we add intersection to our pattern language we can express *Must contain and uppercase letter and a digit*

$$((\cdot)^*[\mathbf{A}-\mathbf{Z}](\cdot)^*) \quad \cap \quad (\cdot)^*[\mathbf{0}-\mathbf{9}](\cdot)^*$$

Those examples suggested that we can think about text-procesing either from an automata perspective or from a pattern perspective.

Some examples to try:

1. Draw a finite automaton recognizing the language consisting of those strings x that do *not* contain aa.

   Build on Example 6.1. This is super-easy once you see the trick.

2. Write a pattern matching the language consisting of those strings x that do *not* contain aa.

   This, in contrast, is *not* "super easy"! We'll have more to say about this phenomenon later.

3. Build a finite automaton recognizing the set of strings over the alphabet $\{a, b\}$ which end in aba. (Compare Example 6.5.)

   This is a bit subtle. You need 8 states. We will return to this example in Example 6.12

4. It was pretty easy to build patterns reflecting various conditions on passwords. It's harder to build automata for this. We built a finite automaton in 6.2 for a very simple constraint: explore the idea of building automata for other constraints (such as in 6.6)

Moral: sometimes finite automaton are easy to build while patterns are harder; other times it's the other way around.

The Pattern Zoo

There are many, many different formalisms for patterns. We are not going to explore the differences, or learn how to be expert in any of them. What's interesting for us, will be the following fundamental facts.

- There is a core set of patterns, called *regular expressions*, that most—though not all—pattern languages can be compiled into. That is, most patterns are syntactic sugar for a regular expression.
- Sometimes that syntactic sugar is straightforward, just a textual substitution, but sometimes not!

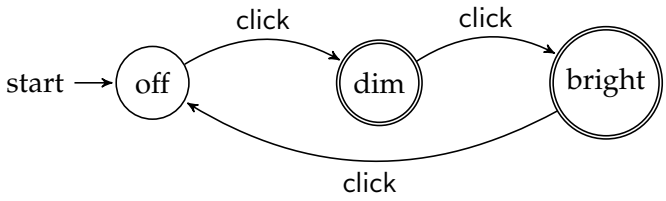
Can patterns express everything automata can do? Vice versa? Can we translate back-and-forth without having to be clever? The answer to all these questions is yes. Most of the first part of this course is devoted to showing why this is true.

6.3 Applications

6.3.1 Modeling Systems

Automata and patterns are useful in describing system behavior.

6.7 Example. A three-way light bulb. To specify the the behaviors of the light bulb that end up with the bulb being turned on (either dim or bright), we choose appropriate accepting states.



The system can be in one of three "states", and there is only one "action" (the click) that drives the system from one state to another.

6.8 Example. Here is a pattern that matches the behaviors of the light bulb that end up with the bulb being turned on (either dim or bright)

$$\mathsf{click}(\mathsf{click}\,\mathsf{click}\,\mathsf{click})^* + \mathsf{click}\,\mathsf{click}(\mathsf{click}\,\mathsf{click}\,\mathsf{click})^*$$

Test your intuition: do you think the following pattern will also work?

$$(\mathsf{click} + \mathsf{click}\,\mathsf{click})(\mathsf{click}\,\mathsf{click}\,\mathsf{click})^*$$

Shoplifting Again

Recall the shoplifting example in 5.3.1.

Using the notation there, here's a pattern $\mathbf{OK_1}$ that captures the sequences that don't have a store exit before purchase *for item 1*:

$$(\mathbf{p_2} + \mathbf{s_2})^* \mathbf{p_1} (\mathbf{p_2} + \mathbf{s_2} + \mathbf{p_1} + \mathbf{s_1})^*$$

We can easily write a similar pattern $\mathbf{OK_i}$ for *each* item i.

We can then write a single pattern that captures the policy that *no item* leaves the store without being paid for:

$$\mathbf{OK_1} \cap \mathbf{OK_2} \cap \ldots \cap \mathbf{OK_k}$$

Richer Policies If we want to be more sophisticated and track other suspicious event, like an item being paid for twice, we can tweak our pattern

$$(\mathbf{p_2} + \mathbf{s_2})^* \mathbf{p_1} (\mathbf{p_2} + \mathbf{s_2} + \mathbf{s_1})^*$$

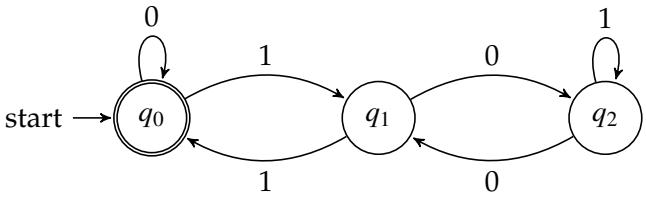
Even better, we can write patterns for lots of different kinds of behaviors we care about, and use intersections and unions to combine them.

Of course this is the time to ask: can we model policies for this system as automata? The answer is yes. We'll see how to build them—starting with the patterns—later.

6.3.2 Modular Arithmetic

Automata (and patterns) can count, in modular arithmetic.

6.9 Example. This automaton recognizes the set of 0-1 strings x such that the number that x codes in binary is divisible by 3. (As a special case, we consider the empty string λ to code the number 0).



Run this machine on a few inputs. For example

- on input 0 (coding 0) we end up in accepting state q_0, which is correct since 0 is divisible by 3
- on input 1 (coding 1) we end up in non-accepting state q_1, which is correct since 1 is not divisible by 3
- on input 10 (coding 2) we end up in non-accepting state q_2, which is correct since 2 is not divisible by 3
- on input 11 (coding 3) we end up in accepting state q_0, which is correct since 3 is divisible by 3
- on input 1011 (coding 11) we end up in non-accepting state q_2, which is correct since 11 is not divisible by 3
- on input 11011 (coding 27) we end up in accepting state q_0, which is correct since 27 is divisible by 3

Fine, but what is going one here? Why does this machine (seem to) work? We give a careful discussion in Example 7.11 later.

6.10 Example. Here is a pattern matching the bitstrings coding numbers divisible by 3. We give it here, honestly, just to show we can do it.

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

6.3.3 Program Designs

A nice approach to writing code is the following

1. make a finite automata to do the job you want (when you can)
2. prove that to be correct (we'll talk about how to do that later);
3. transform that, **in a mechanical way,** to actual program code.

That last step can be done without any human ingenuity (in other words, no opportunity for a human to mess it up). The result is a program that is *correct by construction*.

6.11 Example. Suppose you wanted to write a program to recognize whether a given bitstring encoded, in binary, a number divisible by 3.

Start with the finite automaton in Example 6.9. We can mechanically transform that into code. For simplicity here we'll show C-like pseudocode.

There are four procedures, main plus one for each DFA state.[1] Main starts by calling the "start state" `q0` Each procedure reads one character and jumps to the next procedure based on what is read. If a procedure reads EOF it returns ACCEPT or REJECT and the program halts

```
int main () { q0() }

int q0 () {
  read char c;
  case c==0 : q0();
  case c==1 : q1();
  case c== eof : return accept
  }
```

[1] This could be coded more simply (these are very lightweight procedures!) but in more complex kinds of automata, procedures might do more interesting things, so this is a worthwhile general construction principle.

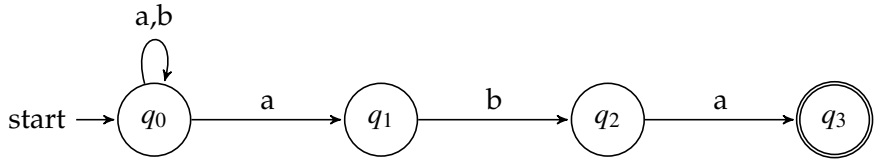
```
int q1 () {
read char c;
case c==0 : q2();
case c==1 : q0();
case c==eof : return reject
}

int q2 () {
read char c;
case c==0 : q1();
case c==1 : q2();
case c==eof : return reject
}
```

This is a pretty simple program. But Finite Automata can capture lots of complex kinds of computations (as we'll see). And I think you 'll agree that it is far easier to find potential bugs–or to argue that there are no bugs—in a finite automaton than in arbitrary program code. In fact we will prove this rigorously towards the end of the course.

6.4 Nondeterminism

We can expand our concept of finite automata to allow for *nondeterminism*.

6.12 Example. Here is a nondeterministic finite automaton recognizing the set of strings over the alphabet $\{a,b\}$ which end in *aba*.



In our previous machines, in every state, for every alphabet symbol, there is a unique transition to the next state.

For this machine: if it is in state q_0 and reads an a, it can either stay in q_0 or it can jump to q_1. Also, if it is in q_1 and reads an a, it blocks and fails. Similarly for q_2 reading a b, and for q_3 reading anything at all!

One of the fun things you will discover is how useful nondeterminism is! You may recall that you were challenged to write a finite automaton for this particular language earlier. Here we built an essentially trivial automaton that works, albeit nondeterministic. Which leads to the obvious questions:

- If we have a nondeterministic automaton, is there always an equivalent deterministic automaton?
- If so, can we build the deterministic one automatically from the nondeterministic one?

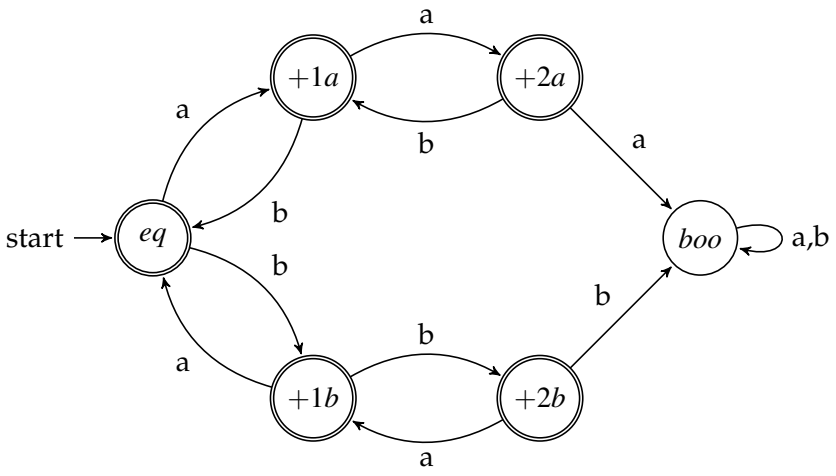
The answer to both of these questions is yes.

As a hint about this: see if you can make a deterministic automaton (one where all transitions are uniquely defined) for the strings ending in *aba*. I suspect you'll find it a bit tricky.

6.5 Limitations

Alas, life isn't as nice as we might like. Some languages cannot be recognized by finite automata. It will be instructive to start with a certain language that *can*:

6.13 Example. The set of strings x over $\{a, b\}$ such that in any prefix of x, the number of as and bs differ by at most 2.



Make sure you see how this machine succeeds in doing its job.

The names of the states are there to help your intuition. (But the names of states have no actual force, they are just like variable names in a program)

But now:

6.14 Example. Consider this language

> The set of strings x over $\{a,b\}$ such that the total number of as and bs in x differ by at most 2.

This is conceptually a simpler language than the one in Example 6.13: being in the language isn't based on running counts of as and bs, just one calculation that looks at the string as a whole.

But this seemingly simpler language *cannot* be recognized by a finite automaton. (Try it!)

We'll be able to *prove* claims like this once we have more techniques in hand.

That was a somewhat artificial example. Here is a scenario that arises in everyday computing.

6.15 Example. Matching parentheses

In a mathematical expression, left and right parentheses have to match up properly.

It's tricky/annoying to say precisely in English what the requirement actually is. Let's look at examples.

These are OK.

$$(\mathsf{stuff})$$
$$((\mathsf{stuff})+(\mathsf{stuff}))$$
$$((((\mathsf{stuff})))+(\mathsf{stuff})+(\mathsf{stuff}))$$

These are not OK

$$(\mathsf{stuff}$$
$$()\,\mathsf{stuff})+)\,\mathsf{stuff}))$$
$$(\mathsf{stuff}(\mathsf{stuff})))(\mathsf{stuff}))$$

Can we write a finite automaton to recognize when an expression has properly matching parentheses?

The answer is **NO**. But if we expand the notion of "finite automaton" to include a (stack) memory, the answer is yes. These are called pushdown automata.

What good is studying limited machines like finite automata, if they can't do some of the things we want to do? Understanding the answer to this is perhaps the most important theme of the course.

For now let's just say that we use simple machines rather than arbitrary programs whenever we can for the same reason that you don't use a chain saw to cut your toenails.

6.6 Looking Ahead

Here are some important themes about this part of the course.

1. There are algorithmic techniques for building finite automata, we don't have to be smart all the time.
2. There are languages that *cannot* be recognized by finite automata.
3. Every finite automaton M has a regular expression E that generates the same language that M recognizes.
4. Every regular expression E has a finite automaton M that recognizes the same language that E matches.
5. We can algorithmically translate back-and-forth between finite automata and regular expressions.

   An important activity we will focus on is building finite automaton *automatically*; patterns are a useful *input* language for such a tool.
6. There are algorithmic procedures for testing whether a given finite automaton does the job you expect it to do. (This is in contrast to arbitrary programs.)
7. For any finite automaton there is a unique best most efficient) finite automaton doing the same job.

6.7 Problems

The important thing in these problems is not to get the details exactly right but rather to develop intuitions about what can and can't be done with finite automaton and patterns.

86. MakeFA

Make finite automaton to model a vending machine.

There are infinite variations on this problem, obviously.

The alphabet symbols can correspond to various coin denominations being input to the machine. You might begin with only one alphabet symbol, denoting "25 cents inserted." The states of the machine can correspond to things like "start", "25 cents input so far", "50 cents input so far", ... Decide how much money your candy bar costs, say that is your accepting state.

Going on from there, think about how to model

- different coin denominations
- allowing "coin return"
- pushing a button to choose different candy bars (maybe with different costs)
- ...hours of fun ...

87. TweakPasswords

Tweak the passwords example by putting various constraints on strings.

88. PatternNoaa

Write a pattern over the alphabet $\{a, b\}$ that matches the set of strings that do not contain the substring aa. You can use any pattern operator except negation.

89. PatternText

Make automata for various text-searches, like "ends in b", "starts with qu", "has even length", "contains cat", and so forth.

Make patterns for these.

90. FAMod5

Define a finite automaton that accepts the set of all bitstrings x such that when x is decoded in binary notation as an integer n, n is equal to 1 mod 5 or equal to 3 mod 5. Be inspired by Example 6.9. As in that example, consider that the empty string codes the number $n = 0$.

91. FAToC

Work out the details of how to transform any finite automaton into a valid C (or choose another language) program, as suggested by Example 6.11.

Chapter 7

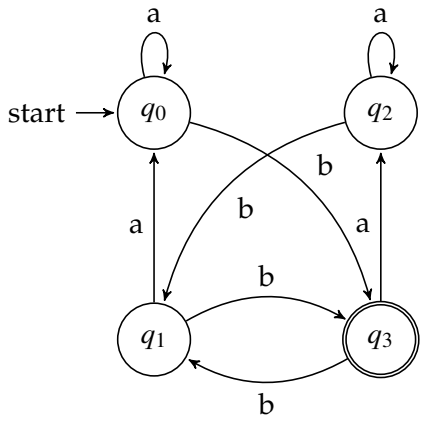
Deterministic Finite Automata

7.1 Prelude

Deterministic Finite Automata, or *DFAs*, are simple machines. They are not powerful enough to do everything that full computers can do, but they are richer than they may seem at first glance. The very fact that they are not as powerful as full computers is what makes them useful: we can build them and analyze them completely for correctness.

Before we dive into the precise mathematical definitions here is a peek at what they are like, informally.

This is a picture of a *DFA*. It has four *states*, one of which, q_0, is the *start* state, and one of which, q_3, is an *accepting* state. (In general there can be any number of accepting states.) It reads strings composed of *a*s and *bs* and bounces from one state to the other, as indicated by the labeled arcs. After the whole string has been read, if the *DFA* is in an accepting state, it *accepts* the input string.



Choose a few strings as input and trace how the machine processes them. Can you predict which strings will be accepted by this *DFA*? This *DFA* will reappear as Example 7.9.

7.2 Definitions

Since we are going to analyze *DFAs* in great detail we need to define everything carefully.

7.1 Definition (Deterministic Finite Automaton)**.** A **deterministic finite automaton** (*DFA*) is a 5-tuple $\langle \Sigma, Q, \delta, q_{st}, F \rangle$, where

- Σ is a finite set, called the *input alphabet*
- Q is a finite set, called the set of *states*
- $q_{st} \in Q$ is a distinguished state called the *start state*
- $F \subseteq Q$ is a distinguished set of states, called the *accepting states*
- $\delta : Q \times \Sigma \to Q$ is the *transition function*.

When $\delta(q,c) = p$ it is often more convenient to use the notation $q \xrightarrow{c} p$.

Given an input string $x \in \Sigma^*$, a *DFA* can embark on a computation, or *run*, as follows.

7.2 Definition. Let $M = \langle \Sigma, Q, \delta, s, F \rangle$ be a *DFA*, and let $x = a_0 a_2 \ldots, a_{n-1}$ be a string.

A **run** of M on x is a sequence of $n+1$ states $\langle q_0, \ldots, q_n \rangle$ such that

- $q_0 = q_{st}$ and
- for each $0 \leq i \leq n-1$, $\quad q_i \xrightarrow{a_i} q_{i+1}$ is a transition in δ.

Note that the concatenation of the symbols in the labelled transitions yields x.

It is suggestive to write

$$q_{st} \xrightarrow{a_0} q_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} q_n \qquad \text{or} \qquad q_{st} \xrightarrow{x} q_n$$

A run is **accepting** if $q_n \in F$.

We say that M **accepts** x if the run of M on x is accepting.

It is easy to see that for a given *DFA* M and word x there exactly one run of M on x. From the definition it follows that for any *DFA*, a run on the empty string λ is simply the length-1 sequence $\langle q_{st} \rangle$.

7.3 Definition. Let M be a *DFA*. The **language** $L(M)$ *accepted by* M is the set of strings accepted by M:

$$L(M) \stackrel{\text{def}}{=} \{x \in \Sigma^* \mid \text{ the run of } M \text{ on } x \text{ is accepting.}\}$$

Sometimes we say: "the language *recognized by* M" to mean the same thing as "the language accepted by M"

Sometimes we will want to consider the behavior of a *DFA* beginning at a state q that is not the actual start-state of the automaton. It is natural to use the same notation

$$q \xrightarrow{a_1} q_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} q_n \qquad \text{or} \qquad q \xrightarrow{x} \!\!\!\!\rightarrow\, q_n$$

in such a case. However we will only use the term "run" for a sequence that begins at the start state.

The $\hat{\delta}$ function

We defined "run" of a *DFA* M on a string x as a certain sequence of states. It is sometimes convenient to have notation that refers directly to the state that M eventually ends up in after processing x (that is, without having to refer to all the states in between).

The *DFA* M has as part of its definition a δ function, saying what happens when M reads a single symbol in a state q. Here we define the function $\hat{\delta}$, the iteration of δ, that says what happens when M reads a *string* in a state q.

7.4 Definition (The $\hat{\delta}$ function)**.** Let $M = \langle \Sigma, Q, \delta, S, F \rangle$ be a *DFA*. The function $\hat{\delta} : Q \times \Sigma^* \to Q$ is defined by recursion:

$$\begin{aligned} \hat{\delta}(q, \lambda) &= q \\ \hat{\delta}(q, xa) &= \delta(\hat{\delta}(q, x), a) \qquad\qquad \text{for } x \in \Sigma^*,\ a \in \Sigma \end{aligned}$$

What's the connection between speaking of runs and speaking of the $\hat{\delta}$ function? Just this

$$q \xrightarrow{w} \!\!\!\!\rightarrow\, r \qquad \text{if and only if } \hat{\delta}(q, w) = r$$

As a matter of fact, here is a little picture of the recursive call in $\hat{\delta}(q, xa)$, it will help

your intuition about $\hat{\delta}$

$$q \xrightarrow{x}\!\!\!\!\rightarrow \hat{\delta}(q,x) \xrightarrow{a} \delta(\hat{\delta}(q,x),a)$$

The truth is that the $\hat{\delta}$ notation is more convenient for *DFAs* but the "runs" notation is more convenient for *NFAs*, coming up shortly.

7.3 Regular Languages

7.5 Definition. A language $A \subseteq \Sigma^*$ is **regular** if there is a *DFA* that accepts it, that is, a *DFA* M such that $A = L(M)$.

Not all languages are regular, see Section 7.5.

7.6 Example. The set $A \stackrel{\text{def}}{=} \{x \in \{a,b\}^* \mid \text{ the length } |x| \text{ of } x \text{ is even}\}$ is regular.

Here is a *DFA* that accepts A.

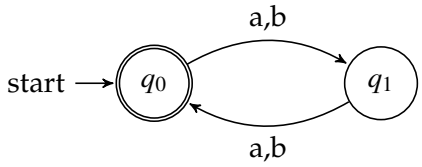
- Σ is (of course) $\{a,b\}$
- Q is $\{q_0,q_1\}$
- the start state q_{st} is q_0
- the set of accepting states is $\{q_0\}$
- the transition function δ is

$$\{((q_0,a),q_1), \quad ((q_0,b),q_1), \quad ((q_1,a),q_0), \quad ((q_1,b),q_0)\}$$

By the way this example shows that it is perfectly ok for the start state to be accepting.

Pictures of *DFAs*

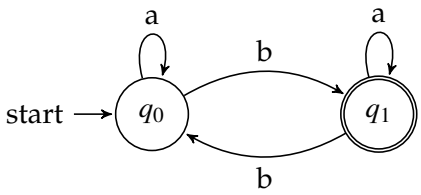
For small *DFAs*, we can capture the same information in a picture. Here is a picture of the *DFA* in Example 7.6 above:



We have introduced a shorthand here: in pictures we can capture more than one arc between the same states by label a single arc with more than one symbol.

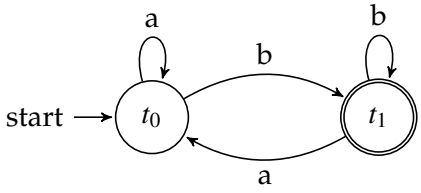
7.4 More Examples

7.7 Example. The set $B \stackrel{\text{def}}{=} \{x \mid x \text{ has an odd number of } b\text{s}\}$ is regular; here is a picture of a *DFA* that accepts B, as a picture and as a table.



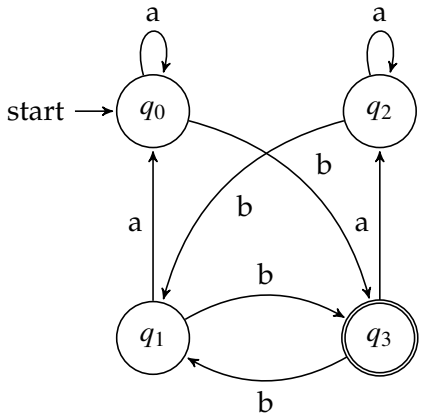
		a	b
start	q_0	q_0	q_1
accepting	q_1	q_1	q_0

7.8 Example. The set $C \stackrel{\text{def}}{=} \{x \mid x \text{ ends with a } b\}$ is regular; here is a *DFA* that accepts C.



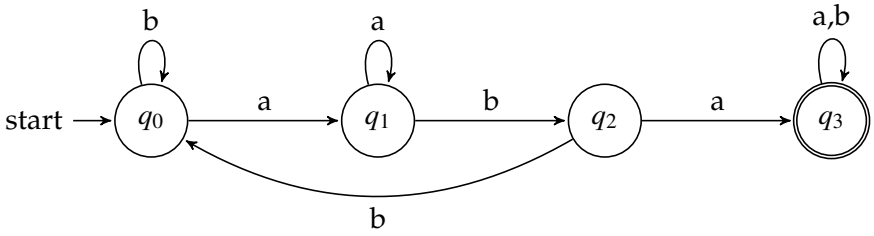
		a	b
start	t_0	t_0	t_1
accepting	t_1	t_0	t_1

7.9 Example. The set $D \stackrel{\text{def}}{=} \{x \mid x \text{ has an odd number of } b\text{s and ends with a } b\}$ is regular; here is a *DFA* that accepts D, as a picture.



This is not too difficult an example; still you might wonder if there is a way to build *DFAs* that is somewhat systematic, relying less on inspiration. There are indeed some tricks, for example we will revisit this language in Example 8.5 later.

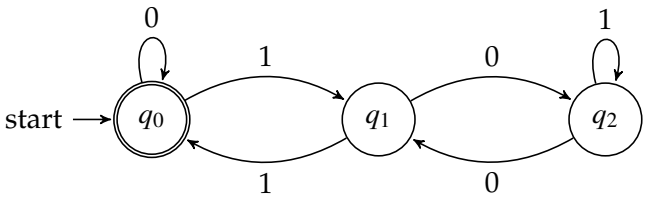
7.10 Example. The set E of strings over the alphabet $\{a, b\}$ which contain aba as a (contiguous) substring is regular; here is a *DFA* recognizing E.



7.11 Example. Let's go back to Example 6.9, counting in modular arithmetic. The set

$$F \stackrel{\text{def}}{=} \{x \mid \text{ the number that } x \text{ codes in binary is divisible by 3 } \}$$

is regular; here is a *DFA* M that accepts F.



To say that this works is to say that

> for every bit string x, if x codes a number that is equal to 0 mod 3 then the run of M on x will end in state q_0.

How do we know that this does its job correctly?

The trick is *state invariants.*

We claim that for any bitstring x,

- the *FA* carries x to state q_0 if the value of x is 0 mod 3;
- the *FA* carries x to state q_1 if the value of x is 1 mod 3;
- the *FA* carries x to state q_2 if the value of x is 2 mod 3.

If we have these claims, then the main claim (the first in the list) gives us the correctness result.

The cool thing is that proving these all at once, by induction over x, is easier than just proving the first one.

Convince yourself that the claim is true by choosing a few bitstrings and simulating the *DFA* on them *slowly* and with attention to the claim, just as we did with 101.

For instance for the string $x = 101$ we start in state q_0; then we read the leftmost 1, which sends us to state q_1; then we read the 0, which has the effect of doubling the value of what is read so far, from 1 to 2, and indeed we transition to state q_2; then we read the final 1, so what we do to the value of the string is "double-and-add-one", so our value of 2 is now 5, but (and here is the mod-counting aspect) 5 is the same as 2 so we transition to (stay at) state q_2. Now we are finished processing $x = 101$ and we end up in state q_2. Since the value of 101 is 5 which is 2 mod 3, this is what we hoped.

It is not hard to turn the reasoning in the previous into a rigorous proof by induction (over the length of the input string x). We'll skip that here.

Textual Representations of *DFAs*

Pictures are great to read but they are very annoying to have to generate in a document. For example, if you have to describe a *DFA* in a homework problem or on an exam, and you want to just use ASCII, you are stuck.[1]

So here is another way to write down *DFAs*. It's a a notation that we will investigate quite thoroughly and generally later in these notes: regular grammars. We are not going to define anything formally here, treat this for now as nothing more than a shorthand.

Example 7.6 could be presented like this:

$$
\begin{aligned}
& q_0 \text{ is the start} \\
q_0 &\to a\, q_1 \\
q_0 &\to b\, q_1 \\
q_0 &\to \lambda \\
q_1 &\to a\, q_0 \\
q_1 &\to b\, q_0
\end{aligned}
$$

We say what the start state is; then there is an entry in the notation for each (state, symbol) pair in the δ function. We have to say what the accepting states are: the convention is to have an entry leading to λ from each accepting state (this seems a weird convention now but it will make sense later).

We can streamline this notation by—optionally—collecting the entries for a given state all on one line, separated by a bar. So a more compact notation for Example 7.6 is:

$$
\begin{aligned}
& q_0 \text{ is the start} \\
q_0 &\to a\, q_1 \mid b\, q_1 \mid \lambda \\
q_1 &\to a\, q_0 \mid b\, q_0
\end{aligned}
$$

[1] Unless you are really good at ASCII art a maximally geeky talent.

Example 7.9 could be presented like this:

$$\begin{aligned} q_0 &\to a\, q_o \mid b\, q_3 \\ q_1 &\to a\, q_0 \mid b\, q_3 \\ q_2 &\to a\, q_2 \mid b\, q_1 \\ q_3 &\to a\, q_2 \mid b\, q_1 \mid \lambda \end{aligned}$$

q_0 is the start

Don't make any mental fuss over this notation: we are just agreeing on a quirky way to convey a picture of a *DFA*, in text. (We'll make a mental fuss over it later.)

7.5 Not All Languages are Regular

We have to do some work before we can actually *prove* certain languages to be non-regular. But it will be helpful for your intuition have a sneak peek at some examples.

7.12 Example. Let Σ to be the alphabet $\{a,b\}$. None of the following languages is regular.

1. $Eq = \{a^n b^n \mid n \geq 0\}$
2. $Eq2 = \{x \mid x$ has an equal number of as and bs$\}$
3. $A = \{a^n \mid n$ is a perfect square$\}$
4. $B = \{a^n \mid n$ is a perfect cube$\}$
5. $C = \{a^n \mid n$ is a power of 2$\}$
6. $\{x \in \{0,1\}^* \mid x$ codes a prime number in binary$\}$
7. the set D of strings of as and bs whose length is a perfect square
8. $E = \{ww \mid w \in \Sigma^*\}$
9. $F = \{ww^R \mid w \in \Sigma^*\}$

If we wanted to presume to find a common theme in those examples, we might say, "*DFAs* can't count." But it's not *quite* that simple, since we have seen, in Example 6.9 that *DFAs* can count in *modular arithmetic*...

In Section 15 we will be able to give a complete answer to the question, "which languages are regular?"

7.6 Problems

92. MakeDFA

For each of the following languages, make an *DFA* recognizing it.

a) Over the alphabet $\Sigma = \{a, b\}$: $\emptyset$, the empty set.

b) Over the alphabet $\Sigma = \{a, b\}$: $\{\lambda\}$. This is the language consisting of one string, the empty string λ.

Caution. This is not the same as the previous part.

c) Over the alphabet $\Sigma = \{a, b\}$: the set of all strings of length at least 1 whose last symbol is a b.

d) Over the alphabet $\Sigma = \{a, b\}$: the set of all strings of length at least 2 whose next-to-last symbol is a b.

Hint. You need four states.

e) Over the alphabet $\Sigma = \{a, b\}$: $\{w \mid$ the third symbol from the end of w is a $b\}$.

Hint. You need eight states!.

f) Over the alphabet $\Sigma = \{a, b, c\}$: the language denoted by a^*b^*, that is, $\{a^n b^m \mid n, m \geq 0\}\}$

g) Over the alphabet $\Sigma = \{a, b, c\}$: the language denoted by $a^*b^*c^*$, that is, $\{a^n b^m c^p \mid n, m, p \geq 0\}\}$

h) Over the alphabet $\Sigma = \{a, b\}$: $\{w \mid w$ contains bb as a substring$\}$.

i) Over the alphabet $\Sigma = \{a, b\}$: $\{w \mid w$ does not contain bb as a substring$\}$.

j) Over the alphabet $\Sigma = \{a, b\}$: $\{w \mid w$ every odd position of w is the symbol b$\}$. (Not clear what to say about λ, is it? It's easiest if we agree that λ is in this language.)

k) Over the alphabet $\Sigma = \{a, b\}$:

$$E \stackrel{\text{def}}{=} \{x \mid \ x \text{ has an odd number of } b\text{s or ends with a } b\}$$

Hint: Look at the *DFA* in Example 7.9; work from there.

93. DFAMod3

a) Generalizing Example 6.9.

Define a *DFA* that accepts the set of all bitstrings x such that when x is decoded as an integer n, n is equal to 1 mod 5 or equal to 3 mod 5. As usual, the empty string codes the number $n = 0$.

b) Generalizing Example 6.9 in another direction.

For a given k let Σ_k denote the alphabet $\{0, 1, ..., k-1\}$. For given k and m let $L_{k,m}$ be the set of strings x over Σ_k that, when viewed as a representing an integer in base-k notation, code a multiple of m. Show that each $L_{k,m}$ is regular, by showing that there is a DFA M with $L(M) = L_{k,m}$. *Note:* your answer to this problem will not be a single *DFA*. Rather, you should give a little recipe for how to construct a *DFA* once you are challenged with a k.

94. DFACount

This problem might seem artificial. But it's important to emphasize that a *DFA* is a finite syntactic object with a concrete representation that we can write down. This point of view will be increasingly important as we go along in the course.

a) Let's explore : how many different *DFAs* are there with one state? There is already a subtlety here, do we care about the fact that two structurally-identical *DFAs* might differ only in the name we use for the state? If so there would be infinitely many different ones ...but surely that is not interesting. In a similar way, we don't want to make a distinction between *DFAs* that only differ in that one has input alphabet, say, $\{0, 1\}$ and the other has input alphabet $\{a, b\}$. So to avoid such trivialities let's be a bit more pedantic, and ask this:

> How many different *DFAs* are there with one state named q_0, over the input alphabet $\Sigma = \{a, b\}$? Draw them.

b) How many different *DFAs* are there with two states named q_0, q_1, over the input alphabet $\Sigma = \{a, b\}$? Do not assume that q_0 is the start state. If two machines are structurally different but accept the same language, count them as different.

c) How many different *DFAs* are there with n states named $q_0, q_1, \ldots q_{n-1}$, over an input alphabet $\Sigma = \{a_0, \ldots a_{k-1}\}$ of size k? Do not assume that q_0 is the start state. If two machines are structurally different but accept the same language, count them as different.

95. DeltaFirst

We defined the extended δ function this way:

$$\begin{aligned} \hat{\delta}(q,\lambda) &= q \\ \hat{\delta}(q,xa) &= \delta(\hat{\delta}(q,x),a) \qquad\qquad \text{for } x \in \Sigma^*,\ a \in \Sigma \end{aligned}$$

In the recursive call we break down the argument by stripping off the last symbol. Suppose we wanted a definition that strips off the *first* symbolx? Fill in the dots to get a definition that makes sense and gives the same function as our official $\hat{\delta}$.

$$\begin{aligned} \hat{\delta}(q,\lambda) &= q \\ \hat{\delta}(q,ax) &= \ldots \qquad\qquad \text{for } x \in \Sigma^*,\ a \in \Sigma \end{aligned}$$

(Be inspired by Definition 7.4.)

Also, draw a suggestive picture of $\hat{\delta}(q,ax)$, like the one just after Definition 7.4, that fits your definition. (You might want to draw the picture *first*, to guide your definition.)

96. DeltaConcat

Prove that for all states q and strings x and y,

$$\hat{\delta}(q,xy) = \hat{\delta}(\hat{\delta}(q,x),y).$$

Prove this by induction on $|y|$.

97. DeltaStuck1

Let M be a DFA and let q be a certain state of M. Suppose that for every alphabet symbol c we have

$$\delta(q,c) = q$$

Prove that for all input strings x we have

$$\hat{\delta}(q,x) = q$$

Hint: Use induction on the length of the string x; apply the definition of $\hat{\delta}$ from page 16.

98. DeltaStuck2

Let M be a DFA and let c be a certain alphabet symbol. Suppose that for every state q we have

$$\delta(q, c) = q$$

Prove that for each state q and natural number n we have

$$\hat{\delta}(q, c^n) = q$$

Hint: Use induction on n; apply the definition of $\hat{\delta}$ from page 16.

Now show that

$$\text{either } \{c\}^* \subseteq L(M) \quad \text{or} \quad \{c\}^* \cap L(M) = \emptyset$$

Chapter 8

Constructing *DFAs*

An important consideration in building *DFAs*, just as with building any kind of system, is developing tools for building them in a modular way, that is building complex *DFAs* from simpler ones.

8.1 The Complement Construction

Suppose A is regular. Can we say that the complement $\overline{A}$ is also regular?

That's a pretty "mathematical-sounding" question. An equivalent question that feels more "computational" is: suppose M is a *DFA*, can we build a *DFA* whose language is the complement of $L(M)$?

The answer is yes, and it is easy to establish.

8.1 Definition (Complement Construction)**.** Let $M = (\Sigma, Q, \delta, s, F)$ be a *DFA*. We build a new *DFA* M', called the **complement** of M, as follows: $M' = (\Sigma, Q, \delta, s, (Q - F))$.

Be careful not to read too much into the word "complement" here. A *DFA* is not a set and so M' is not the complement of M in the same sense that you know from set theory. We use the word "complement" as a name to suggest the purpose of the construction, to take the complement (in the traditional sense of set theory) of the *language* of M. Indeed:

8.2 Theorem. *When M' is constructed from M as in Definition 8.1, $L(M') = \overline{L(M)}$.*

Proof. It is obvious that for any $x \in \Sigma^*$ and any states p and q, $p \overset{x}{\longrightarrow\!\!\!\!\!\rightarrow} q$ in M if and only if $p \overset{x}{\longrightarrow\!\!\!\!\!\rightarrow} q$ in M'. So taking p to be the start state, we see that the run of M on x ends at the same state as the run of M' on x. Since q will be an accepting state in M' if and only if it is not an accepting state in M, the result follows. ///

So now have the result we seek.

8.3 Theorem. *If A is regular then $\overline{A}$ is regular.*

Proof. Directly from Theorem 8.2. ///

8.2 The Product Construction

The product construction shows how to build automata for the intersection and for the union of regular languages.

8.4 Definition. Let M_1 and M_2 be *DFAs* over the same input alphabet:

$$M_1 = (\Sigma, Q_1, \delta_1, s_1, F_1)$$
$$M_2 = (\Sigma, Q_2, \delta_2, s_2, F_2)$$

We build a new *DFA* P, called the **product** of M_1 and M_2, as follows.

$$P = (\Sigma, (Q_1 \times Q_2),\ \delta_P,\ (s_1, s_2), (F_1 \times F_2)),\ \text{where } \delta_P \text{ is given by}$$

for each $c \in \Sigma$, and states p_1, p_2, q_1, q_2,

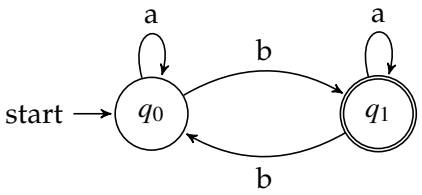
$$\begin{aligned} (p_1, p_2) \xrightarrow{c} (q_1, q_2) \quad & \text{precisely when} \\ p_1 \xrightarrow{c} q_1 \quad & \text{in } \delta_1 \text{ and} \\ p_2 \xrightarrow{c} q_2 \quad & \text{in } \delta_2. \end{aligned}$$

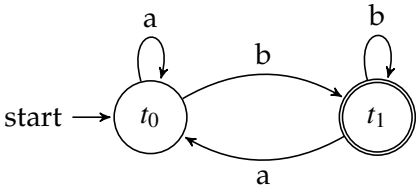
Caution. As with the complement construction, be careful not to read too much into the word "product" here. We use the word "product" as a name to suggest the way the construction works, by taking the cartesian product of various components of the machines. But machines are not sets, so we aren't formally taking a cartesian product of the machines themselves.[1]

This construction builds a *DFA* that accepts the *intersection* of the languages accepted by the original *DFAs*. Before proving that, let's see an example.

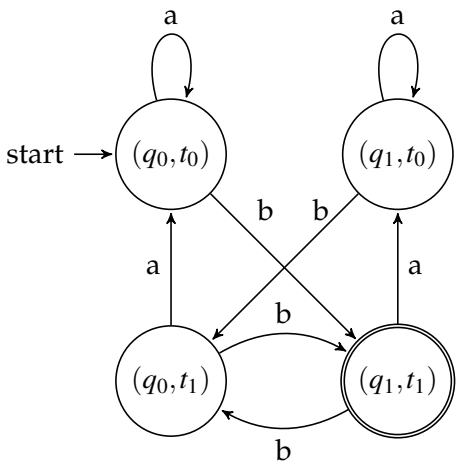
[1] Some authors even use the notation $M_1 \times M_2$ for our P above, but this is really asking for trouble.

8.5 Example. Recall Example 7.9 which showed a *DFA* for the language of strings that have an odd number of *b*s and end in *b*. Let M and N be the following *DFAs*. The first accepts strings with an odd number of *b*s. The second accepts strings that end in *b*.





The language we are interested in is precisely the intersection of the languages accepted by these two automata. Here is the result of applying the product construction on these automata; an automaton that accepts those strings that have an odd number of *b*s and end in *b*.



As you can check, this is the same *DFA* we saw in Example 7.9, except that here the states have funny names.

Once you understand this construction it should be intuitively clear that the following theorem is true. But we give a careful proof of correctness, as a model for how such proofs are done.

8.6 Theorem. *When P is constructed as in Definition 8.4, $L(P) = L(M_1) \cap L(M_2)$*

Proof. We prove the following claim: for any $x \in \Sigma^*$ and states p_1, p_2, q_1, q_2,

$$\begin{aligned}(p_1, p_2) &\xrightarrow{x}\!\!\!\!\rightarrow (q_1, q_2) \quad \text{if and only if} \\ p_1 &\xrightarrow{x}\!\!\!\!\rightarrow q_1 \text{ and} \\ p_2 &\xrightarrow{x}\!\!\!\!\rightarrow q_2.\end{aligned}$$

Notice that this looks quite a lot like the *definition* of δ we gave for P. The difference is that here we speak of *sequences* of transitions for a string x as opposed to *single* transitions for a single symbol c. So the claim is not true "by definition:" we have to prove it.

Also note that the claim is not a claim about *runs,* it is more general than that: we do not say that p_1 and p_2 are start states.

But after we do prove the claim, it will be true about runs in particular. And the rest of the proof will be easy.

So, to prove the claim. There are two directions to prove, the "if" and the "only if". We use induction on the length of x.

- *The base case:* When the length is 0, x is λ, so the claim is that

$$\begin{aligned}(p_1, p_2) &\xrightarrow{\lambda}\!\!\!\!\rightarrow (q_1, q_2) \quad \text{if and only if} \\ p_1 &\xrightarrow{\lambda}\!\!\!\!\rightarrow q_1 \text{ and} \\ p_2 &\xrightarrow{\lambda}\!\!\!\!\rightarrow q_2.\end{aligned}$$

  - *The "if" direction:* If $p_1 \xrightarrow{\lambda}\!\!\!\!\rightarrow q_1$ and $p_2 \xrightarrow{\lambda}\!\!\!\!\rightarrow q_2$ then $p_1 = q_1$ and $p_2 = q_2$. So certainly $(p_1, p_2) \xrightarrow{\lambda}\!\!\!\!\rightarrow (q_1, q_2)$.
  - *The "only if" direction:* Conversely, if $(p_1, p_2) \xrightarrow{\lambda}\!\!\!\!\rightarrow (q_1, q_2)$ then $(p_1, p_2) = (q_1, q_2)$, so $p_1 = q_1$ and $p_2 = q_2$, and thus $p_1 \xrightarrow{\lambda}\!\!\!\!\rightarrow q_1$ and $p_2 \xrightarrow{\lambda}\!\!\!\!\rightarrow q_2$.

- *The inductive step:* For the inductive step, we write x as cy with $c \in \Sigma$ and $y \in \Sigma^*$, and we may assume that the claim is true for y.

  - *The "if" direction:* If $p_1 \xrightarrow{cy}\!\!\!\!\rightarrow q_1$ and $p_2 \xrightarrow{cy}\!\!\!\!\rightarrow q_2$ then in M_1 we have some p_1' with $p_1 \xrightarrow{c} p_1' \xrightarrow{y}\!\!\!\!\rightarrow q_1$; and in M_2 we have some p_2' with $p_2 \xrightarrow{c} p_2' \xrightarrow{y}\!\!\!\!\rightarrow q_2$. So in P we have $(p_1, p_2) \xrightarrow{c} (p_1', p_2')$ [by definition of P] and $(p_1', p_2') \xrightarrow{y}\!\!\!\!\rightarrow (q_1, q_2)$ [by the induction hypothesis]. Glueing these together, we have $(p_1, p_2) \xrightarrow{cy}\!\!\!\!\rightarrow (q_1, q_2)$.

– *The "only if" direction:* Conversely, suppose $(p_1,p_2) \xrightarrow{cy}\!\!\!\!\rightarrow (q_1,q_2)$ By definition of P this means that for some pair (p_1',p_2') we have $(p_1,p_2) \xrightarrow{c} (p_1',p_2') \xrightarrow{y}\!\!\!\!\rightarrow (q_1,q_2)$. So we have $p_1 \xrightarrow{c} p_1'$ and $p_2 \xrightarrow{c} p_2'$ [by definition of P], and we have $p_1' \xrightarrow{y}\!\!\!\!\rightarrow p_1'$ and $p_2' \xrightarrow{y}\!\!\!\!\rightarrow p_2'$ [by the induction hypothesis]. Thus $p_1 \xrightarrow{x}\!\!\!\!\rightarrow q_1$ and $p_2 \xrightarrow{x}\!\!\!\!\rightarrow q_2$.

This finishes the proof of the claim.

Now that we have the claim we can finish the proof of the Theorem, by showing that for any $x \in \Sigma^*$, P accepts x if and only if each M_i accepts x.

P accepts x if and only if there is an accepting run of P on x, iff for some $(q_1,q_2) \in (F_1 \times F_2)$, we have $(s_1,s_2) \xrightarrow{x}\!\!\!\!\rightarrow (q_1,q_2)$. By the claim above this happens iff $s_1 \xrightarrow{x}\!\!\!\!\rightarrow q_1$ and $s_2 \xrightarrow{x}\!\!\!\!\rightarrow q_2$, which says that $x \in L(M_1)$ and $x \in L(M_2)$. ///

Make sure you can explain why it was important in the above proof that we did not state the claim only about *runs*, sequences that necessarily began at start states.

Product Construction for Union There is a closely related construction, which builds an *DFA* to accept the *union* of languages accepted by two given *DFAs*. Starting with M_1 and M_2 in the notation above, if we build M' as

$$M' = (\Sigma, (Q_1 \times Q_2),\ \delta,\ (s_1,s_2), ((F_1 \times Q_2) \cup (Q_1 \times F_2)) \text{ with } \delta \text{ as above}$$

so that the set of accepting states is the set of pairs (q_1,q_2) with at least one of the q_i accepting in its original machine, then we have

$$L(M') = L(M_1) \cup L(M_2)$$

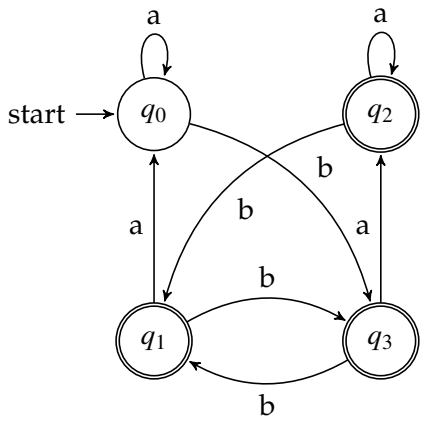
8.7 Check Your Reading. *Prove that* $L(M') = L(M_1) \cup L(M_2)$.

Hint. *This is easier than you might think at first. Look at the proof of Theorem 8.6. You only have to change a word or two!*

8.8 Example. Recall the language

$$E = \{x \mid \ x \text{ has an odd number of } b\text{s or ends in a } b\ \}$$

from Problem k). A *DFA* for E can be built by the product construction, just as in Example 7.9 but with different accepting states, and as expected, we get exactly the *DFA* that the hint in Problem k) was leading you to. It looks superficially different from a product-construction *DFA* just because we named the states without using ordered-pair notation.



8.3 Regular Closure: Intersection and Union

These two constructions prove the following theorem.

8.9 Theorem. *If languages A and B are regular then so are $A \cap B$ and $A \cup B$.*

Proof. Immediate from Theorem 8.6 and the observation in 8.7. ///

8.4 Other Operations

So far we have shown that we can construct *DFAs* motivated by the Boolean operations on language: complement, intersection, and union. It is natural to wonder about the operations of concatenation and Kleene star. That is we might ask:

- If M_1 and M_2 are *DFAs*, can we construct a *DFA* M such that $L(M)$ is the concatenation language $L(M_1)L(M_2)$?
- If M_1 is *DFA*, can we construct a *DFA* M such that $L(M)$ is the language $L(M_1)^*$?

The answer to each of these questions is yes. But the easiest way to do the construction involves an excursion into *nondeterministic* finite automata. Stay tuned.

Caution!

We have shown that the union of two regular languages is regular. By iterating that argument we can conclude that the union of any finite number of regular languages is regular. But we **cannot** conclude that the union of an infinite number of regular languages is regular. If that were true, then *every* language would be regular!

After all, any language $A = \{x_1, x_2, \dots\}$, regular or not, can be written as the union of singleton languages: $A = \{x_1\} \cup \{x_2\} \cup \dots$. And each of the singleton languages $\{x_i\}$ are certainly regular.

So be careful about that.

If you know about countable and uncountable sets: This is a little bit like the danger of generalizing from the fact that the cartesian product of finitely many countable sets is countable to the non-fact that the cartesian product of infinitely many countable sets is countable. The latter is certainly not the case, since the (uncountable) set of infinite bit strings is nothing more than the infinite product of $\{0,1\}$ with itself. Generalizing arguments about finite iterations of a process to infinite iterations is fraught with peril!

8.5 Problems

99. DFADiff

Describe an algorithm for the following problem

> *DFA Language Difference*
>
> INPUT: two *DFAs* M and N
>
> OUTPUT: a *DFA* D such that $L(D) = L(M) - L(N)$

Remember that $L(M) - L(N)$ means $L(M) \cap \overline{L(N)}$.

100. DFAUnion

Let M_1 and M_2 be *DFAs*. Here is another way to build a *DFA* accepting the union $L(M_1) \cup L(M_2)$. Motivated by DeMorgan's law, that

$$\text{for any sets } X \text{ and } Y: \; X \cup Y = \overline{\overline{X} \cap \overline{Y}}$$

we can do the following:

1. use the complement construction on M_1 and on M_2, obtaining M_1' and M_2';
2. use the product construction for intersection, obtaining a *DFA* M' with $L(M') = L(M_1') \cap L(M_2')$;
3. use the complement construction on M' to obtain M''

Verify for yourself that $L(M'') = L(M_1) \cup L(M_2)$. Now : what is the relationship between this M'' and the *DFA* we get by using the product construction for union as described in the text?

101. WhyReg

Explain why the following language L over the alphabet $\Sigma = \{a, b, c\}$ is regular.

> L = the set of all strings x such that : x starts with a **and** x has even length **and** furthermore either x has at least 17 cs **or** x does **not** have length divisible by 3.

Hint. I hope the silliness of this language definition is a clue that you should not give an explicit construction of a *DFA* as an answer!

102. DFAPuzzle

Let $\Sigma = \{0, 1\}$. Draw the obvious 2-state *DFA* M such that $L(M)$ is the set of words that end in 0. Draw the obvious 2-state *DFA* N such that $L(N)$ is the set of words that end in 1. Make the product construction on M and N, obtaining a *DFA* P whose language is $L(M) \cap L(N)$. Explain what is going on, in light of the fact that $L(M) \cap L(N) = \emptyset$.

Chapter 9

Nondeterministic Finite Automata

Here is a generalization of *DFAs*.

9.1 Definition (Nondeterministic Finite Automaton)**.** A **nondeterministic finite automaton** (*NFA*) is a 5-tuple $\langle \Sigma, Q, \delta, q_{st}, F \rangle$, where

- Σ is a finite set, called the *input alphabet*
- Q is a finite set, called the set of *states*
- $q_{st} \in Q$ is a state, called the *start state*
- $F \subseteq Q$ is a distinguished set of states, called the *accepting states*
- $\delta \subseteq Q \times \Sigma \times Q$ is the *transition relation*: an arbitrary set of triples (p, c, q). Such a tuple is more suggestively written $p \xrightarrow{c} q$.

So the only difference between *DFAs* and *NFAs* is that for *NFAs*, the δ is a transition *relation*, not necessarily a function. Note that since a function is a just a certain kind of relation, a *DFA* is automatically an *NFA*.

9.1 Runs of an *NFA*

The definition of *run* of an *NFA* looks on the surface just like the definition for a *DFA*.

9.2 Definition. Let $M = \langle \Sigma, Q, \delta, q_{st}, F \rangle$ be an *NFA*, and let $x = a_0 a_1 \ldots, a_{n-1}$ be a string.

A **run** of M on x is a sequence of $n+1$ states $\langle q_0, \ldots, q_n \rangle$ such that

- $q_0 = q_{st}$ and
- for each $0 \leq i \leq n-1$, $\quad q_i \xrightarrow{a_i} q_{i+1}$ is a transition in δ.

But note the following important differences between *NFAs* and *DFAs*.

- For an *NFA* N and a word x there may be more than one run of N on x.
- With an *NFA* N and a word x, it is possible that is possible that there are *no* runs of N on x. Here is how this can happen: As the machine tries to construct a run on x, it can find itself in a state q while reading a symbol a and there is simply no transition out of q on a, in other words, there is no state q' such that $(q, a, q) \in \delta$. In this case we say that the machine "blocks."

  We do *not* call such a sequence of states an accepting run on the input: the input has to be completely processed before we call it a run at all.

  Thus it can happen that every attempt to make a run blocks at some point, in which case there are no runs of N on x. (Of course it follows that there are no accepting runs on x.)

We still say that an *accepting run* of M on x is a run whose last state is an accepting state.

But what should we say about whether an *NFA* accepts a string? For *DFAs* is was obvious what to say: a *DFA* accepts a string x if **the** run on x is accepting. But since there can be more than one run of an *NFA* on a string, we have to be a bit more subtle in our notion of acceptance for an *NFA*.

9.3 Definition. Let M be an *NFA*. A string x is accepted by M if **there exists** an accepting run of M on x. The **language** $L(M)$ *accepted by* M is the set of strings accepted by M:

$$L(M) \stackrel{\text{def}}{=} \{x \in \Sigma^* \mid \text{ there exists an accepting run of } M \text{ on } x\}$$

Later in this section we will prove the following fact:

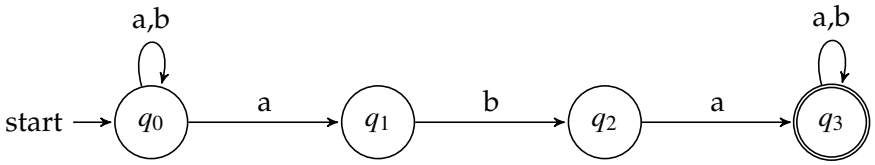
For any *NFA* N there is a *DFA* M such that $L(M) = L(N)$.

So why do we bother to define *NFAs*? Because they are more convenient to construct than *DFAs*. Read on.

By the way, it is reasonable to ask: why didn't we say that an *NFA* accepts x if *every* run on x is accepting? That wouldn't be crazy. But it turns out that the convenience of using *NFAs* arises because of this liberal interpretation of acceptance. And anyway, we wouldn't end up with anything new: see Problem 110.

9.2 Examples

9.4 Example. This *NFA* accepts the set of strings over the alphabet $\{a, b\}$ which contain *aba* as a (contiguous) substring.



This is good time to clarify the notion of a run. Suppose the input to the *NFA* is *aaba*. One run is:

$$q_0 \xrightarrow{a} q_0 \xrightarrow{a} q_0 \xrightarrow{b} q_0 \xrightarrow{a} q_0$$

This is not an accepting run. But the existence of such a non-accepting run does not tell us *anything* about whether *aaba* is in $L(N)$. Another attempt at a run is

$$q_0 \xrightarrow{a} q_1 \xrightarrow{???}$$

which blocks, since there no transition defined out of q_1 on a. This isn't a run (on *aaba*) at all!

Another run is:

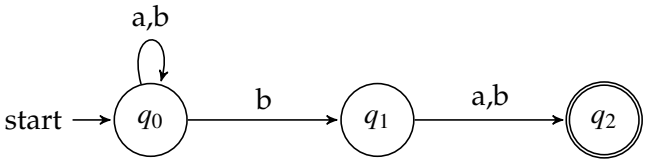
$$q_0 \xrightarrow{a} q_0 \xrightarrow{a} q_1 \xrightarrow{b} q_2 \xrightarrow{a} q_3$$

That's an accepting run. Since there *exists* an accepting run, we say that *aaba* is accepted by N, and so $aaba \in L(N)$.

Earlier we saw a *DFA* that accepts the same language. But the *NFA* captures more directly what the specification of the language is.

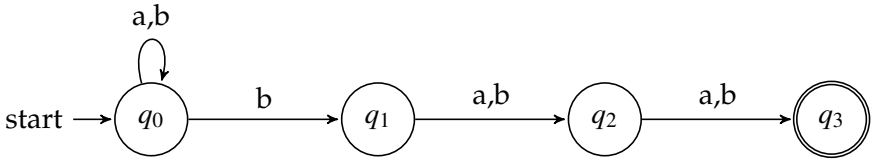
9.5 Example. This *NFA* accepts

$$\{x \in \{a, b\}^* \mid \text{ the next-to-last symbol in } x \text{ is b.}\}$$



It's easy enough to make a *DFA* for this language. But let's generalize this language:

9.6 Example. Let Σ be $\{a, b\}$ and let L_3 be the set of strings whose 3rd-to-last symbol is an a. For example $bbaba \in L$, while $bbbbab \notin L_3$. It is easy to make an *NFA* recognizing L_3:

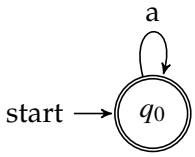


We can obviously play the same game for "the 4th-to-last symbol", etc.

It can be shown that the smallest *DFA* for L_3 has 8 states. Indeed, if we wanted to accept strings who kth-from-last input is a given symbol, the smallest *DFA* has 2^k states. See Problem 143, later.

This set of examples sows that *NFAs* can be exponentially more succinct than *DFAs*.

9.7 Example. Let $\Sigma = \{a, b\}$ This *NFA* accepts $\{a^n \mid n \geq 0\}$.



Of course, $\{a^n \mid n \geq 0\}$ can be viewed as a language over the alphabet $\Sigma = \{a\}$ or as a language over a larger alphabet such as $\Sigma = \{a, b\}$. A picture of an *NFA* such as the one given doesn't by itself say which way to think about the language; if it is important to know the base alphabet one must declare that independently of the picture.

9.8 Check Your Reading. *Why didn't we make the remark about the ambiguity of that alphabet back when we drew pictures of DFAs?*

Textual Representations of *NFAs*

If you want to represent an *NFA* in text, the representation introduced in Section 7.4 works just fine. The fact that thee can zero or several transitions from a given state on a given input doesn't change the idea. One example should make this clear.

Example 9.5 could be presented like this:

$$
\begin{aligned}
& q_0 \text{ is the start} \\
q_0 &\to a\, q_0 \\
q_0 &\to b\, q_0 \mid b\, q_1 \\
q_1 &\to a\, q_2 \mid b\, q_2 \\
q_2 &\to \lambda
\end{aligned}
$$

9.3 The Subset Construction: From *NFAs* to *DFAs*

We have two kinds of automata we've been working with so far, each of which has its charms: *DFAs* are a more intuitive *computational* model, while *NFAs* are more succinct and have other advantages that we will see eventually.

We observed when we defined finite automata that *DFAs* are automatically *NFAs*. In this section we will show that these two types of machines are actually equally powerful, in the sense that any language accepted by an *NFA* is in fact recognizable by some *DFA*.

You may ask at this point: "why did we bother to define these different formalisms, if we are going to end up showing them to equivalent?" That's a very reasonable question. And the answer is extremely important.

Having more than one different way to define systems is very powerful. It means that if you have a certain job to do (for example, building a system, or reasoning about it) you get to *choose* which definition to use, based on which is more convenient *for you* to do the job at hand. We will see lots of examples of this as we go forward but it should already be easy to get an intuition why this is true. Consider *DFAs* and *NFAs*, and suppose for now that we have already proved their equivalence. If your goal is to define an automation to do a certain job, you are likely to have an easier time of it building an *NFA*: you take advantage of the non-determinism. On the other hand, if you want to prove that there can be no *FA* to so a certain job, it is likely to be easier to prove that there is no *DFA* that works, precisely because they are so limited. In principle one could have used *NFAs* or *DFAs* in either situation (since theoretically they are equivalent) but as a practical matter you—and your reader—will be better off for your having chosen one or the other formalism.

The Construction Before stating the theorem let's describe the construction. Suppose $N = (\Sigma, Q, \delta_N, s, F)$ is an *NFA*. We are going to build a *DFA* D recognizing the same language as N.

Let $Pow(Q)$ denote the set of subsets of Q. These will be the states of D. The key idea is this: when N processes a string x there are several (due to non-determinism) *possible* states of N that can result. But we can view this collection of possible N-states as a *single* D-state. We just need to define the δ function of D in such a way to maintain this invariant: after processing any string x, the D-state we end up in will be comprised of all possible N-results on x.

The details are in Algorithm 2.

Algorithm 2: *NFA* to *DFA*

Input: a *NFA* $N = (\Sigma, Q, \delta, s, F)$
Output: a *DFA* D such that $L(D) = L(N)$
The states of D will be elements of $Pow(Q)$;
The start state of our D will be $\{s\}$. ;
The set $\mathcal{F}$ consists of those sets of N-states that contain at least one accepting state of N.;
We define δ_D by

$$P \xrightarrow{c} R \text{ if there is some state } p \in P \text{ and some state } r \in R \text{ with } p \xrightarrow{c} r \text{ in } N.$$

return $D = (\Sigma, Pow(Q), \delta D, \{s\}, \mathcal{F})$

9.9 Theorem. *Let N be an NFA. The DFA D constructed in Algorithm 2 satisfies $L(D) = L(N)$.*

Proof. During this proof we will use capital letters to name states of the *DFA* D (as we did in the algorithm) to help remind you that such a state in D began life as a *set* of states in N.

To prove the theorem it suffices to establish that the following holds for any string x. Here $\longrightarrow\!\!\!\!\!\rightarrow$ means "zero or more steps of $\longrightarrow$." Since there are two machines being discussed, we use a subscript to make it clear which one we're talking about.

Claim:

$P \xrightarrow{x} R$ in D if and only if there is some state $p \in P$ and some state $r \in R$ with $p \xrightarrow{x} r$ in N.

Note that this looks just like the *definition* of $\longrightarrow$ in D, except that it is about $\longrightarrow\!\!\!\!\!\rightarrow$ instead. That is, it is about strings rather than characters.

And it is important to note that we are not allowed to *simply declare* that the claim is true: once we have defined $\longrightarrow$ then $\longrightarrow\!\!\!\!\!\rightarrow$ is determined, and the claim is either true or false.

We prove this claim by induction on the length of x.

When $x = \lambda$: $P \overset{\lambda}{\longrightarrow\!\!\!\!\!\rightarrow} R$ in D precisely when $R = P$. And $p \overset{\lambda}{\longrightarrow\!\!\!\!\!\rightarrow} r$ in N precisely when $p = r$. So the claim holds there.

For the inductive step, suppose that $x = cy$ for $c \in \Sigma$ and $y \in \Sigma^*$. We want to show that

$$P \overset{cy}{\longrightarrow\!\!\!\!\!\rightarrow} \{r \mid \exists p \in P,\ p \overset{cy}{\longrightarrow\!\!\!\!\!\rightarrow} r \text{ in } N\}$$

Using the definition of $\longrightarrow$ we have

$$P \overset{c}{\longrightarrow} \{p_1 \mid \exists p \in P,\ p \overset{c}{\longrightarrow} p_1 \text{ in } N\} \quad \ldots \text{call this set } P_1$$

Using the induction hypothesis on y we have

$$P_1 \overset{y}{\longrightarrow\!\!\!\!\!\rightarrow} \{r \mid \exists p_1 \in P_1,\ p_1 \overset{y}{\longrightarrow\!\!\!\!\!\rightarrow} r \text{ in } N\}$$

Putting these together we have

$$\begin{aligned} P \overset{cy}{\longrightarrow} & \{r \mid \exists p \in P\ \exists p_1 \in P_1,\ p \overset{c}{\longrightarrow} p_1 \overset{y}{\longrightarrow\!\!\!\!\!\rightarrow} r \text{ in } N\} \\ = \quad & \{r \mid \exists p \in P\ p \overset{cy}{\longrightarrow\!\!\!\!\!\rightarrow} r \text{ in } N\} \end{aligned}$$

which is what we wanted.

Now given that claim we are in good shape. Assuming the claim, we have that for any string x,

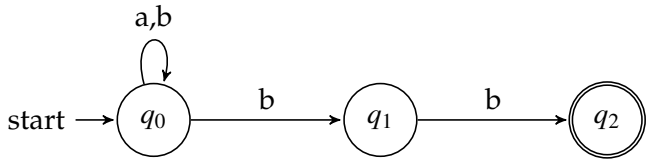
$$\{s\} \overset{x}{\longrightarrow\!\!\!\!\!\rightarrow} R \text{ if there is some state } s \in \{s\} \text{ and some state } r \in R \text{ with } s \overset{x}{\longrightarrow\!\!\!\!\!\rightarrow} r \text{ in } N.$$

Since s is the only state in $\{s\}$ this says that

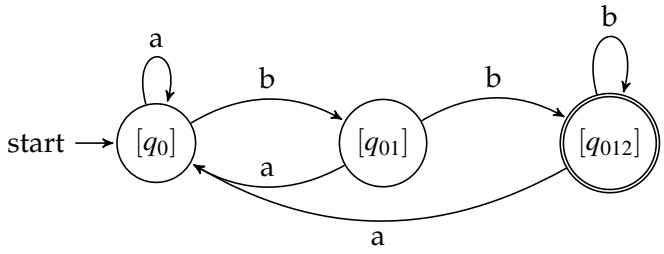
$$\{s\} \overset{x}{\longrightarrow\!\!\!\!\!\rightarrow} R \text{ if there is some state } r \in R \text{ with } s \overset{x}{\longrightarrow\!\!\!\!\!\rightarrow} r \text{ in } N.$$

Now, x is accepted by N if and only if the set on the right-hand side above contains a state in F. But this just to say that the set on the right-hand side is in $\mathcal{F}$, which is the same as saying that D accepts x. ///

9.10 Example. Here is an *NFA* accepting strings that end in *bb*.

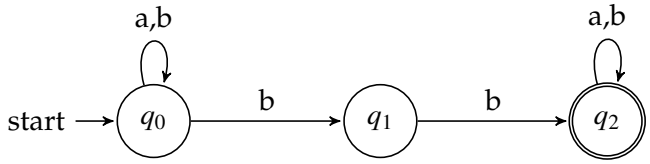


After the subset construction, we get

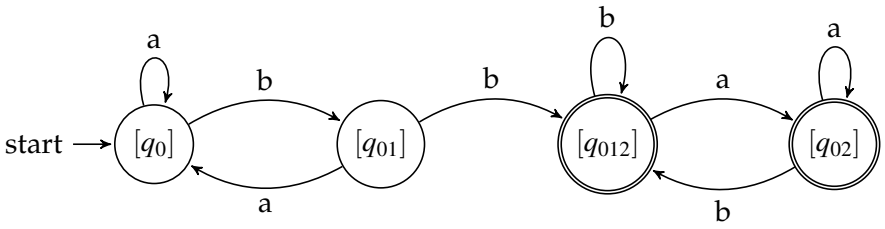


This is the *DFA* you would probably write directly if given this specification.

9.11 Example. Here is a slight variation on Example 9.10. Here we accept strings that *contain bb*.



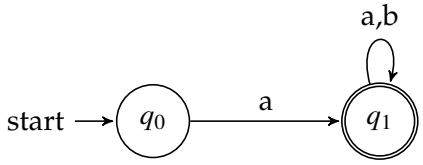
After the subset construction, we get



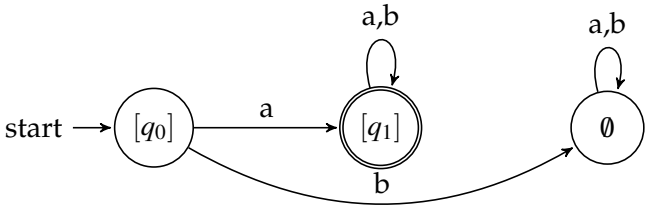
This is probably *not* the *DFA* you would probably write directly if given this specification. If you think about it you will see that this is not the smallest *DFA* we could write for this language. We'll come back to this point, in Section 14.

9.12 Example. Sometimes an *NFA* will fail to be a *DFA* only because some transitions are not defined (as opposed to having more than one transition for certain inputs). It is easy to repair this kind of non-determinism by adding a non-accepting "fail" state, and sending all the missing transitions there. It is amusing to see that the subset construction does this same thing.

For example, here is an *NFA* that accepts strings that start with a.



After the subset construction, we get



9.4 Closure Constructions on *NFAs*?

Complement The complement construction 8.1 on *NFAs* fails, in the sense that swapping accepting and non-accepting states in an *NFA* does *not* have the effect of complementing the accepted language. Please do Problem 105.

Intersection In Section 8.2 we showed how to build a "product" of two *DFAs* M and N, yielding a *DFA* P with $L(P) = L(M) \cap L(N)$, It turns out that this construction works, without modification. As a matter of fact, the proof of Theorem 8.6 still applies, word-for-word. In other words, nowhere in the proof of that theorem did we make use of the fact that we were working with *DFAs* as opposed to *NFAs*.

Union But—in the category of "don't jump to conclusions"—the result breaks down for unions. That is to say, if you look at the variation on the product construction that captures the *union* of *DFA* languages, it fails to do the right thing on *NFAs*. Please do Problem 108.

Concatenation and Kleene Star We noted in our discussion of *DFAs* that it is not so clear how to take a *DFA* M and build a that recognizes $L(M)^*$. And it is not so clear how to take a *DFAs* M_1 and M_2 and build a *DFA* that recognizes $L(M_1)L(M_2)$.

We need one more idea, a tweak on the notion of *NFA*, to arrive at the kind of machine that makes all union, concatenation, and Kleene star all easy to handle. That new kind of machine is a NFA_λ.

9.5 Introducing $NFA_\lambda s$

An NFA_λ is an *NFA* that, in addition to transitioning from one state to another while reading an input symbol, has the capacity to transition from one state to another without consuming any input at all. But, as we will see soon, these new kind of automata have exactly the same recognizing power as ordinary *NFAs*, and hence *DFAs*.

This capacity is more natural than it might seem at first; it can be used to model systems in which some state-changing activity takes place internally to the machine, as opposed to being in response to some stimulus from the outside world. Since we are only looking at the most elementary notions of automata we won't explore this aspect. But one important thing we *do* want to exploit is the fact that NFA_λ are very easy to combine, under all the operations that regular expressions use.

9.13 Definition (NFA_λ)**.** A *nondeterministic finite automaton with* λ *transitions* (NFA_λ) is a 5-tuple $\langle \Sigma, Q\delta, q_{st}, F\rangle$, where Q, Σ, q_{st}, and F are as defined for *NFA*, but where δ allows, in addition to transitions $p \xrightarrow{c} q$, transitions $p \xrightarrow{\bullet} q$.

A **run** of an NFA_λ on $x = a_1 a_1 \ldots a_n$ is a sequence of states $\langle q_0, \ldots, q_k\rangle$ such that

- $q_0 = q_{st}$;
- for each $i \geq 0$, either for some a, $q_i \xrightarrow{a} q_{i+1}$ is a transition in δ or $q_i \xrightarrow{\bullet} q_{i+1}$ is a λ transition in δ, and
- the concatenation of the symbols in the labelled (non-λ) transitions yields x

A run is **accepting** if $q_n \in F$, M **accepts** x if there exists an accepting run of M on x, and the language $L(M)$ accepted by M is the set of strings accepted by M.

It is traditional to refer to transitions $p \xrightarrow{\bullet} q$ as "λ" transitions.[1]

[1] And some authors will actually write $p \xrightarrow{\lambda} q$. We don't like this notation because it seems to suggest that λ is an alphabet symbol, a common confusion.

The procedure for converting $NFA_\lambda s$ into $NFAs$ is conceptualy simple but subtle in details so let us first motivate why we care about $NFA_\lambda s$.

NFA_λ and Closure Properties

NFA_λ are very convenient for building new automata from old ones. We will see how, given $NFAs$ M and N, we can use λ-transition tricks to build $NFA_\lambda s$ P accepting (i) the union of the languages M and N accept; (ii) the concatenation of the languages M and N accept; (iii) the Kleene star of the language M accepts.

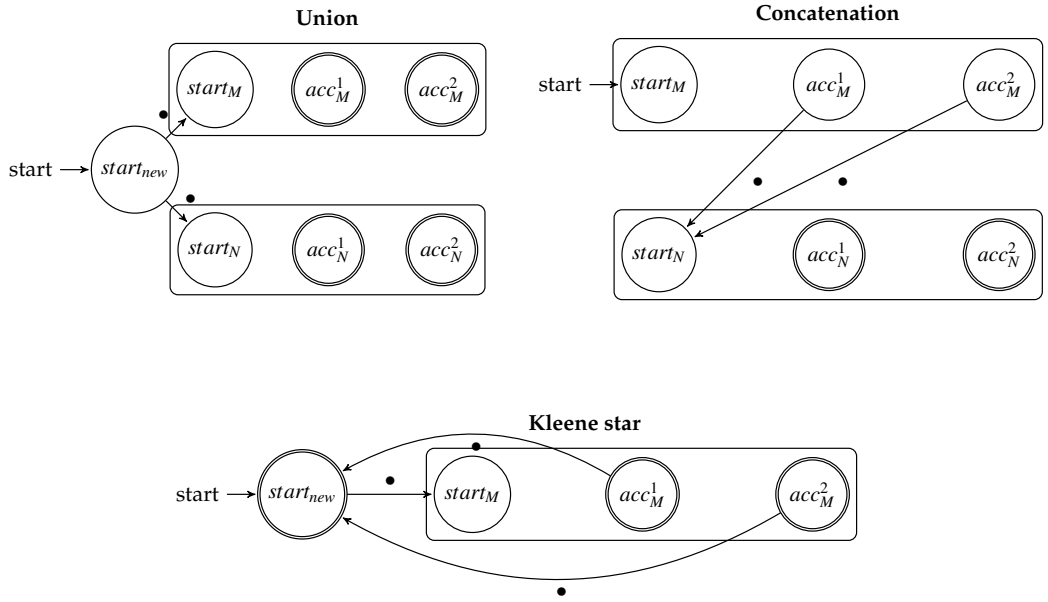
Union To build an NFA_λ recognizing the union of the languages of M and N, all we have to do is make a new start state, and let the machine "guess" which automaton to silently jump to.

Concatenation To build an NFA_λ recognizing the concatenation of the languages of M and N, we just allow the accepting states of the first machine to optionally silently jump to the start state of the second machine.

Kleene star To build an NFA_λ recognizing the Kleene-star of the language of M, we add a new start state, make it accepting in order to accept λ, let it jump silently to the old start state, and let each previously accepting state jump silently back to this new start state.

Algorithms 3, 4, and 5 constitute precise descriptions of these constructions.

See Problem 114 for a subtlety concerning these constructions.



Algorithms and Correctness

Algorithm 3: NFA_λ-Union

Input: two NFA_λs M and N
Output: an NFA_λ P such that $L(P) = L(M) \cup L(N)$

Let $M = (\Sigma, Q_M, \delta_M, s_M, F_M)$
Let $N = (\Sigma, Q_N, \delta_N, s_N, F_N)$
By renaming states if necessary, we may assume that $Q_M \cap Q_N = \emptyset$;

- Let s_{new} be a new state, not in Q_M or Q_N.
- The state space of P is $Q_M \cup Q_N \cup \{s_{new}\}$.
- The start state of P is $\{s_{new}\}$.
- The set of accept states of P is $F_M \cup F_N$.
- The transition function δ_P of P consists of the following transitions
  - Every transition of δ_M and of δ_N is a transition of M;
  - λ transitions $s \xrightarrow{\bullet} s_M$ and $s \xrightarrow{\bullet} s_N$.

return $P = (\Sigma,\ Q_M \cup Q_N \cup \{s_{new}\},\ \delta_P,\ \{s_{new}\},\ F_M \cup F_N)$

The following theorem states the correctness of our algorithms.

9.14 Theorem. *If M and N are NFA_λs then*

1. *the NFA_λ P returned by Algorithm 3 satisfies $L(P) = L(M) \cup L(N)$;*
2. *the NFA_λ P returned by Algorithm 4 satisfies $L(P) = L(M)L(N)$;*
3. *the NFA_λ P returned by Algorithm 5 satisfies $L(P) = L(M)^*$.*

Proof. We omit the proof here. ///

Your first reaction to the construction in Algorithm 5 might well be: it doesn't have to be that complicated. Problem 115 asks you to talk yourself out of that reaction.

Algorithm 4: NFA_λ-Concatenation

Input: two NFA_λs M and N
Output: an NFA_λ P such that $L(P) = L(M)L(N)$

Let $M = (\Sigma, Q_M, \delta_M, s_M, F_M)$
Let $N = (\Sigma, Q_N, \delta_N, s_N, F_N)$
By renaming states if necessary, we may assume that $Q_M \cap Q_N = \emptyset$;

- The state space of P will be $Q_M \cup Q_N$
- The start state of P is s_M, the start state of M
- The set of accept states of P is F_N, the accept states of N
- The transition function δ_P of P consists of the following transitions
  - Every transition of δ_M and of δ_N is a transition of P;
  - For every state f in F_M, a λ transition $f \xrightarrow{\bullet} s_M$

  Caution: the old start state of N is no longer starting, as a state of P, and the old accept states of M are no longer accepting, as states of P.

return $P = (\Sigma,\ Q_M \cup Q_N,\ \delta_P,\ s_M,\ F_N)$

9.6 From NFA_λ to NFA

What remains now if to show that we didn't really *need* to use NFA_λ in order to get automata capturing regular expressions: ordinary NFA can do the job as well. That is, we want to prove the following claim.

> For every NFA_λ M there exists an NFA M' such that
>
> 1. $L(M') = L(M)$
> 2. M' has no λ transitions
>
> Furthermore, there exists an algorithm to compute M' from M.

Note that we can't just *delete* λ transitions of course: we must compensate for removing them.

9.15 Check Your Reading. *Give an example of an NFA_λ M with the property that if we build M' by simply removing λ transitions then $L(M') \neq L(M)$.*

The strategy for building M' from M is clever.

Algorithm 5: NFA_λ-Kleene-closure

Input: an NFA_λ M
Output: an NFA_λ P such that $L(P) = (L(M))^*$

Let $M = (\Sigma, Q_M, \delta_M, s_M, F_M)$

- Let s_{new} be a new state, not in Q_M.
- The state space of P is $Q \cup \{s_{new}\}$.
- The start state of P is s_{new}
- There is only one accept state of P, namely s_{new}
- The transition function δ of P consists of the following transitions
  - Every transition of M is a transition of P
  - For every state f in F, a λ transition $f \xrightarrow{\bullet} s_{new}$
  - A λ transition $s_{new} \xrightarrow{\bullet} s_M$

  Caution: the old start and accepting states of M are no longer starting or accepting, as states of P.

1. Build an auxiliary NFA_λ M^+;
2. Define M' as M^+ with all λ transitions deleted

The key idea of the algorithm is contained in the following lemma. It expresses the fact that certain transformations of an NFA_λ leave the language of the NFA_λ unchanged.

9.16 Lemma. *Suppose NFA_λ M has the transition $p \xrightarrow{\bullet} q$.*

If M has a transition $q \xrightarrow{\bullet} r$ and we build a new NFA_λ by adding *the transition $p \xrightarrow{\bullet} r$, the resulting NFA_λ accepts exactly the same language as did M.*

If M has a transition $q \xrightarrow{a} r$ and we build a new NFA_λ by adding *the transition $p \xrightarrow{a} r$, the resulting NFA_λ accepts exactly the same language as did M.*

Proof. Easy: each such added transition can be simulated by the original transitions. ///

That lemma seems to take us in the wrong direction: it *adds* transitions rather than *removing* the ones we don't want. But the trick is that after all such transitions have been added, we can *then* just delete the bad transitions. Here is this idea expressed as an algorithm.

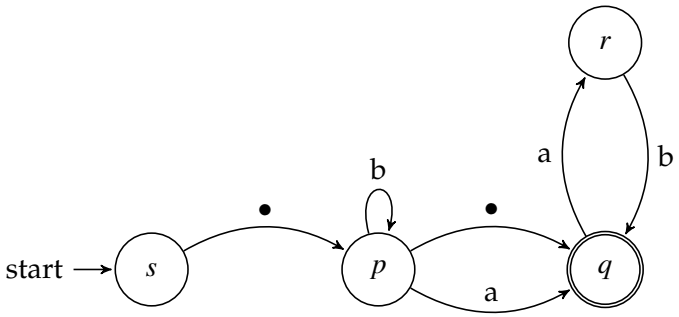
In the statement of the algorithm below, observe that in building M' from M we do not change any components of M except the transition function δ and the accepting set F'.

Algorithm 6: NFA_λ to NFA

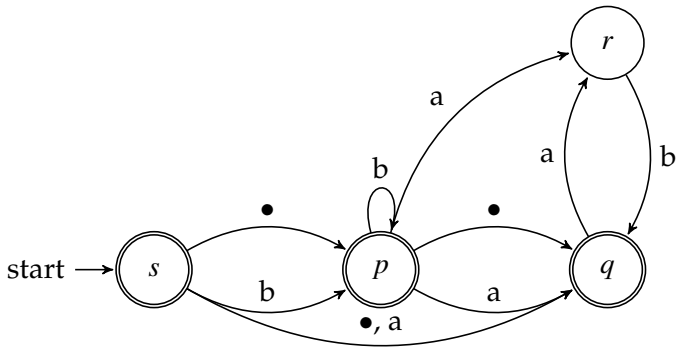
Input: a NFA_λ $M = (\Sigma, Q, \delta, s, F)$
Output: a NFA M' such that $L(M') = L(M)$
initialize: set δ^+ to be δ
repeat
  if δ^+ has a transition $p \xrightarrow{\bullet} q$ and a transition $q \xrightarrow{\bullet} r$ then add $p \xrightarrow{\bullet} r$ to δ^+
  if δ^+ has a transition $p \xrightarrow{\bullet} q$ and a transition $q \xrightarrow{a} r$ then add $p \xrightarrow{a} r$ to δ^+
until *no change in* δ^+;
let F' be $F \cup \{p \mid$ there is a λ transition $p \xrightarrow{\bullet} f$ to a state f in $F\}$;
let $M^+ = (\Sigma, Q, \delta^+, S, F')$;

define δ' to be δ^+ with all λ transitions removed ;
return $M' = (\Sigma, Q, \delta', S, F')$

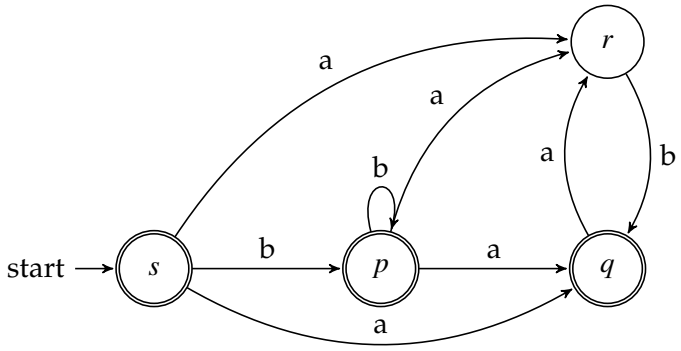
9.17 Example. Let M be the NFA_λ.



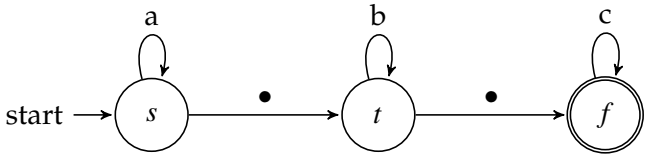
After adding transitions in the first phase of Algorithm 6 we get



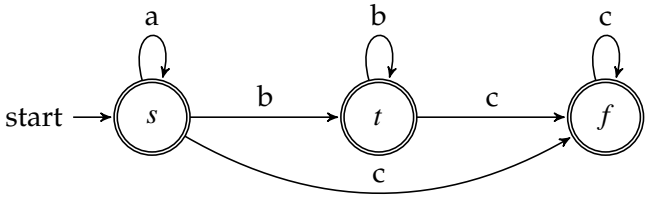
Erasing the λ-transitions yields an *NFA* equivalent to the original.



9.18 Example. Another example. Let M be the NFA_λ



Eliminating λ transitions yields the *NFA*



9.19 Theorem. *For any NFA$_\lambda$ M, Algorithm 6 will return an NFA M' such that $L(M') = L(M)$.*

Proof. Termination of the algorithm is easy. Having fixed Q and Σ, there are only finitely many possible transitions that could conceivably be present in M^+. (How many?) Each iteration of the repeat loop adds one, and so there can be only finitely many iterations.

It is obvious from the algorithm that the output M' is an *NFA* (has no λ transitions)

To prove $L(M') = L(M)$, we first prove $L(M^+) = L(M)$, and then prove $L(M') = L(M^+)$.

To prove $L(M^+) = L(M)$: Since we start with $\delta^+ = \delta$ and we never delete transitions, it is obvious that $L(M) \subseteq L(M^+)$. To prove $L(M^+) \subseteq L(M)$ it suffices to prove that *at each stage* of the construction of the transitions δ^+ the automaton at that stage accepts no new Σ^+ strings. This is the content of Lemma 9.16.

Now to prove $L(M') = L(M^+)$: Since the transitions of M' are a subset of those of M^+ it is obvious that $L(M') \subseteq L(M^+)$. The only interesting part of the proof is what remains, proving $L(M^+) \subseteq L(M')$.

It suffices to prove that for any string $w \in L(M^+)$ there is an accepting run in M^+ that does not use λ transitions.

For this it suffices to show the following claim:

> If $s \xrightarrow{w}\!\!\!\!\rightarrow f$ is a accepting run of M^+ on w with the least number of transitions, then no λ transition of M^+ is used.

If we establish this claim it will follow that we can just remove the λ transitions without changing which strings are accepted, that is, that M' accepts all the words that M^+ does.

Proof of claim: For sake of contradiction suppose that somewhere a λ transition was used, let us focus on the first one. There are two cases. One case is when the last λ transition is actually the last transition in the run:

$$s \xrightarrow{w}\!\!\!\!\rightarrow p \xrightarrow{\bullet} f \quad \text{where } f \in F'$$

Thus, in M^+, p is a state with a λ transition to a state f in F', and this case the algorithm added p to F'. This means that

$$s \xrightarrow{w}\!\!\!\!\rightarrow p \quad \text{where } p \in F'$$

is an accepting run on w in M^+, with fewer transition than the one we started with, contrary to our assumption.

The other case is when the part of the run after the last λ transition has at least one transition:

$$s \xrightarrow{x}\!\!\!\!\rightarrow p \xrightarrow{\bullet} q \xrightarrow{a} r \xrightarrow{y}\!\!\!\!\rightarrow f \quad \text{where } w = xay \text{ and } f \in F'$$

This notation allows the possibility that $y = nil$ and $r = f$. In this cases the algorithm ensures that there is a transition $p \xrightarrow{a}\!\!\!\!\rightarrow r$. So eliminating the transition from p to q yields a shorter accepting run, contradicting our assumption:

$$s \xrightarrow{x}\!\!\!\!\rightarrow p \xrightarrow{a} r \xrightarrow{y}\!\!\!\!\rightarrow f$$

This completes the proof that *NFA* M' performs as advertised. ///

A variation on the construction?

In Algorithm 6 we focused on adding $p \xrightarrow{a} r$ to δ^+ whenever we saw the pattern $p \longrightarrow q \xrightarrow{a} r$. A natural question is: how about patterns $p \xrightarrow{a} q \longrightarrow r$? Should we add $p \xrightarrow{a} r$ to δ^+ when we see *that*? The answer is: it wouldn't be wrong, in the sense that if we added those in our algorithm we would still build a correct *NFA*. That's clear: the justification for adding these transitions is the same as the one for adding them in the original case. But we don't *need* to deal with these patterns. The proof of correctness of our Algorithm 6 tells us that.

Here's another question: suppose we had done the algorithm based on these $p \xrightarrow{a} q \longrightarrow r$ patterns *instead of* the $p \xrightarrow{a} q \longrightarrow r$ patterns we did use? In other words, suppose in Algorithm 6 we *replaced* the line

> if δ^+ has a transition $p \xrightarrow{\bullet} q$ and a transition $q \xrightarrow{a} r$ then add $p \xrightarrow{a} r$ to δ^+

by

> if δ^+ has a transition $p \xrightarrow{a} q$ and a transition $q \longrightarrow r$ then add $p \xrightarrow{a} r$ to δ^+

Would the algorithm still be correct?

The answer is no. Problem 115 asks you to find a counterexample.

9.7 Problems

103. NFASmaller

For each of the languages in Problem 92, see if you can make an *NFA* for the language with fewer states than the *DFA* you made there.

104. NFAToDFA

Do lots of examples of NFA to DFA conversion. At the very least, convert all of the *NFA* used as examples in this chapter: Example 9.4, Example 9.5, Example 9.7.

Do Example 9.7 twice, over $\Sigma = \{a\}$ and over $\Sigma = \{a, b\}$.

105. NFAComplement

An important problem!

a) Let M be an *NFA* accepting the language L. Suppose we build the *NFA* M' by using the same states and transitions as M but declaring that a state in M' is accepting if and only if it is *not* accepting in M. Show by giving a concrete example that the language accepted by M' is *not* necessarily $\overline{L}$, the complement of L (that is, the set $\Sigma^* - L$).

b) Show that if N is an *NFA* accepting the language L, then there is an *NFA* accepting the language $\overline{L}$.

c) Explain why the previous two parts do not contradict each other..

106. NFACount

In the spirit of Problem 94.

a) Let's explore: how many different *NFAs* are there with one state? As in Problem 94 we want to ignore superficial differences, so we ask: How many different *NFAs* are there with one state named q_0, over the input alphabet $\Sigma = \{a_0, a_1\}$? Draw them. If two machines are structurally different but accept the same language, count them as different.

b) How many different *NFAs* are there with two states named q_0, q_1, over the input alphabet $\Sigma = \{a_0, a_1\}$? Do not assume that q_0 is the start state. If two machines are structurally different but accept the same language, count them as different.

107. MakeNFAs

Fix any alphabet Σ.

a) Let K be any language containing a single string. Show that K is regular.

Hint. Start your proof like this.

Let K consist of the single string $x = a_1 \dots a_n$ where each a_i is an element of Σ. We can build an NFA M whose language is $\{x\}$ as follow. ...

Fill in the dots.

b) Let K be a finite language, that is, K is a language containing finitely many strings. Show that K is regular.

Hint. Start your proof like this.

By the previous part, there are k NFAs, accepting precisely the individual strings x_i ...

Fill in the dots.

c) Let K be a *cofinite* language, that is, K is a language whose *complement* contains only finitely many strings. Show that K is regular.

Hint. Quote a theorem.

108. NFAProduct

Here we explore the proposal to use the product construction, in the variation that was used to capture union, but on *NFAs* this time. That is, suppose M_1 and M_2 are *NFAs*, and we apply Definition 8.4, with accepting states being those (f_1, f_2) such that *either* f_i is accepting in its original *NFA*. This defines a machine P which is a legal *NFA*.

Show by giving a concrete example that it is *not* necessarily true that $L(P) = L(M_1) \cup L(M_2)$. So the "product construction for unions" fails for *NFAs*.

Can you think of a natural condition on *NFAs* M_1 and M_2 that will ensure that the "product construction for unions" will succeed? (Saying "M_1 and M_2 must be *DFAs*" is a boring answer ...).

109. NFASimulation

Given an algorithm for simulating an NFA efficiently (that is, without constructing an equivalent DFA first). Your algorithm should run in time $O(|w||Q|^2)$ where Q is the set of states of N.

Formally: If N be an NFA, show how to write an algorithm which on input w will decide whether or not $w \in L(N)$. If N has n states your algorithm should run in time polynomial in n and the length of w.

Hint. The key is the data structure. Suppose N has n states, which we may identify with the numbers 0 through $n-1$. A bit-array with array indices 0 through $n-1$ can represent a subset of states (obviously, the states where a 1 is stored). Use two of these!

110. ForallNFAs

As you know, a standard NFA N accepts a string w if *there exists* an accepting run of N on w. But we can imagine a new kind of finite automaton, called an *for-all-NFA*. Such a machine A has the same "hardware" as an NFA:

$$A = (\Sigma, Q, \delta, s, F)$$

as before, with δ mapping (state, symbol) pairs into sets of states. The difference is that we declare that a string w is accepted by a for-all-NFA if *every* run of the machine on w ends in a state in F.

Prove that for-all-NFAs accept precisely the regular languages.

Hint. First prove that for every regular language L there is an for-all-NFA A with $L(A) = L$. Then prove that for any for-all-NFA A the language $L(A)$ is regular. The first part is easy. For the second part, think about the subset construction.

111. Hamming

If x and y are bit strings, the *Hamming distance* $H(x,y)$ between them is the number of places in which they differ. If x and y have different lengths then we declare their Hamming distance to be infinite. This is an important concept in analyzing transmission of signals in the presence of possible errors.

If x is a string and L is a language over $\{0,1\}$ then we define

$$H(x,L) \stackrel{\text{def}}{=} \min\{H(x,y) \mid y \in L\}$$

Now for any language L over $\{0,1\}$ and any $k \geq 0$ we may define

$$N_k(L) \stackrel{\text{def}}{=} \{x \mid H(x,L) \leq k\}$$

For example, if L were the language $\{00,001\}$ then $N_0(L) = L$, $N_1(L) = L \cup \{10,01,101,011,000\}$, and $N_2(L)$ is the set of all bit strings of length 2 or three except for the string 110.

Prove that if $L \subseteq \{0,1\}^*$ is regular, then so is $N_1(L)$.

Is there anything special about a Hamming distance of 1 here, or is the result true for any distance?

Hint. Suppose L is $L(M)$ for a DFA with state set Q. Build an NFA for $N_1(L)$ with state set $Q \times \{0,1\}$. The second component tells how many "errors" you have seen so far. Use non-determinism generously!

112. HalvesRegular

Suppose L is a language; we may define the following language based on L:

$$\text{Half}L = \{x \mid \exists y |y| = |x| \text{ and } xy \in L\}$$

Suppose L is regular. Is Half L guaranteed to be regular?

[This is a hard problem.]

113. ReverseDFA

If L is a language let us write L^R for the language consisting of the reverses of the strings in L. Show how, given a *DFA* M for L, we can construct an NFA_λ for L^R.

114. KleeneStarGotcha

The following seems like it should be a simpler way to do the construction in Algorithm 5, building an NFA_λ for A^* out of an NFA_λ for A. Rather than add a new start state s_{new}, just (i) make the old start state accepting (so as to accept λ, since this is always in A^*), and (ii) add λ transitions from each accepting state of M to the start states (in order to allow the automaton to iterate).

Show by example that this idea fails.

115. NFANilGotcha

Suppose in Algorithm 6 we **replaced** the line

> if δ^+ has transition $p \xrightarrow{\bullet} q$ and transition $q \xrightarrow{a} r$, add $p \xrightarrow{a} r$ to δ^+

by

> if δ^+ has transition $p \xrightarrow{a} q$ and transition $q \longrightarrow r$, add $p \xrightarrow{a} r$ to δ^+

Prove, by showing a counterexample, that this revised algorithm does not correctly eliminate λ-transitions.

116. NFANilFussiness

The construction in Algorithm 3 is very simple. But there is one subtlety: note that we wrote, " By renaming states if necessary, we may assume that $Q_M \cap Q_N = \emptyset$." Why is that fussiness necessary? Give an example that shows that Algorithm 3 can construct an automaton that does *not* accept $L(M) \cup L(N)$ if we neglect to ensure $Q_M \cap Q_N = \emptyset$.

The point being made in this problem applies to Algorithm 4 as well.

117. NFANilBigO

The asymptotic worst case running time of Algorithm 6 is very bad. Give an example to show that it does *not* run in polynomial time. Use as your measure the number of transitions in the automataon.

Hint. For each n, consider the very dumb NFA_λ with n states $\{q_0, \ldots, q_{n-1}\}$ and n λ-transitions $q_i \xrightarrow{\bullet} q_{i+1}$.

118. NFAUnionBigO

Let's compare the running time of two algorithms for the following problem.

> *NFA Union*
>
> INPUT: two *NFAs* M and N
>
> QUESTION: a *DFA* P such that $L(P) = LM) \cup L(N)$

Note that we start with *NFAs* but we want a *DFA* as our answer.

1. Algorithm A is: convert M and N separately to *DFAs* using the subset construction; then do the product construction on the results.
2. Algorithm B is: use Algorithm 3 on M and N, followed by conversion of the resulting NFA_λ to a *DFA*.

Give the worst-case asymptotic running time of each algorithm, as a function of the number of states on M and N. Is one of them preferable in the sense of asymptotic running time?

In your calculations,

- charge $O(2^q)$ for a call to the subset construction on an automata with q states
- charge $O(q_1 q_2)$ for a call to the product construction on automata with q_1 and q_2 states
- charge $O(q_1 + q_2)$ for a call to Algorithm 3 on automata with q_1 and q_2 states

Chapter 10

Patterns and Regular Expressions

We met patterns informally in Chapter 6. Let's start studying them carefully now; we'll start the discussion from scratch.

10.1 Introduction

One way to describe a language—the most general way, really—is to use traditional set notation, as in

$$\{x \in \{a,b\}^* \mid x \text{ starts and ends with the same letter } \}$$

That's fine but one of the themes of this course is to look for simpler formalisms to use when they suffice, since they can be easier to reason about, execute, etc. Regular expressions are a formalism for describing (certain) languages that occur frequently.

The simplest possible languages are the empty language and languages with only a single string consisting of a single symbol. We can just write these explicitly:

$$\emptyset, \quad \{a\}, \quad \{b\}, \quad \textit{etc.}$$

We can get languages beyond singleton letters by using union and by using concatenation, for example

$$(\{a\} \cup \{b\}), \quad ((\{a\}\{b\})\{a\}),$$

To add some excitement we can allow ourselves to use Kleene star:

$$((\{a\}^*\{b\}^*), \quad (\{a\} \cup \{b\})^*), \textit{ etc.}$$

We could use other Boolean operations such as intersection and complement but (mysteriously) we will skip those for a while. See Section 10.3.

Not every language can be described using just these tools. But the ones that can are especially interesting. To ease our study of them let's streamline our notation a bit.

1. Let's agree to suppress the set-brackets around singleton languages. So we'll write **a** instead of $\{a\}$.
2. Since concatenation is associative, we can suppress parentheses grouping concatenations. So we'll write **aba** instead of **a**(**ba**), which is already shorthand for $\{a\}(\{b\}\{a\})$.

10.2 Pure Regular Expressions

We will call a language description written using the above notation a *pure regular expression*. Since we will be studying these expressions, and the corresponding languages, quite carefully, we give a formal inductive definition in Definition 10.1.

10.1 Definition. Let Σ be an alphabet. The regular expressions over Σ and the languages they denote are defined inductively as follows.

- **O** is a regular expression.
  It denotes the empty language: $L(\mathbf{O}) =$ the empty set.
- For each $a \in \Sigma$, **a** is a regular expression.
  It denotes the one-element language consisting of the length-1 string a: $L(\mathbf{a}) = \{a\}$.
- If E_1 and E_2 are regular expressions then $(E_1 + E_2)$ is a regular expression.
  It denotes the union of the languages of E_1 and E_2: $L(E_1 + E_2) = L(E_1) \cup L(E_2)$.
- If E_1 and E_2 are regular expressions then $(E_1 E_2)$ is a regular expression.
  It denotes the concatenation of the languages of E_1 and E_2: $L(E_1 E_2) = L(E_1)L(E_2)$.
- If E is a regular expressions then E^* is a regular expression.
  It denotes the Kleene star of the language of E: $L(E^*) = L(E)^*$.

Caution. Be sure to make the distinction between a regular *expression*, a syntactic object, and the *language* it denotes.

For example, when we write $L(E_1 + E_2) = L(E_1) \cup L(E_2)$, the expression $E_1 + E_2$ found on the left-hand-side is the syntactic combining of two expressions, while the right-hand-side $L(E_1) \cup L(E_2)$ refers to the *language* operation of union.

When we write, above, $L(E_1 E_2) = L(E_1)L(E_2)$, the expression $L(E_1)L(E_2)$, found on the left-hand-side is the syntactic combining of two expressions, while the right-hand-side $L(E_1)L(E_2)$ refers to the *language* operation of concatenation. A similar remark applies to the expressions denoting Kleene star.

To help see the difference notice that there can be many different regular *expressions* denoting the same language (for example $E + \mathbf{O}$ will certainly denote the same language as E). In fact Section 5.7.1 has lots more examples.

Syntax Conventions As usual we use parentheses in the concrete syntax as needed. To avoid a riot of parentheses we adopt the conventions that (i) the star has higher precedence than $+$ and concatenation; (ii) concatenation has higher precedence than $+$. We also take advantage of associativity of concatenation and union and treat them, syntactically, as multi-arity operations. So for example, instead of writing the official

$$((\mathbf{ab})(\mathbf{c}^*)) + \mathbf{d}$$

we may write

$$\mathbf{abc}^* + \mathbf{d}$$

10.2 Examples. Fix the alphabet $\Sigma = \{a, b\}$.

- $\mathbf{a}^*$ denotes the set of all finite sequences of as (including the empty string)
- $(\mathbf{aa})^*$ denotes the set of all even-length finite sequences of as (including the empty string)
- $((\mathbf{aa}) + (\mathbf{aaa}))^*$ denotes the set of all finite sequences of a except for the single string a. (You have to think about this a little bit.)
- $(\mathbf{a} + \mathbf{b})^*$ denotes the set of all strings over $\{a, b\}$
- $\mathbf{a}^* + \mathbf{b}^*$ denotes the set of all strings that are either all-a or all-b.
- $(\mathbf{a} + \mathbf{b})^*\mathbf{b}(\mathbf{a} + \mathbf{b})^*\mathbf{b}(\mathbf{a} + \mathbf{b})^*$ denotes the set of all strings with at least two occurrences of b.
- $((\mathbf{a} + \mathbf{b})(\mathbf{a} + \mathbf{b}))^*$ denotes the set of all even-length strings.
- $(\mathbf{a} + \mathbf{b})^*\mathbf{aab}(\mathbf{a} + \mathbf{b})^*$ denotes the set of all strings with the string aab as a substring.
- $(\mathbf{ab} + \mathbf{b})^* + (\mathbf{ab} + \mathbf{b})^*\mathbf{a}^*$ denotes the set of strings that do not have aab as a substring.

The last two examples suggest an interesting question. If E is a given regular expression, must there always be a regular expression E' that denotes the complement of the language that E denotes? Based on the operations that regular expressions provide, this would seem to be unlikely (they are all "positive" in the sense that combining two regular expressions always yields one that matches more strings than the original). But amazingly, the answer is yes, for every language denoted by a regular expression there is a regular expression denoting its complement. But it will take some work for us to show this.

In practice you will see the phrase "regular expression" used to describe more general expressions than we have defined above. Let's discuss this next.

10.3 Syntactic Sugar : From Regular Expressions to Patterns

If you are familiar with the things called "regular expressions" in the context of UNIX, or Perl, or Python, you may feel like there are some things missing here. When regular expressions are used in applications, they are almost always given in a richer form, with other operators such as negation, intersection, symbol classes, back-references to subexpressions, etc.

These extended regular expressions are what we have been calling *patterns*.

But many of the operators used in applied settings can be treated as syntactic sugar over the official, minimal set of operators. Rather than be systematic we will be content with some examples.

1. Sometimes authors include λ as a regular expression, denoting the language consisting only of the empty string: $L(\lambda) = \{\lambda\}$. But this is unnecessary, since the standard regular expression $\mathbf{O}^*$ already denotes $\{\lambda\}$.

2. Suppose x is any string over some Σ, say $x = c_1 c_2 \ldots c_n$ where each $c_i \in \Sigma$. Then x itself is often treated as a *RegExp*, since each of the c_i is one of the base cases in the definition of *RegExp*, and x is their concatenation.

   To be perfectly precise, when we view x as a regular expression we are really speaking of the fully-parenthesized iteration of concatenations $(\ldots((c_1 c_2)c_3)\ldots c_n)$. But there is no need to be *that* pedantic: we just write $c_1 c_2 \ldots c_n$.

3. The expression Σ, which is understood to match any single alphabet symbol, is a shorthand for $\mathbf{a_0} + \mathbf{a_1} + \ldots + \mathbf{a_n}$ where these a_i are the individual symbols in Σ.

4. The expression E^+, which is understood to match one or more occurrences of strings matching E, is just a shorthand for EE^*.

5. The expression $E?$, which is understood to match zero or one occurrences of strings matching E, is just a shorthand for $\lambda + E$.

6. The expression $\sim E$, understood to match those strings which do *not* match E, can always be replaced by an "official" regular expression, but this is not so easy to see. The form of the official expression depends heavily on E. Nevertheless it is a theorem that whenever E is a regular expression, there exists a regular expression E' such that $L(E') = \overline{L(E)}$

   We have to do some work before we can prove this.

7. The expression $E_1 \cap E_2$, understood to match those strings which match *both* E_1 and E_2, can always be replaced by an "official" regular expression, but again this is not so easy to see. Nevertheless it is a theorem that whenever E_1 and E_2, are regular expressions, there exists a regular expression E such that $L(E) = L(E_1) \cap L(E_2)$.

   By the way, note that once you know that regular expression can capture complement then (since they can capture union, by definition) you can conclude that they can capture intersection as well. This follows from DeMorgan's Law $X \cap Y = \overline{\overline{X} \cup \overline{Y}}$

We will justify our claims about intersection and complement in Section 11.5.

So now you may ask: why not include operators such as intersection and complement in the official definition of regular expression? The main reason is this. It is good theory-engineering to work with a minimal set of concepts at the core. The idea is to have a small language mathematically, and define richer notations as syntactic sugar over these. We provide the rich set of operators to humans to work with, but when it is time to implement the language, or prove results about the language, we have much less work to do, because there are fewer cases to consider.

10.3.1 Regular Expression Membership

It can be difficult to some what language a regexp denotes, and even if you are convinced it can be tricky to prove that you are right. Let's work through an example. This expression is similar to the last example in Examples 10.2; we have helped ourselves to some syntactic sugar in using λ.

10.3 Example. Let E be $(\mathbf{ab}+\mathbf{b})^*(\lambda+\mathbf{a})^*$. Here is a proof that the language $L(E)$ denoted by E is the set of strings that do not contain *aab*.

There are two things to prove: (i) everything the expression matches fails to contain *aab*, and (ii) everything that fails to contain *aab* matches the expression.

1. Suppose x is matched by E. We want to show that x cannot have *aab*.

   We have that that $x = x_1x_2$ with x_1 matching $(\mathbf{ab}+\mathbf{b})^*$ and x_2 matching $(\lambda+\mathbf{a})^*$. Based on that here are some easy observations.

   - The string x_1 cannot have *aab* in it;
   - The string x_1 ends in a b;
   - The string x_2 cannot have aab in it.

Now, if $x = x_1x_2$ were to have *aab* in it, it would have to be in x_1 (but we showed it can't) or in x_2 (but we showed it can't) or the *aab* would have to straddle x_1 and x_2 (but that can't happen because x_1 ends in *b*).

So $x = x_1x_2$ cannot have *aab* in it.

2. Next suppose x fails to contain *aab*. We want to show that x can be matched by E. We do this by induction on the length of x.

   - if x is λ: then it is matched by E (easy)
   - if x starts with b : then $x = by$ for some string y. Since x doesn't contain *aab*, y doesn't either. Since y is shorter than x, induction says it is matched by E. Since b is matched by $(ab+b)$, by is matched by E (intuitively, add one more iteration of $(ab+b)$ to the front of the match for y).
   - if x starts with a : we need two subcases
     - if x starts with ab : the $x = aby$ for some string y. Then as in the previous case, the y is matched by E by induction, the ab is matched by $(\mathbf{ab}+\mathbf{b})$, and so aby is matched by E.
     - if x is just a, or starts with aa : then there can be no b anywhere in x ! So x is a string of a, which is matched by $(\lambda+a)^*$, and so is matched by E.

10.4 Algorithms over Regular Expressions

One thing that is nice about regular expressions as compared with automata is that regular expressions are defined *inductively*.

- There are two base cases : **O** and **c**
- Then there are three inductive cases: concatenation, union, Kleene star

Every regular expression has a **unique construction** of this form. This means we can do algorithms and reasoning *inductively*. This is the main advantage of regular expressions over automata

To make this explicit, consider the *RegExp* E:

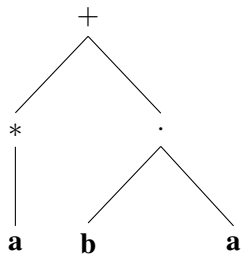
$$\mathbf{a}^* + \mathbf{ba}$$

This E has the following structure:

- It is a union: the arguments of the union are $\mathbf{a}^*$ and **ba**.
- The left argument $\mathbf{a}^*$ is a star: the single argument of that star is **a**.

- The right argument is a concatenation: the two arguments of that concatenation are **b** and **a**
- Each of **a** and **b** is a base case.

All those words are suggested by a picture:



Now the key idea is that to process a regular expression, we just walk the tree, processing it from the top down, **recursively**

Let's do a very simple example. Consider the following problem. Please do not object that you could solve this problem just by processing the input regular expression as a string. We are picking a simple example to make you familiar with the technique.

RegExp CountLetters

INPUT: a regular expression E

OUTPUT: the number of occurrences of alphabet symbols in E

Here is a recursive algorithm to compute this function.

Algorithm 7: CountLetters

Algorithm Reverse (E) :=
case $E = \mathbf{O}$: return 0
case $E = \mathbf{c}$ for $c \in \Sigma$: return 1
case E is $E_1 + E_2$: return $\mathsf{CountLetters}(E_1) + \mathsf{CountLetters}(E_2)$
case E is $E_1 E_2$: return $\mathsf{CountLetters}(E_1) + \mathsf{CountLetters}(E_2)$
case E is $(E_1)^*$: return $\mathsf{CountLetters}(E_1)$

10.4 Check Your Reading. *Make up some small regular expressions and trace through the computation of the above algorithm by hand. Don't skip steps!*

By the way, notice that when processing expressions by algorithms we have to—formally—remember that concatenation is a *binary* operation. So we can't pretend as we usually do that an expression like **abc** is a three-way concatenation, we have to treat it officially as (**ab**)**c** or as **a**(**bc**). That's not a choice the algorithm has to make: the actual precise (non-human-readable) input will be one or the other.

Here is a slightly more interesting example.

RegExp Reverse

INPUT: a regular expression E

OUTPUT: a regular expression E' such that $L(E') = L(E)^R$

That is, E' is supposed to represent the reverses of the strings in $L(E)$. Our goal is to write an algorithm to compute E' from E.

Here is an utterly straightforward recursive algorithm to walk the structure of the input.

Algorithm 8: RevRegExp

Algorithm Reverse (E) :=
case $E = \mathbf{O}$: return $\mathbf{O}$
case $E = \mathbf{c}$ for some $c \in \Sigma$: return $\mathbf{c}$
case E is $E_1 + E_2$: return $\mathsf{RevRegExp}(E_1) + \mathsf{RevRegExp}(E_2)$
case E is E_1E_2 : return $(\mathsf{RevRegExp}(E_2))\ (\mathsf{RevRegExp}(E_1))$. (Note the reversing of the subscripts!)
case E is $(E_1)^*$: return $(\mathsf{RevRegExp}(E_1))^*$.

10.5 Check Your Reading. *Make up some small regular expressions and trace through the computation of the above algorithm by hand. Don't skip steps!*

Suggestions: $(\mathbf{a}(\mathbf{ab}))^*$

$(\mathbf{a}+\mathbf{b})^*(\mathbf{a}(\mathbf{bc}))$

10.6 Check Your Reading. *Why didn't we reverse the subscripts in the union case, as we did in the concatenation case? Would we have been wrong if did so?*

Since, as we will show, regular expressions represent the same languages as automata, one can imagine a machine-oriented proof that involves reversing the arrows in automata (see Problem 113). But working with regular expressions directly is way easier.

10.5 Equations between Regular Expressions

In Example 5.21 we noted some equations that hold between languages. Note that the operations there—union, intersection, concatenation, and Kleene star—are the operations that regular expressions talk about. So if we rewrite a regular expression E_1 into another regular expression E_2 using the equations corresponding to the laws in Example 5.21 then we can be sure that $L(E_1) = L(E_2)$.

For example we know that the two regular expressions $\mathbf{a}^*(\mathbf{b}+(\mathbf{ac}))$ and $\mathbf{a}^*\mathbf{b}+\mathbf{a}^*\mathbf{ac}$ define the same language, due to the general language equality $X(Y \cup Z) = XY \cup XZ$.

The question of *algorithmically deciding* whether two regular expressions denote the same language is interesting. We'll be able to give an algorithm for solving this problem eventually. For now just pause a minute to see for yourself that it is not all obvious how one might tackle this question, naively. See Problem 126 for example.

It is standard to overload the equality symbol and write $E_1 = E_2$ to mean $L(E_1) = L(E_2)$.

10.7 Example. Let prove that $(\mathbf{ab})^*\mathbf{a} = \mathbf{a}(\mathbf{ba})^*$. It suffices to prove that for each k, $(\mathbf{ab})^{\mathbf{k}}\mathbf{a} = \mathbf{a}(\mathbf{ba})^{\mathbf{k}}$. We prove that fact by induction on k.

case $k = 0$ we simply have to show that $\lambda\mathbf{a} = \mathbf{a}\lambda$, which is immediate.

Next we want to show that $(\mathbf{ab})^{\mathbf{k+1}}\mathbf{a} = \mathbf{a}(\mathbf{ba})^{\mathbf{k+1}}$; our induction hypothesis is that $(\mathbf{ab})^{\mathbf{p}}\mathbf{a} = \mathbf{a}(\mathbf{ba})^{\mathbf{p}}$ for any $p \leq k$.

$$\begin{aligned}
(\mathbf{ab})^{\mathbf{k+1}}\mathbf{a} &= (\mathbf{ab})^{\mathbf{k}}(\mathbf{ab})\mathbf{a} \\
&= (\mathbf{ab})^{\mathbf{k}}\mathbf{a}\,(\mathbf{ba}) \\
&= \mathbf{a}(\mathbf{ba})^{\mathbf{k}}\,(\mathbf{ba}) \qquad \text{by induction hypothesis} \\
&= \mathbf{a}(\mathbf{ba})^{\mathbf{k+1}}
\end{aligned}$$

The essential thing that makes this work is that $(\mathbf{ab})\mathbf{a} = \mathbf{a}(\mathbf{ba})$. Problem 127 asks you to confirm that intuition.

10.6 Some Languages Cannot be Named by Expressions

Every regular expression names a language, but it is **not true** that every language can be described by a regular expression.

Regular expressions describe the languages you get using union, concatenation, and Kleene-star **starting only with the empty language and singleton languages.** That is, we start with the simplest possible languages we can understand: $\emptyset, \{a\}$, and $\{\lambda\}$, then see what we can get by building up from these using the "regular operations" of union, concatenation, and Kleene-star.

If you studied the cardinality section of these notes, or are otherwise familiar with cardinality, you will believe this is true: there are uncountably many languages but only countably many regular expressions.

This is not so satisfying, since it would be nice to have concrete examples of languages that cannot be described by regular expressions.

Here are two examples.

- Let E be the set of all finite bitstrings with an equal number of 0s and 1s.
- Let $P = \{10, 11, 101, 111, \ldots\}$ be the set of bit strings coding a prime number.

Each of E and P is a perfectly good language, but there is no regular expression you can write down that matches either of them. We'll be in a position to prove such facts soon.

10.7 From Regular Expressions to Automata

In this chapter we fulfill half of our promise to show that finite automata recognize precisely the languages denoted regular expressions: we show how to build a finite automaton recognizing the language of any given regular expression.

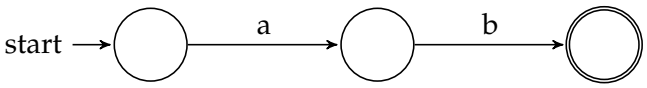
10.7.1 From Regular Expressions to $NFA_\lambda s$

We've done all the work to show that NFA_λ can recognize anything regular expressions can denote: it's just a matter of assembling all the constructions above into a recursive algorithm, Algorithm 9.

10.8 Lemma. *If E is a regular expression then Algorithm 9 constructs an NFA_λ M with $L(M) = L(E)$.*

Proof. The correctness of Algorithm 9 follows from the correctness of the sub-algorithms it calls, which we have proven. ///

10.9 Example. Let's make an *NFA* for the *RegExp ab*. This expression is so simple that you could build an *NFA* by intuition...



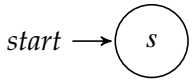
...but to make a point we will build it mechanically as in Algorithm 9.

Start with *NFAs* for a and for b (these are base cases in Algorithm 9).

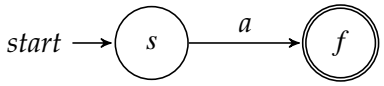
Join the *NFAs* with an empty transition (pay attention to what is accepting):

Algorithm 9: *RegExp* to NFA_λ

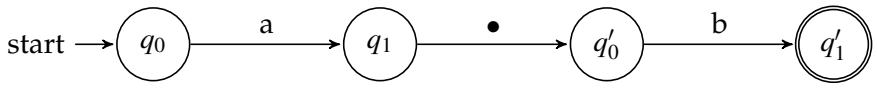
Input: a regular expression E
Output: an NFA_λ M such that $L(M) = L(E)$
The construction is recursive, and proceeds by cases on the form of E.
case *E is* **O***:* **do**
    return



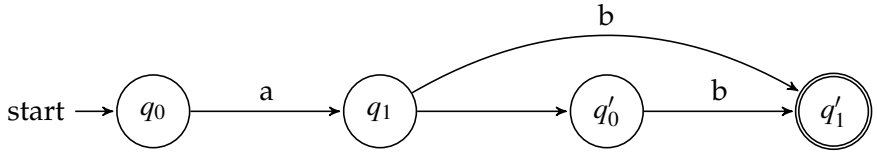
case *E is a:* **do**
    return



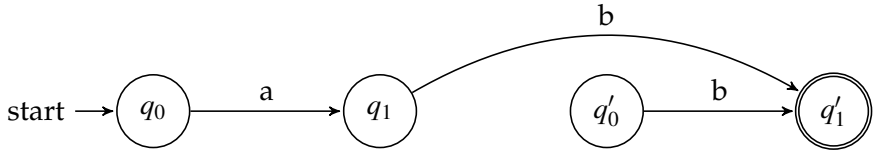
case *E is* $E_1 + E_2$*:* **do**
    Let M_1 := *RegExp*-to-NFA_λ (E_1) ;
    Let M_2 := *RegExp*-to-NFA_λ (E_2) ;
    return *the result of* NFA_λ*-Union (Algorithm 3) on* M_1, M_2
case *E is* E_1E_2*:* **do**
    Let M_1 := *RegExp*-to-NFA_λ (E_1);
    Let M_2 := *RegExp*-to-NFA_λ (E_2);
    return *the result of* NFA_λ*-Concatenation (Algorithm 4) on* M_1, M_2
case *E is* E_1^**:* **do**
    Let M_1 := *RegExp*-to-NFA_λ (E_1);
    return *the result of* NFA_λ*-Kleene-Closure (Algorithm 5) on* M_1



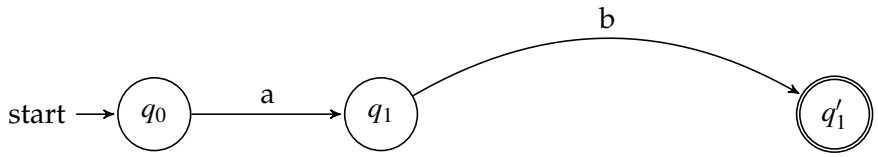
First step in eliminating empty transitions: add the new transitions (there is only one here):



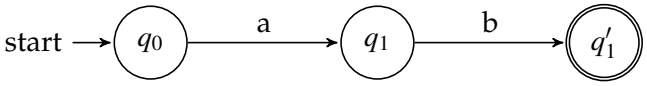
Second step: remove the empty transitions:



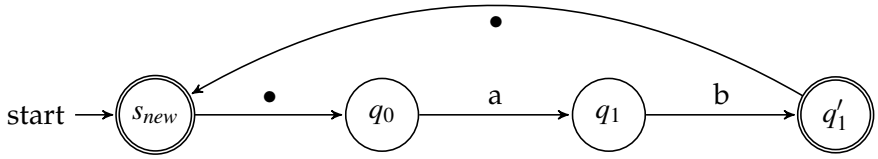
That's a correct answer, but notice that state q_0' is unreachable, so we might as well remove it. We end up with what your intuition knew in the first place.



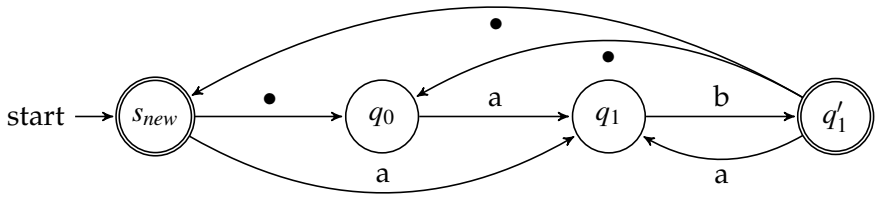
10.10 Example. Let's mechanically make an *NFA* for $(ab)^*$. Start with an *NFA* for ab (you can view this as a continuation of Example 10.9 if you like):



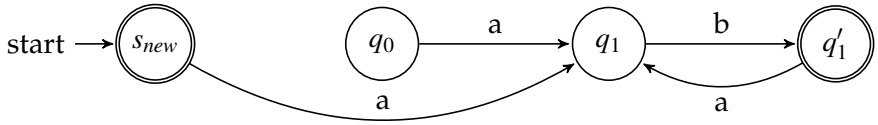
According to Algorithm 9 here is an NFA_λ for $(ab)^*$.



First step in eliminating empty transitions: add the new transitions and make q_1' accepting (since it has a λ-transition to the accepting s_{new})



Second step: remove the empty transitions



Finally we notice that state q_0 is unreachable, so we might as well delete it:

Maybe you would have come up with that on your own. Anyway the **most important thing** is this: you are not required to have any insight or cleverness construct an *NFA* from a *RegExp*. The process is completely mechanical and can be programmed. (And yes, the human optimizations we saw could be mechanized as well.)

10.7.2 Regular Expression Languages are Regular

Now we can say that any language denoted by a regular expression is regular, that is—by definition—is recognized by a *DFA*.

10.11 Theorem. *If E is a regular expression then there is a DFA M with $L(M) = L(E)$. Furthermore, there is an algorithm to build M from E.*

Proof. Algorithm 9 constructs an NFA_λ, Algorithm 6 converts that NFA_λ to an *NFA*, and Algorithm 2 converts that *NFA* to a *DFA*. ///

Perspective on Regular Closure Properties

We have proven the following facts. If A and B are accepted by *DFAs*, then so are

$$A \cap B \quad A \cup B \quad \overline{A} \quad AB \quad A^*$$

The variety of proof techniques used are worth noting. The first result was shown by using the product construction, which works naturally on either *NFAs* or *DFAs*. Closure under complementation required working with *DFAs*. Closure under union, concatenation and Kleene star required working with $NFA_\lambda s$, so we needed to know that $NFA_\lambda s$ were no more expressive than *NFAs* (Theorem 9.19).

It is not so clear that the converse of Theorem 10.11 holds. The languages accepted by automata are closed under intersection and complement; notice that in going from regular expressions to automata we only use the closure under union, concatenation, and Klene star. Maybe the operations of complement or intersection takes us out of the realm of regular expressions?

It is not at all clear, for example, that if A is a language named by a regular expression, then the complement $\overline{A}$ is also named by a regular expression. A similar remark holds for the intersection of two languages named by regular expressions.

In Section 11 we tackle the converse of Theorem 10.11 (and so get nice answers to all the the questions just raised).

10.8 Problems

119. RegExpMem

Let $\Sigma = \{a, b\}$.

For each given regular expression E and string x decide whether $x \in L(E)$. Defend your answer.

(This problem is partly just to get you familiar with thinking about regular expressions, but its devious secondary purpose is to persuade you that deciding matching for regular expressions seems to be a hard problem. So you will suitably impressed when you see fast algorithms to do it.)

a) E is $\mathbf{a}^*\mathbf{b}\mathbf{a}^*\mathbf{b}(\mathbf{a}+\mathbf{b})^*$

1. Is *ab* in $L(E)$?
2. Is *aaabbaaa* in $L(E)$?
3. Is *babab* in $L(E)$?

b) E is $\mathbf{a}^*(\mathbf{a}^*\mathbf{b}\mathbf{a}^*\mathbf{b}\mathbf{a}^*)^*$

1. Is *ab* in $L(E)$?
2. Is *aaabbaaa* in $L(E)$?
3. Is *babab* in $L(E)$?

c) E is $(\mathbf{b}+\mathbf{a}\mathbf{b})^*(\lambda+\mathbf{a})$

1. Is *ab* in $L(E)$?
2. Is *aaabbaaa* in $L(E)$?
3. Is *babab* in $L(E)$?

120. RegExpElt

For each of the following regular expressions E give a shortest non-λ string in $L(E)$. Assume that the alphabet Σ is always $\{a, b\}$. (There may be more than one correct answer to a given question...)

a) $\mathbf{aba}+\mathbf{bab}$

b) $(\lambda + \mathbf{a})\mathbf{b}$

c) $(\mathbf{a} + \mathbf{bb})\mathbf{a}^*\mathbf{b}$.

d) $(\mathbf{bb} + (\mathbf{ab} + \mathbf{ba})(\mathbf{aa} + \mathbf{bb})^*(\mathbf{ab} + \mathbf{ba}))^*$

e) $((\mathbf{aa} + \mathbf{bb})^* + (\mathbf{aab} + \mathbf{bba})^*)^*$

f) $\mathbf{a}^*\mathbf{b}^*$

g) $\mathbf{a}(\mathbf{ba})^*\mathbf{b}$

h) $\mathbf{a}^* + \mathbf{b}^*$

i) $(\mathbf{aaa})^*$

121. BuildRegExp

Let $\Sigma = \{a, b\}$. For each language $K \subseteq \Sigma^*$ described below, construct a regular expression E with $L(E) = K$.

a) The set of strings of odd length.

b) $\{x \mid x$ does not contain $ba\}$

c) $\{x \mid x$ has odd length and contains $ba\}$

d) $\{x \mid x$ does not contain aaa $\}$

e) $\{x \mid x$ contains aa at least twice$\}$ (Be careful about the string aaa: we do want to include this.)

f) The set of strings that contain exactly three occurrences of b.

g) The set of strings starting with a and ending with bb.

h) The set of strings that contain the substring abb.

122. RegExpIdentifiers

Let $\Sigma = \{a, b, 1, 2, -\}$. (Think of this little Σ as standing in for a richer alphabet that has all the digits 0–9, and all the letters).

Pick a programming language you know, inspired by that, make up a rule for what strings should be considered legal identifiers. Write a regular expression that matches precisely the legal identifiers.

123. RegExpPasswords

Let $\Sigma = \{a, b, A, B, 1, 2\}$. (Think of the little Σ as standing in for a richer alphabet that has all the digits 0–9, and all the letters).

Make up a rule for acceptable passwords over this alphabet. Write a regular expression that matches precisely the legal passwords.

124. RegExpFloats

Let $\Sigma = \{0, 1, 2, \ldots 9, -, .\}$. Think of the "-" as being the minus sign and the "." as the dot in a floating point number.

Write a regular expression matching the strings that denote a floating-point number. If you have any questions about what should be legal (such as "-0" or whatever) do not fret about these. Just make up a rule that is reasonable and keep going. The point is just to see how regular expressions can be used to make such definitions.

125. RegExpComplClever

We discussed the fact that although the syntax for regular expressions does not explicitly provide for intersection nor for complement, it is an amazing fact that these operations can be simulated—on a case-by-case basis—by pure regular expressions as we have defined them. We'll prove that later, but in this problem you are asked to anticipate this general result, by doing some specific examples (just by being clever).

1. For each of the languages in Examples 10.2, give a regular expression for the complement of that language.
2. For each pair of languages in Examples 10.2, give a regular expression for the intersection of those languages.

Some of these might be hard. If you get stumped, that's ok: as we develop the theory of regular expressions we will see how to solve problems like this algorithmically, that is, without having to be clever!

126. RegExpCompare

For each pair of regular expressions below, there are four possible relationships between them:

(i) they denote the same language

(ii) the language denoted by the first expression is a proper subset of the language denoted by the second expression

(iii) the language denoted by the second expression is a proper subset of the language denoted by the first expression

(iv) neither of the languages denoted is a subset of the other

State which of these is true for each pair. If (i) holds give a proof. If (ii) or (iii) holds, give a string in one set and not the other. If (iv) holds, give two strings showing that.

a) $\mathbf{O}^*$ and λ^*

b) $(\mathbf{a}+\mathbf{b})^*$ and $\mathbf{a}^*+\mathbf{b}^*$

c) $(\mathbf{a}^*\mathbf{b})^*$ and $(\mathbf{a}^*\mathbf{b}^*)^*$

d) $(\mathbf{ab}+\mathbf{a})^*$ and $(\mathbf{ba}+\mathbf{a})^*$

e) $\mathbf{a}^*\mathbf{ba}^*\mathbf{b}(\mathbf{a}+\mathbf{b})^*$ and $(\mathbf{a}+\mathbf{b})^*\mathbf{b}(\mathbf{a}+\mathbf{b})^*\mathbf{b}(\mathbf{a}+\mathbf{b})^*$

f) $\mathbf{a}^*\mathbf{ba}^*\mathbf{b}(\mathbf{a}+\mathbf{b})^*$ and $\mathbf{a}^*(\mathbf{a}^*\mathbf{ba}^*\mathbf{ba}^*)^*$

g) $(\mathbf{ab})^*\mathbf{a}$ and $\mathbf{a}(\mathbf{ba})^*$

h) $(\mathbf{bba})^*\mathbf{bb}$ and $\mathbf{bb}(\mathbf{abb})^*$

i) $\mathbf{a}(\mathbf{bca})^*\mathbf{bc}$ and $\mathbf{ab}(\mathbf{cab})^*\mathbf{c}$

127. RegExpSwap

We build on Example 10.7. Let Σ be any alphabet and choose some symbol, we might as well use $\mathbf{a}$.

What is really going on in Example 10.7 is that the funny property that $X\mathbf{a} = \mathbf{a}Y$ is actually rather robust. Specifically, this $X\mathbf{a} = \mathbf{a}Y$ relationship between X and Y is preserved by taking unions, concatenations, and Kleene star.

a) For starters, find three concrete pairs of expressions X and Y satisfying $X\mathbf{a} = \mathbf{a}Y$.

b) Show that if X, X', Y, Y' satisfy $X\mathbf{a} = \mathbf{a}Y$ and $X'\mathbf{a} = \mathbf{a}Y'$, then in fact $(X+X')\mathbf{a} = \mathbf{a}(Y+Y')$.

c) Show that if X, X', Y, Y' satisfy $X\mathbf{a} = \mathbf{a}Y$ and $X'\mathbf{a} = \mathbf{a}Y'$, then in fact $(XX')\mathbf{a} = \mathbf{a}(YY')$.

d) Prove that if X and Y are regular expressions satisfying $X\mathbf{a} = \mathbf{a}Y$, then in fact $X^*\mathbf{a} = \mathbf{a}Y^*$.

e) Prove that $(\mathbf{ab}+\mathbf{a})^*\mathbf{a}$ and $\mathbf{a}(\mathbf{ba}+\mathbf{a})^*$ denote the same language. (This should be easy now)

128. RegExpShortest

Write a recursive algorithm to compute a shortest string in a given *RegExp* E or FAIL if $L(E)$ is empty.

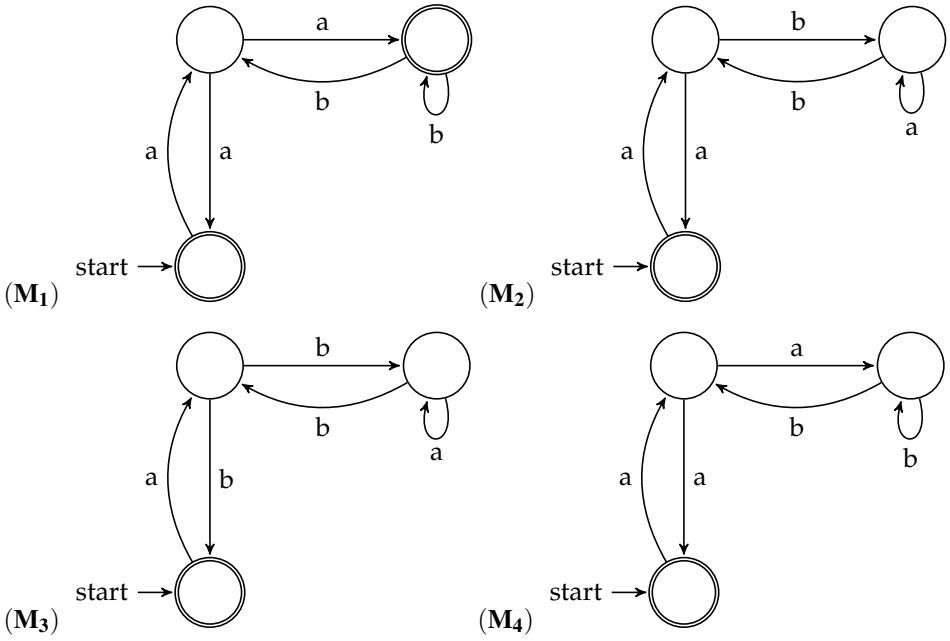
> *RegExp Shortest*
>
> INPUT: a regular expression E
>
> OUTPUT: a string x of minimal length in $L(E)$, or FAIL if $L(E) = \emptyset$

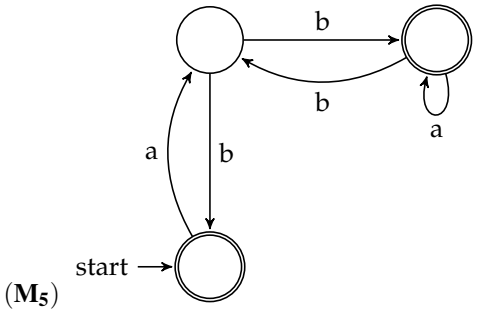
Use the approach in Section 10.4.

129. MatchRegExpFA 1

Eventually we will show that a language can be named by a regular expression if and only if it can be accepted by an *NFA*, and indeed we will show that there are algorithms to translate back and forth. This problem and the next are less ambitious, they ask you to explore the correspondence intuitively, to build your intuitions.

Match each *NFA* with an equivalent regular expression. (From [Koz97].)



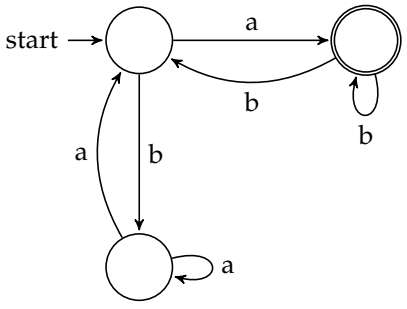


($\mathbf{M_5}$)

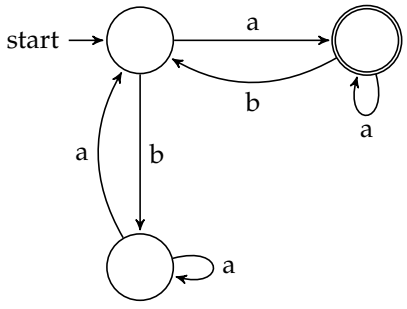
- $\lambda + \mathbf{a}(\mathbf{ab}^*\mathbf{b} + \mathbf{aa})^*\mathbf{ab}^*$
- $\lambda + \mathbf{a}(\mathbf{ba}^*\mathbf{b} + \mathbf{ba})^*\mathbf{ba}^*$
- $\lambda + \mathbf{a}(\mathbf{ba}^*\mathbf{b} + \mathbf{aa})^*\mathbf{a}$
- $\lambda + \mathbf{a}(\mathbf{ab}^*\mathbf{b} + \mathbf{aa})^*\mathbf{a}$
- $\lambda + \mathbf{a}(\mathbf{ba}^*\mathbf{b} + \mathbf{ba})^*\mathbf{b}$

130. MatchRegExpFA 2

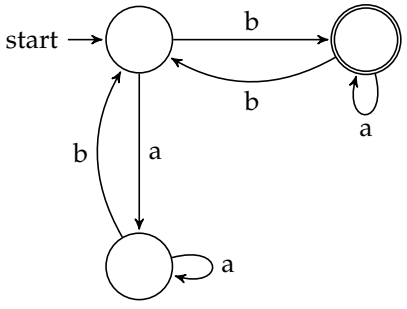
For each *NFA N* below, give a regular expression E such that $L(E) = L(N)$.



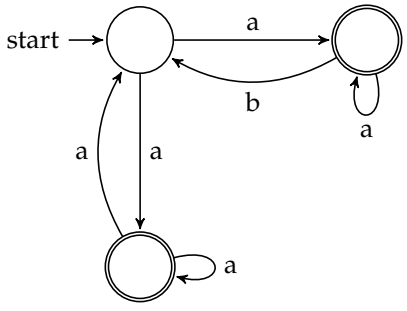
($\mathbf{N_1}$)



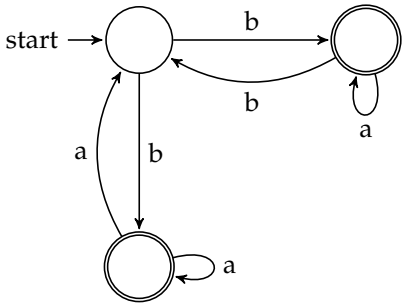
($\mathbf{N_2}$)



($\mathbf{N_3}$)



($\mathbf{N_4}$)



($\mathbf{N_5}$)

131. RegExpToNFA

For each regular expression E below, build an *NFA* recognizing the language $L(E)$. (First build an NFA_λ then eliminate λ-transitions.)

Feel free to be clever in building your NFA_λ, take shortcuts. If you follow the official algorithm the automata get annoyingly huge.

a) $(01+011+0111)^*$

You should be able to construct an *NFA N* with four states.

b) $(00+11)^*(01+10)(00+11)^*$

c) $(000)^*1+(00)^*1$

d) $(0(01)^*(1+00)+1(10)^*(0+11))^*$

132. RegExpCountable

Prove that the set of all regular expressions over the alphabet $\{0,1\}$ is countable.

This question is slightly subtle. Make sure you understand that a regular *expression* is a finite sequence of ASCII symbols. Don't confuse a regular *expression E* with the language (set of strings) that *E* denotes!

Chapter 11

From Automata to Regular Expressions

In this section we complete the story of the equivalence among *DFA*, *NFA*, regular grammars, and regular expressions, by showing the one remaining simulation result. Our goal is to prove the following theorem.

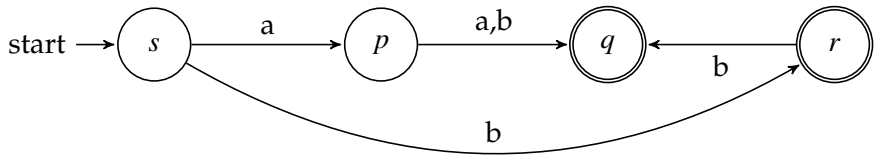
11.1 Theorem. *Given any NFA M we can construct a regular expression E such that $L(E) = L(M)$.*

We prove the theorem by describing an algorithm to derive E from M. Here's the idea.

1. For each **state** q we introduce a **set variable** Q to stand for the language which is the set of those strings w that take q to an accepting state.
2. We view the transition rules of the *NFA* as **equations** which define each Q. A subtlety is that such equations won't necessarily have unique solutions, just like ordinary equations over numbers typically don't. But they will have (unique) **minimal** solutions, and these are precisely what we intend when we write our equations.
3. We use ordinary algebraic techniques to solve these equations, ultimately arriving at a regular expression for S. We use Arden's Lemma, in the next section, to handle the case where an equation is "recursive."

11.1 Using Equations to Capture *NFAs*

11.2 Example. Here is a first easy example, which shows off the substitution techniques but doesn't require Arden's Lemma.



We get the following equations.

$$\begin{aligned} S &= aP + bR \\ P &= aQ + bQ \\ Q &= \lambda \\ R &= bQ + \lambda \end{aligned}$$

We want to solve for S. We work back to it, using substitution. Substituting for Q:

$$\begin{aligned} S &= aP + bR \\ P &= a\lambda + b\lambda \\ R &= b\lambda + \lambda \end{aligned}$$

But since λ is the identity for concatenation, we get to

$$\begin{aligned} S &= aP + bR \\ P &= (a+b) \\ R &= b + \lambda \end{aligned}$$

Substituting for R:

$$\begin{aligned} S &= aP + b(b+\lambda) \\ P &= (a+b) \end{aligned}$$

One more substitution and we get

$$S = a(a+b) + bb + b$$

So our answer is the regular expression $\mathbf{a(a+b)+bb+b}$

Now, you can see that the last answer is correct just by looking at the machine. In fact you could probably have seen this answer from the beginning, without all this machinery. But the virtue of the method we are developing is that it is completely algebraic, works on automata of any size, and does not require any creative insights.

Regular expression identities are your friends When doing these little calculations you can always replace some expression by an equivalent one, if you feel like it will make things simpler. In actual practice there is one fact that gets frequently used: the fact that concatenation distributes over union. This is expressed by the equations:

$$E(F+G) = EF + EG \qquad \text{and} \qquad (F+G)E = FE + GE$$

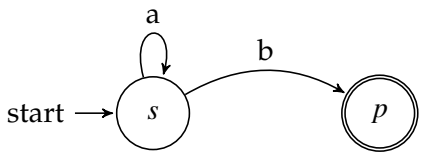
It is sometimes a matter of taste whether it is clearer to write an expression in compact form like $E(F+G)$ or in exploded form like $EF+EG$. Just do what feels good.

Boldface or not? By the way, why did we switch to boldface when we declared our answer? It's just a reflection of our notational convention to write concrete regular expressions in boldface. During the intermediate stages of the equation-solving we were not dealing with official regular expressions really, just these hybrid expressions with alphabet symbols.

In fact, as you have perhaps already noticed, many (most?) people don't even bother to use fonts to distinguish alphabet symbols (like a) from regular expressions (like **a**), trusting that the reader will figure out what is meant based on the context of use. There is no harm in this, once everyone is sophisticated enough to know what type of things is being talked about at any given moment.

Back to Work The technique we just used can only take us so far, though.

11.3 Example. Consider the following NFA



A regular expression for the language this NFA accepts is obviously a^*b. But we can't find that answer using the simple substitution technique above. We start with the equations

$$S = aS + bP$$
$$P = \lambda$$

which simplify to

$$S = aS + b$$

But we can't mechanically eliminate S since it occurs on both sides of the equation.

We need a new idea.

11.2 Arden's Lemma

Arden's Lemma is from [Ard61].

If you prefer, you can skip the proof of Arden's Lemma on first reading, just learn what it says and skip to seeing how it is used below.

11.4 Lemma (Arden's Lemma)**.** *Let A and B be any languages. Then the equation $X = AX + B$ has $X = A^*B$ as a minimal solution.*

Proof.

A^*B is a solution: This means verifying that $A^*B = A(A^*B) + B$. The left-hand side, A^*B is the union of all A^kB taken over all $k \geq 0$. On the right-hand side:

- the term $A(A^*B) = (AA^*)B$ is the union of all A^kB taken over all $kgeq1$, and
- the term B is A^0B

Thus the right-hand is also the union of all A^kB taken over all $kgeq0$. So the left and right sides are equal.

A^*B is minimal: To show minimality means to show that that A^*B is a subset of every solution.

So let C be any language satisfying $C = AC + B$. We want to show that $A^*B \subseteq C$. What we will use from our assumption is the fact that $AC + B \subseteq C$, which entails that (i) $B \subseteq C$, and (ii) $AC \subseteq C$.

Now it suffices to show that for every k we have $A^kB \subseteq C$. We'll do this by induction on k. When $k = 0$ this means $B \subseteq C$, which is (i) above. When $k > 0$ we want to show that $AA^{k-1}B \subseteq C$. By induction hypothesis $A^{k-1}B \subseteq C$. Concatenating each side by A we get $AA^{k-1}B \subseteq AC$. But $AC \subseteq C$ by (ii), so indeed we get $AA^{k-1}B \subseteq C$.

///

You may be wondering about the phrase "minimal solution." Well, consider the equation $X = AX + B$, and suppose that A happened to have λ as an element. Then there could be lots of solutions to the equation. For example, taking X to be the set of all strings yields a solution. But since we set up our variables A, B, X, etc to describe *only* those strings that carry a state to an accepting state, a non-minimal solution to the equations we started with would not capture that idea. In other words, it really is the *minimal* solutions to these equations that we seek in the first place.

In Problem 138 you are asked to prove that λ is the only troublemaker in this story: that is, if λ is not in A then AB is the *only* solution to $X = AX + B$.

11.3 Using Arden's Lemma: First Steps

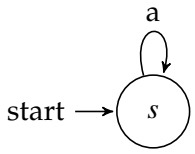
Next are some examples that do use Arden's Lemma but, since they have only one state, don't require the substitution technique.

11.5 Example. Return to the NFA from the previous example, where we arrived at the equation

$$S = aS + b$$

Arden's Lemma immediately yields $S = a^*b$, which matches the answer we get by just looking at the NFA. Good.

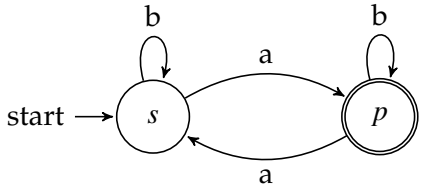
11.6 Example. As another baby example, consider this NFA, which accepts no strings at all:



Here the equation we would write down is $S = aS$. (Note that s is not accepting!) That is, the (implicit) B in the framework of Arden's Lemma is $\emptyset$. So Arden's Lemma would yield the regular expression $\mathbf{a}^*\mathbf{O}$. And indeed $a^*\emptyset = \emptyset$.

11.4 The General Technique: Arden's Lemma and Substitution

Here's a more interesting M; we'll have to use substitution as well as Arden. Note that $L(M)$ is the language of strings with an odd number of as.



This time we get the equations

$$S = aP + bS$$
$$P = aS + bP + \lambda$$

We want to solve for S. At our very first step, eliminating the variable P, we see that P occurs on the right-hand side of its own definition. This is where Arden's Lemma comes in handy.

We can rearrange the equation for P as

$$P = bP + (aS + \lambda) \tag{11.1}$$

Now we can apply Arden's Lemma for the variable P with the "Q" being the language denoted by $(aS + \lambda)$. Arden's Lemma says that

$$P = b^*(aS + \lambda) \tag{11.2}$$

Simplifying just a little bit, we get

$$P = b^*aS + b^* \tag{11.3}$$

Now substitute this last expression for P back into that for S and combine terms:

$$\begin{aligned} S &= a(b^*aS + b^*) + bS && (11.4)\\ &= (ab^*a)S + ab^* + bS && (11.5)\\ &= (ab^*a + b)S + ab^* && (11.6) \end{aligned}$$

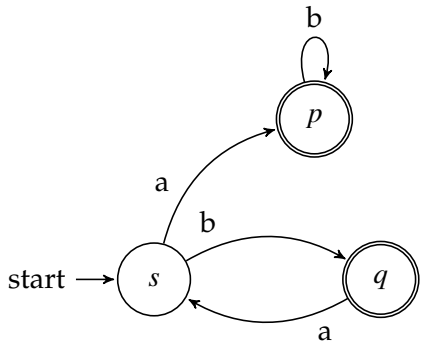
One more application of Arden's Lemma and we're done:

$$S = (ab^*a + b)^*ab^* \tag{11.7}$$

Convince yourself that this regular expression really does define the set of strings with an odd number of as.

Several Accepting States There is nothing special needed to deal with *FAs* with more than one accepting state.

11.7 Example.



The equations are:

$$S = aP + bQ$$
$$P = bP + \lambda$$
$$Q = aS + \lambda$$

Note that the only effect of having more than one accepting state is that we have more than one equation that mentions λ.

Applying Arden's Lemma to the equation for P we derive $P = b^*\lambda = b^*$. Substituting for P and for Q in the equation for S we get

$$S = ab^* + b(aS + \lambda) \quad = baS + (ab^* + b)$$

A final application of Arden's Lemma yields our answer:

$$S = (ba)^*(ab^* + b)$$

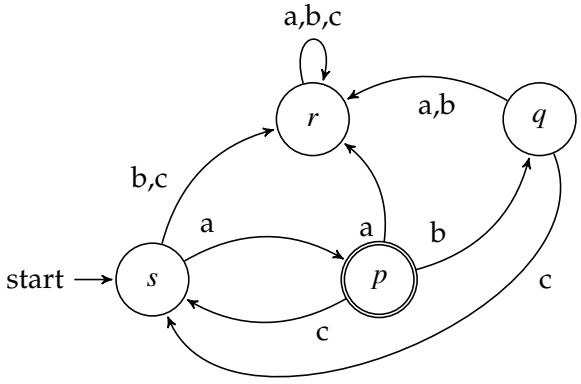
About λ-transitions Suppose the original NFA has λ-transitions? There's nothing special to do in this case.

11.8 Example. Suppose we have an NFA_λ with the corresponding equation below (you can draw the NFA_λ for yourself):

$$S = P + aQ$$
$$P = \lambda$$
$$Q = R$$
$$R = bS + aR$$

So $R = a^*bS$. Thus $Q = a^*bS$. Substituting, we get $S = \lambda + aQ = \lambda + a(a^*bS)$. Rearranging, we get $S = (aa^*b)S + +\lambda$, which yields $S = (aa^*b)^*$

11.9 Example. This example is adapted from Robin Milner's beautiful little book on concurrency: *Communicating and Mobile Systems: the π-calculus*.



The equations are:

$$S = aP + (b+c)R \tag{11.8}$$
$$P = aR + bQ + cS + \lambda \tag{11.9}$$
$$Q = (a+b)R + cS \tag{11.10}$$
$$R = (a+b+c)R \tag{11.11}$$

Now, we can always proceed in a robotic manner but let's be sensitive to some simplifications. Note that R is the empty language! (You can see this by thinking about it for a moment, or by computing using Arden's Lemma: $R = (a+b+c)^*\mathbf{O}$, which is $\mathbf{O}$.

Having noted that $R = \mathbf{O}$ we can simplify the equations above.

$$S = aP \tag{11.12}$$
$$P = bQ + cS + \lambda \tag{11.13}$$
$$Q = cS \tag{11.14}$$

Now we can proceed as in the previous example. First substitute the 2nd equation into the first:

$$S = a(bQ + cS + \lambda) \tag{11.15}$$
$$= abQ + acS + a \tag{11.16}$$

Now substitute the 3nd equation into the first:

$$S = ab(cS) + acS + a \tag{11.17}$$
$$= (abc + ac)S + a \tag{11.18}$$

Now Arden gives us our answer:

$$S = (abc + ac)^* a \tag{11.19}$$

Of course there can be more than one regular expression capturing a given language. This is reflected in the fact that we have strategic choices we can make as we solve the equations.

11.10 Example. For this example we'll just start with the equations.

$$S = aQ + aR + \lambda$$
$$P = bS$$
$$Q = aP + bQ$$
$$R = bS + bP + \lambda$$

First we eliminate P. It is not defined recursively so there is no need for Arden's

lemma at this step. We get

$$\begin{aligned} S &= aQ + aR + \lambda \\ Q &= abS + bQ \\ R &= bS + bbS + \lambda \end{aligned}$$

Then eliminate R

$$\begin{aligned} S &= aQ + a(bS + bbS + \lambda) + \lambda \\ Q &= abS + bQ \end{aligned}$$

Now we want to eliminate Q. We use Arden first

$$Q = b^*abS$$

Then

$$S = ab^*abS + a(bS + bbS + \lambda) + \lambda$$

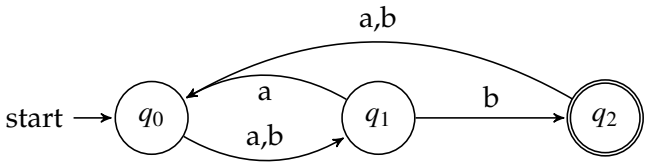
Collect terms, then use Arden

$$\begin{aligned} S &= (ab^*ab + ab + abb)S + a + \lambda \\ &= (ab^*ab + ab + abb)^*(a + \lambda) \end{aligned}$$

So the language accepted by our NFA is defined by the regular expression

$$(ab^*ab + ab + abb)^*(a + \lambda)$$

11.11 Example. Another example; this time we just give an answer, leaving it to you to work through the steps.



One answer is: $((a+b)(a+b(a+b)))^*(a+b)b$

Now, it may very well be that you arrive at an answer that looks different from this one. But remember that very different-looking regular expressions can denote the same language. So what you need to check is whether your answer is *equivalent* to the one above; it doesn't have to be identical.

11.5 Extended Regular Expressions Revisited

After our work above we are in a position to fulfill the promise we made in Section 10.3 to show that adding intersection and complement to regular expressions are each syntactic sugar, that is it does not change the expressive power of regular expressions.

11.12 Theorem. *Let E and F be regular expressions. Then*

- *There is a regular expression E' such that $L(E') = \overline{L(E)}$*
- *There is a regular expression D such that $L(D) = L(E) \cap L(F)$*

Proof. Here is an algorithm to build the regular expression E' for the first claim.

- From E build a *DFA* M such that $L(M) = L(E)$ (using the algorithms to go from regular expressions to $NFA_\lambda s$, then to an *NFA*, then to a *DFA*).
- From M build a *DFA* M' such that $L(M') = \overline{L(M)}$ (using the easy trick of swapping accepting and non-accepting states).
- From M' build the regular expression E' (using the technique of this section) such that $L(E') = L(M')$. This is the regular expression we seek.

The argument for intersection is exactly the same idea: move into "finite automata space" where intersection is easy (using the product construction) then move back into "regular expression space." The argument is a little easier, since the product construction—for intersection—works just fine even on *NFAs*. ///

The takeaway then, is that we *could* have added complement and intersection to our operations defining regular languages, without changing the expressive power. But it is usually good when formally defining a language to use a small core of operators and provide richer ones to your human users as syntactic sugar.

11.6 Perspective

At this point we have shown that regular expressions and finite automata have exactly the same expressive power in the sense that they determine the same set of languages.

Equations about Machines

The equivalence between *RegExp* and *FAs* means that we can mentally tack back and forth between two intuitively quite different intuitions. An *FA* is naturally viewed as a dynamic thing—it's a machine, after all—while a *RegExp* is an algebraic notation. If M is an *FA* and E is a *RegExp* denoting $L(M)$, we can view E as an algebraic specification of M's behavior. And the equalities between *RegExp* mean that we have a way of capturing "equalities" between machines.

This is one more example of the phenomenon recurring throughout these notes: having more than one way of thinking about a concept provides important insights.

11.7 Problems

133. FAToRegExpPractice1

a) Write a *RegExp E* such that $L(E)$ is the set of strings over $\Sigma = \{a,b\}$ with an odd number of bs, by starting with the relevant *DFA* in Example 8.5 and using the technique in this section.

b) Write a *RegExp E* such that $L(E_2)$ is the set of strings over $\Sigma = \{a,b\}$ ending in b, by starting with the relevant *DFA* in Example 8.5 and using the technique in this section.

c) Write a *RegExp E* such that $L(E)$ is the set of strings over $\Sigma = \{0,1\}$ such that the number they code in binary is divisible by 3. Start with the *DFA* in Example 6.9 and using the technique in this section.

(If you had been asked just after the introduction to regular expressions to construct an expression that captured divisibility mod 3 you would have considered it a wildly unfair question, right? But after this chapter, it is routine. Being able to pass back and forth between machines and expressions is pretty useful!)

134. FAToRexpPractice2

For each of the automata in Problem 130, construct equivalent regular expressions using our equation-solving technique; compare your answers with the expressions there.

Keep in mind that your answer for any given *FA* may be correct even if it is not identical with any of the expressions given in the list of choices there: but a correct answer will be *equivalent* to one of the given regular expression.

135. RegExpState

Let M be an *DFA*, with start state s, and let q be any state of M. Explain how to construct a regular expression for the set of all strings x such that $\hat{\delta}(s,x) = q$.

136. ArdenNotRecursive

What happens if we insist on applying Arden's Lemma to an equation that is not really recursive, *i.e.*, one that looks like $X = B$?

137. RegExpComplAlg

Go back to Problem 125. There you were asked to complement regular expression, by being clever. Some of those problems were not easy to do just by intuition. Do you see how, after the work in this section, each of the problems there now submit to an *algorithmic process* for deriving the answer?

To see how this works let's do an example about complement.

Fix the alphabet $\Sigma = \{a,b\}$. Let E be the regular expression $(a+b)^*(bb)(a+b)^*$. Obviously $L(E)$ is the language of strings containing bb as a substring. Suppose we want to write a regular expression E' for the $\overline{L(E)}$, the language of strings not containing bb as a substring. (This is easy enough so that you could probably do it by hand, but we are showing off the general method here.)

So construct E' as follows.

1. Make a *DFA* M for $L(E)$. If necessary, this can be done mechanically, by translating E to an NFA_λ mechanically and convert that NFA_λ mechanically to a *DFA*. This example is easy, so just build the *DFA* using human cleverness.
2. Make a *DFA* M' such that $L(M') = \overline{L(M)} = \overline{L(E)}$. Obviously this step is mechanical.
3. Make a *RegExp* E' such that $L(E') = L(M')$. This step is mechanical, using the techniques of this section.

138. ArdenUnique

Refer to the statement of Arden's Lemma. Prove that AB is the *unique* solution to $X = AX + B$ if $\lambda \notin A$.

Hint. Having proved that A^*B is a minimal solution in any event, it suffices to prove that if C is any solution, then $C \subseteq A^*B$.

Chapter 12

Proving Languages Not Regular

In this section we give a foolproof technique for showing that a language K is not regular. By "foolproof" we mean that a language K is regular if and only if it is passes the test. This doesn't mean that the test is always simple to apply, but it is always accurate! In mathematical jargon we say that it is a *characterization* of being regular.

You may have studied the "Pumping Lemma" in previous classes, as a means of showing languages to be non-regular. The technique we use here is very different. I think it is easier to apply. And it is superior to the Pumpling Lemma in one technical way: there are languages which are not regular but which cannot be shown to be non-regular by the Pumping Lemma. That is to say, the Pumping Lemma does not give a characterization of regularity.

Note that proving a language A to be non-regular is a somewhat amazing feat. It amounts to saying that no one can ever build a *DFA* accepting A. This is not a statement about the current state of human knowledge: it says that no matter how clever someone is, even a smart person who haven't been born yet, they cannot make a *DFA* M such that $L(M) = A$.

The strategy is to first isolate a certain nice property (Corollary 12.11) that all regular languages are guaranteed to have. After that, if we can show that some language A doesn't have that property, we conclude that it can't be regular.

We need a new idea in order to define that nice property.

12.1 The K-Equivalence Relation

Fix an alphabet Σ. Suppose K is any language over Σ (regular or not). In an obvious sense, K partitions the world Σ^* into two parts: the strings that are in K and the

strings that are not. That's boring. What we will do in this section is something more subtle and useful. We will show how any given language induces a different partition of Σ^*, into a number of classes, sometimes infinitely many, and this partition gives a lot of information about K.

As an example, consider the language $K = \{a^n b^m \mid n, m \geq 0\}$. Let x be the string aab and let y be the string a. Both x and y are in K. But if we now imagine concatenating certain strings z to each string, obtaining xz and yz, we see that x and y act differently. For example if we were to take z to be the string a, then $xz = aaba$ is not in K, while the string $yz = aa$ is in K. So the string a "separates" the two original strings aab and a with respect to K.

On the other hand if we were to take x to be aab as before and take y to be ab then there is no string z that will separate x and y: no matter what z is chosen, either xz and yz will both be in K or neither one of them will. (Check that for yourself before reading further.)

12.1 Definition. Let K be *any* language over alphabet Σ. Two strings x and y in Σ^* are *equivalent with respect to* K, or *K-equivalent*, written $x \equiv_K y$, if for every string z, $xz \in K$ if and only if $yz \in K$.

So x and y fail to be K-equivalent if there is some string $z \in \Sigma^*$ such that exactly one of xz and yz is in K. In this case we say that x and y are *K-inequivalent*, and we write $x \not\equiv_K y$.

Note that the relation $\equiv_K$ is defined without any reference to machines or grammars. It is a purely "combinatorial" idea, intrinsic to strings.

The K-Equivalence Classes

It is easy to show that $\equiv_K$ is an equivalence relation on Σ^*, that is, it is reflexive, symmetric, and transitive. Since $\equiv_K$ is an equivalence relation, it partitions Σ^* into classes. Let us write $[x]_K$ for the equivalence class of a string x. That is,

$$[x]_K = \{y \mid x \equiv_K y\}$$

This idea, partitioning the space of all strings into classes, or blocks, is the crucial idea of the whole chapter.

12.2 Check Your Reading. *Before you go any further, do Problem 139. Seriously.*

Counting Classes for a Language We will see later that the number of $\equiv_K$ equivalence classes of a language K is a very important measure of how "complicated" K is.

12.3 Definition. Let K be any language. The *index* of K is the number of $\equiv_K$ equivalence classes of K. If there are not finitely many classes we just say "the index of K is infinite."

12.2 Examples

This is a complicated idea so let's do lots of examples.

12.4 Example. Let $A = \{w \in \{0,1\}^* \mid w \text{ has length divisible by } 3\}$. There are three $\equiv_A$ classes:

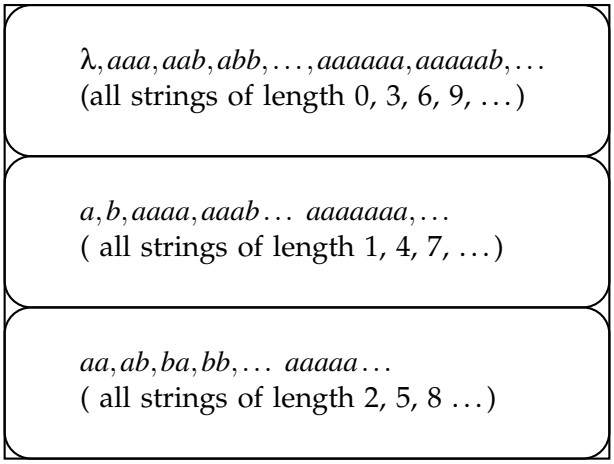
1. the set of strings whose length is equal to 0 mod 3
2. the set of strings whose length is equal to 1 mod 3
3. the set of strings whose length is equal to 2 mod 3

Note that strings in the first class happen to be in A, while no strings in the latter two classes lie in A.

The index of A is 3.

Here is a picture of the equivalence classes.

A is $\{x \in \{a,b\}^* \mid x \text{ has length divisible by } 3\}$



The $\equiv_A$ equivalence classes

12.5 Example. Let $B = \{a^n b^m \mid n, m \geq 0\}$. There are three $\equiv_B$ classes:

1. A class containing $\lambda, a, aa, \ldots$, ie all the strings of the form a^*
2. A class containing $b, ab, aab, aabb, \ldots$, ie all the strings of the form a^*bb^*
3. A class containing all other strings, ie all the strings of the form $(a \cup b)^* ba (a \cup b)^*$

Note that strings in the first two classes happen to be in B, while no strings in B live in the last class.

The index of B is 3.

12.6 Example. Let $C = \{a^n b^n \mid n \geq 0\}$. There are *infinitely many* $\equiv_C$ classes, that is, C has infinite index.

Proof. Consider the infinite collection of strings $\{a^n \mid n \geq 0\}$. Each of these strings is inequivalent to the other with respect to C. Since: for $i \neq j$ the strings a^i and a^j are separated by the string $z = b^i$. (Note by the way that the string $z = b^j$ would also separate them). ///

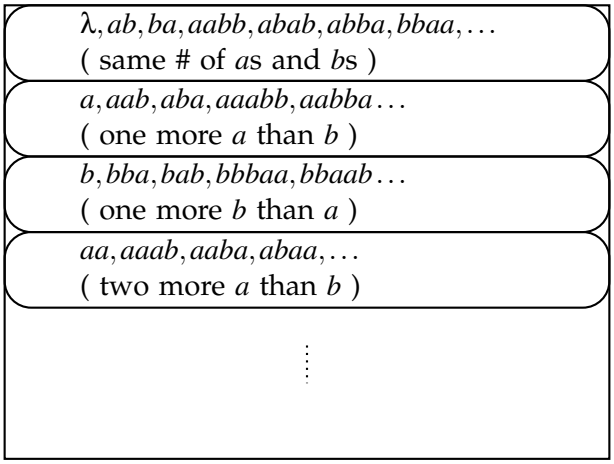
12.7 Example. Let $D = \{w \in \{a,b\}^* \mid$ w has an equal number of as and bs $\}$. Then D has infinitely many $\equiv_D$ classes, that is, D has infinite index.

Proof. Consider the infinite collection of strings $\{a^n \mid n \geq 0\}$. (Yes, this is the same collection we considered for the language $C \ldots$)

Each of these strings is inequivalent to the others with respect to D. Indeed the same argument as we used for C applies: for $i \neq j$ the strings a^i and a^j are separated by the string $z = b^i$. ///

Here is a picture of the equivalence classes.

D is $\{x \in \{a,b\}^* \mid x$ has the same number of as and bs$\}$



Infinitely many $\equiv_D$ equivalence classes!

12.8 Example. Let $Pal = \{x \in \{a,b\}^* \mid x = x^R\}$ There are infinitely many $\equiv_{Pal}$ classes, that is, *Pal* has infinite index.

Proof. Consider the infinite collection of strings $\{a^n b\ ;\mid n \geq 0\}$. Each of these strings is inequivalent to the other with respect to E. Since: for $i \neq j$ the strings $a^i b$ and $a^j b$ are separated by the string $z = a^i$. ///

12.9 Example. Let $Q = \{a^{n^2} \mid n \geq 0\}$, over the alphabet $\{a\}$. There are infinitely many $\equiv_Q$ classes.

Proof. In this case it happens that each class is a singleton, that is, for each pair of strings $x = a^{n^2}$ and $y = a^{m^2}$ there is a z which separatees them.

Let us suppose that $n < m$. Take z to be the string a^{2n+1}. We now argue that $xz \in Q$ while $yz \notin Q$.

Certainly $xz \in Q$ since $xz = a^{n^2}a^{2n+1} = a^{n^2+2n+1} = a^{(n+1)^2}$. To argue that $yz = a^{m^2}a^{2n+1} = a^{m^2+2n+1}$ is not in Q it suffices to argue that m^2+2n+1 cannot be a perfect square no matter what n and m are.Certainly it is greater than m^2. But it is also smaller than $(m+1)^2$, since $(m+1)^2 = m^2+2m+1$, and $2n+1 < 2m+1$.

So we have found our infinite collection of pairwise Q-inequivalent strings. ///

12.3 Regular Languages Have Finitely Many Classes

Recall that if M is a *DFA*, for every string x there is a single run of M on x: when s is the start state we have $s \overset{x}{\longrightarrow\!\!\!\!\!\rightarrow} p$ for exactly one state p.

Note that this notation would not make any sense if M were an *NFA* not a *DFA*, since $\hat{\delta}$ would not behave like a function.

The following lemma is easy to prove but it turns out to be crucially important in determining which languages are regular.

12.10 Lemma (K-Equivalence Lemma)**.** *Let K be a* regular *language, and let M be a DFA recognizing K. If $\hat{\delta}_M(s,x) = \hat{\delta}_M(s,y)$ then $x \equiv_K y$.*

Proof. Suppose $\hat{\delta}_M(s,x) = \hat{\delta}_M(s,y)$. Let z be any string; we want to show that $xz \in K$ if and only if $yz \in K$. It's enough to show that M takes xz and yz to the same state of M. But $\hat{\delta}_M(s,x) = \hat{\delta}_M(s,y)$ simply means that M takes x and y to the same state of M. So certainly M takes xz and yz to the same state of M. This says that $x \equiv_K y$. ///

Note that the this lemma is equivalent to saying that whenever M is a *DFA* recognizing language K then if $x \not\equiv_K y$ then M takes x and y to different states. So we have

12.11 Corollary. *If K is a regular language then the number of $\equiv_K$ equivalence classes is finite.*

In fact, if M is a DFA recognizing K then the number of $\equiv_K$ equivalence classes is less than or equal to the number of states of M.

Proof. This follows easily from Lemma 12.10. ///

Now we easily have

12.12 Corollary. *Let K be a language, and suppose there is an infinite collection $x_1, x_2, \ldots$ of strings such that any two of them are K-inequivalent. Then K is not regular.*

Proof. If there were a *DFA* M recognizing K then runs of M would have to take each x_i to a different state. This is impossible since M can have only finitely many states. ///

So now we know that each of the languages K for which we found infinitely many $\equiv_K$-classes is non-regular, by Corollary 12.12.

It's also true, by the way, that each of the languages there for which there were only finitely many $\equiv$-classes is regular. This is not a coincidence, as we will see in Section 15. But for now, Problem 142 should be very helpful for your intuition.

12.4 Using K-Equivalence to Prove Languages Not Regular

Let's see how to use Corollary 12.12 for proving languages to be *not* regular.

These proofs all follow a common script.

1. Based on K, we define a set X of infinitely many strings.
2. Then we give an argument that shows that for any two strings chosen from X, they are K-inequivalent.

12.13 Example. Let us prove that the following language is not regular.

$$K \stackrel{\text{def}}{=} \{a^n b^n \mid n \geq 0\}$$

We follow the script.

Define $X \stackrel{\text{def}}{=} \{a^n \mid n \geq 0\}$. X is clearly infinite.

Claim. if x and y are different strings from X, then $x \not\equiv_X y$.

Proof of claim: let $x = a^i$ and $y = a^j$, where $i \neq j$. Let z be b^i. Then $xz \in K$ but $yz \notin K$, thus $x \not\equiv_K y$. This completes the proof.

> *I have written the following proofs in a very robotic way, to emphasize the fact that the strategy is the same for all of them: to prove diferent languages non-regular you only need to vary your choice of inequivalent strings.*

A Hint. When a language has infinitely many equivalence classes, it might be very difficult to describe them in a general way, by a formula. So don't make your life more difficult than it has to be: if you want to prove some language to be non-regular, you need only give an argument that infinitely many classes exist, you don't have describe *all* the classes. In order to be rigorous and convincing that infinitely many classes exist you will probably have to give some formula for identifying infinitely many classes. But that's different from giving a formula to describe all classes. That could be hard. But that's ok, because you don't have to do it!

12.14 Example. Let $A = \{w \in \{a,b\}^* \mid w \text{ has the same number of } a\text{s as } b\text{s}\}$

Proof that A is not regular:

1. It suffices to show that A has infinitely many $\equiv_A$ classes.
2. To show this, here are infinitely many strings that are each inequivalent from the others. $\lambda, a, aa, \ldots, a^i, \ldots$
3. To show that there are inequivalent, consider any a^i and a^j.

   We have $a^i \not\equiv_A a^j$ because we can use the word $z = b^i$ to separate them.

12.15 Example. Let $B = \{w \in \{a,b\}^* \mid w \text{ is a palindrome}\}$

Proof that B is not regular:

1. It suffices to show that B has infinitely many $\equiv_B$ classes.
2. To show this, here are infinitely many strings that are each inequivalent from the others. $b, ab, aab, \ldots, a^ib, \ldots$
3. To show that there are inequivalent, consider any a^ib and a^jb.

   We have $a^ib \not\equiv_B a^jb$ because we can use the word $z = ba^i$ to separate them.

12.16 Example. Let

$$C = \{w \in \{(,)\}^* \mid w \text{ is properly-paired string of parentheses}\}$$

For example:

These are balanced: $x_1 = (())$, $x_2 = ()()$, $x_3 = (()())$, $x_4 = ((((()))))()()$

These are not balanced: $y_1 = (()$, $y_2 =)($, $y_3 = ())($

Proof that C is not regular:

1. It suffices to show that C has infinitely many $\equiv_C$ classes.
2. To show this, here are infinitely many strings that are each inequivalent from the others. $\lambda, (, ((, \ldots (^i \ldots$
3. To show that there are inequivalent, consider any $(^i$ and $(^j$.
   We have $(^i \not\equiv_C (^j$ because we can use the word $z =)^i$ to separate them.

Comment. Don't be unsettled by the fact that C also contains strings like "()()" and "((()()))" even though the proof above ignored these and only focussed on "(((...)))" We don't need to say something about *all* $\equiv$-classes in a non-regularity proof, we just have to find infinitely many.

12.17 Example. *Let* $Q = \{a^n \mid n \text{ is a perfect square}\}$

Proof that Q is not regular:

1. It suffices to show that Q has infinitely many $\equiv_C$ classes.
2. To show this, we claim that Q *itself* is a set of strings which are each inequivalent from the other. That is, for each pair of words $x = a^{n^2}$ and $y = a^{m^2}$ we will show that there is a z which separatees them.
3. To show that there are inequivalent, suppose that $x = a^{n^2}$ and $y = a^{m^2}$, with $n < m$. Take z to be the word a^{2n+1}. We now argue that $xz \in Q$ while $yz \notin Q$.
   Certainly $xz \in Q$ since $xz = a^{n^2}a^{2n+1} = a^{n^2+2n+1} = a^{(n+1)^2}$. To argue that yz is not in Q, we calculate: $yz == a^{m^2}a^{2n+1} = a^{m^2+2n+1}$, and $m^2 + 2n + 1$ cannot be a perfect square no matter what n and m are. Why? Certainly it is greater than m^2. But it is also smaller than $(m+1)^2$, since $(m+1)^2 = m^2 + 2m + 1$, and $2n + 1 < 2m + 1$.

If you think hard about what is going on here you will see that the essence of the proof is that the gap between any two successive squares increases. The "gap" between n^2 and $(n+1)^2$ is $(2n+1)$, and if we add that to n^2 we get a perfect square but if we add that to any larger m^2 we do not get a perfect square.

12.18 Example. Let $P = \{a^n \mid n \text{ is prime}\}$

To do this example we need to do a little preliminary math. First look back at Example 12.17. That example hinged on the fact that the gap between any two successive squares increases. We use the same idea here, but not *exactly*, since it is not true that the gap between successive primes increases uniformly, as we had with squares.

But what we *can* show pretty easily is that there are arbitrarily large gaps between primes. Specifically, let's show that for any n there is a sequence of at least $n-1$ consecutive composite numbers. And for that we just need to consider (having fixed an n) the numbers

$$n!,\ (n!+2),\ (n!+3),\ \ldots\ (n!+n)$$

each of these is composite, since each $(n!+k)$ is divisible by k.

Now, having done that, for a given prime p let's write $g(p)$ for the difference between p and the next prime beyond p. Then we can define an infinite sequence of primes $p_1, p_2, p_3, \ldots$ such that the gaps $g(p_i)$ are strictly increasing. In other words, if $i < j$ then $g(p_i) < g(p_j)$. This is what we need for our proof.

Proof that P is not regular:

1. It suffices to show that P has infinitely many $\equiv_C$ classes.
2. To show this, let $a^{p_1}, a^{p_2}, a^{p_3}, \ldots$ be the sequence of words corresponding to the sequence of primes described above.

   We claim that these a^{p_i} are pairwise inequivalent.
3. To show that there are inequivalent, suppose that $x = a^{p_i}$ and $y = a^{p_j}$, with $i < j$. Take z to be the word $a^{g(p_i)}$. Then the number of as in xz is exactly the next prime beyond p_i, namely $p_i + g(p_i)$. So $xz \in P$. But the number of as in yz is $p_j + g(p_i)$, and this is not a prime because $g(p_i) < g(p_j)$. So $yz \notin P$. So this z separates x and y.

12.5 Problems

139. EquivPractice

Fix $\Sigma = \{a, b\}$.

1. Let A be the language

$$\{w \mid w \text{ contains an occurrence of } abb\}$$

   (a) The words ab and ba are $\equiv_A$-inequivalent, that is $ab \not\equiv_B ba$.
   Find a specific word z that witnesses that fact.

   (b) The words λ and abb are $\equiv_A$-inequivalent. Find a specific word z that witnesses that fact.

   (c) The words λ and ba are $\equiv_A$-inequivalent. Find a specific word z that witnesses that fact.

   (d) Explain why the words abb and $babba$ are $\equiv_A$-equivalent, that is, $abb \equiv_A babba$.

2. Let B be the language

$$\{w \mid |w| \text{ is even}\}$$

   (a) The words aab and ab are $\equiv_B$-inequivalent. Find a specific word z that witnesses that fact. That is, find a z such that $aabz \in B$ yet $abz \notin B$, or vice versa.

   (b) The words λ and a are $\equiv_B$-inequivalent. Find a specific word z that witnesses that fact.

3. Let C be the language

$$\{a^i b^j \mid i < j\}$$

   (a) The words ab and ba are $\equiv_C$-inequivalent. Find a specific word z that witnesses that fact.

   (b) The words λ and abb are $\equiv_C$-inequivalent. Find a specific word z that witnesses that fact.

   (c) Explain why the words bba and ba are $\equiv_C$-equivalent.

140. EquivClassesPractice

Let

$$E = \{x \in \{a, b\}^* \mid x \neq \lambda \text{ and } x \text{ begins and ends with the same symbol}\}.$$

There are 5 $\equiv_E$ classes, that is, the index of E is 5. Write down one string in each of the $\equiv_E$ classes.

141. DFAMinSize

a) Let $A \subseteq \{0,1\}^*$ be the language $\{w \mid$ length of w is divisible by 4 $\}$. Prove that any *DFA* recognizing A must have at least four states, by writing down four strings that are pairwise $\equiv_A$-inequivalent.

Note for this and subsequent parts: To prove your 4 strings $\{w_1, w_2, w_3, w_4\}$ mutually ininequivalent you must consider each of the pairs w_i, w_j with $i \neq j$ and for each of these pairs, construct a string z such that exactly of the strings $w_i z$ and $w_j z$ is in A. Since you have 4 strings for this part, you will have $\binom{4}{2} = 6$ pairs to address.

b) Let $C \subseteq \{0,1\}^*$ be the language $\{w \in \{0,1\}^* \mid$ w has at most three 1s $\}$. Prove that any *DFA* recognizing C must have at least five states, by writing down five strings that are pairwise $\equiv_C$-inequivalent, and proving them to be $\equiv_C$-inequivalent.

(Rather than giving $\binom{5}{2} = 10$ individual arguments for inequivalence you might prefer to give a generic argument!

c) Let $B \subseteq \{0,1\}^*$ be the language $\{0^i 1^j \mid i, j \geq 0\}$. Prove that any *DFA* recognizing B must have at least three states, by writing down three strings that are pairwise $\equiv_B$-inequivalent, and proving them to be $\equiv_B$-inequivalent, as outlined in the previous part.

142. DFASizeConjecture

For the examples 12.4 and 12.5 in Section 12.2 (where the number of $\equiv$-classes was finite) construct a *DFA*. Look for a pattern between the $\equiv$-classes and the states of your *DFA*. Make a conjecture.

143. DeterminismBlowup

Over the alphabet $\Sigma = \{0,1\}$ let L_n be the set of strings whose nth-to-last symbol is 1. Precisely: L_n is $\{u0v \mid u, v \in \{0,1\}^*, v$ has length $(n-1)\}$

1. Explain how to build an *NFA* for L_n with $n+1$ states. (We've already done this, as one of the first examples to see why *NFAs* are useful)
2. Show that any two different bitstrings x and y of length n are L_n-inequivalent
3. From this we can conclude that the smallest *DFA* recognizing L_n has at least 2^n states ... which result in the text tells us this?
4. You just proved:

   > *The conversion from NFAs to equivalent DFAs can induce an exponential blowup in the size of the state space.*

Motivation: by putting these pieces together we see that the *NFA*-to-*DFA* transformation necessarily involves an exponential blowup in the number of states, for some *NFAs*.

144. LangIsClass?

True or False: for any language K, K is itself one of the equivalence classes of the relation $\equiv_K$.

If you answer True, give a proof, if you answer False, give a specific counterexample.

A series of non-regular examples

145. NonRegPractice

In the next problems you are asked to show that certain language are non-regular. These problem are roughly in order of difficulty. A few of them might repeat examples from the previous text; but it is useful to have these collected together.

General Hint. When asked to show a language to be non-regular, if you choose to use Corollary 12.12, the proofs always have the same shape. You should proceed as follows. You want to describe an infinite set of strings that are mutually inequivalent. Your job is to (i) describe the infinite collection of strings formally, then (ii) give an argument that for any two strings x and y in your family, there is a string z that separates them.

a) $A \stackrel{\text{def}}{=} \{a^n b^{2n} \mid n \geq 0\}$

b) $B \stackrel{\text{def}}{=} \{a^n b^m c^n \mid n, m \geq 0\}$

c) $C \stackrel{\text{def}}{=} \{a^n b^m \mid n \leq m\}$

d) $D \stackrel{\text{def}}{=} \{a^i b^n c^n \mid i \geq 0, n \geq 0\}$.

e) $E \stackrel{\text{def}}{=} \{w \in \{a,b\}^* \mid \exists x \in \{a,b\}^*,\ xx = w\}$.

f) $R \stackrel{\text{def}}{=} \{w \in \{a,b\}^* \mid \exists x \in \{a,b\}^*,\ xx^R = w\}$.

(Remember that x^R means the reverse of string x.)

g) The set of strings of as and bs whose length is a perfect square .

h) $\{a^n \mid n \text{ is a perfect cube}\}$

i) $\{a^n \mid n \text{ is a power of 2}\}$

146. InfSubsetAnBn

Prove that no infinite subset of $\{a^n b^n \mid n \geq 0\}$ is regular.

147. BoundedExponents

Exactly one of the following languages is regular. Which one? Prove it. Prove the other one to be non-regular.

a) $A = \{a^i b^j \mid i \geq j \text{ and } j \leq 100\}$

b) $B = \{a^i b^j \mid i \geq j \text{ and } j \geq 100\}$

148. NonRegGotcha

Zach Hacker thinks he has an argument that the language $R = \{a^n b^m \mid n, m \geq 0\}$ is not regular.

He says

> *Consider the following infinite set of pairs :*
>
> $$(a,b),(a,bb),(a,bbb),...(a,b^j),...$$
>
> *These pairs are all distinguishable: for any j, we have $a \not\equiv_R b^j$, since the string $z = a$ distinguishes them ($aa \in R$, but $b^j a \notin R$).*
>
> *Since we have an infinite set of $\not\equiv_R$ pairs, there must be infinitely many $\equiv_R$ classes, thus R is not regular.*

Clearly *something* is wrong here, since R is indeed regular. What is wrong with Zach's argument?

149. RegSubset

1. Prove or disprove: If L is regular and $K \subseteq L$ then K is regular.
2. Prove or disprove: If L is regular and $L \subseteq K$ then K is regular.

150. ApplyingClosure

Let R be a regular language and let N be a language which is not regular.

1. Suppose X is a language such that $X = R \cap N$. Does it follow that X is necessarily regular? If so, say why. Does it follow that X is necessarily non-regular? If so, say why. If your answers to the two previous questions was no, name a regular R and non-regular N satisfying $X = R \cap N$ with X non-regular, and name a regular R and non-regular N satisfying $X = R \cap N$ with X regular.

2. Suppose X is a language such that $N = R \cap X$. Does it follow that X is necessarily regular? If so, say why. Does it follow that X is necessarily non-regular? If so, say why. If your answers to the two previous questions was no, name a regular R and non-regular N satisfying $N = R \cap X$ with X non-regular, and name a regular R and non-regular N satisfying $N = R \cap X$ with X regular.

3. Suppose X is a language such that $R = N \cap X$. Does it follow that X is necessarily regular? If so, say why. Does it follow that X is necessarily non-regular? If so, say why. If your answers to the two previous questions was no, name a regular R and non-regular N satisfying $R = N \cap X$ with X non-regular, and name a regular R and non-regular N satisfying $R = N \cap X$ with X regular.

4. Suppose X is a language such that $R = \overline{X}$. Does it follow that X is necessarily regular? If so, say why. Does it follow that X is necessarily non-regular? If so, say why. If your answers to the two previous questions was no, name a regular R satisfying $R = \overline{X}$ with X non-regular, and name a regular R satisfying $R = \overline{X}$ with X regular.

5. Suppose X is a language such that $N = \overline{X}$. Does it follow that X is necessarily regular? If so, say why. Does it follow that X is necessarily non-regular? If so, say why. If your answers to the two previous questions was no, name a non-regular N satisfying $N = \overline{X}$ with X non-regular, and name a non-regular N satisfying $N = \overline{X}$ with X regular.

151. ClosureTricks1

For this problem, *assume* that the language $Eq = \{a^n b^n \mid n \geq 0\}$ is not regular.

Prove that the following languages over $\{a,b\}$ are not regular, without doing any new reasoning about $\equiv$-classes.

Hint. Use closure properties.

a) $Neq = \{a^k b^l \mid k \neq l\}$. (Caution: this set is not the same set as $\overline{Eq}$)

b) $K = \{w \mid w$ has an unequal number of as and bs$\}$ (Caution: this set is also not the same set as $\overline{Eq}$)

c) For this problem we introduce the temporary notation $\overline{w}$ to stand for the result of taking a string w and replacing as by bs and vice versa.

Let $G = \{w\overline{w} \mid w \in \{a,b\}^*\}$. Prove that G is not regular.

152. ClosureTricks2

Assume that the following language is not regular. $A = \{a^n \mid n$ is a perfect square$\}$

Prove that the set $D \subseteq \{a,b\}^*$ of strings of as and bs whose length is a perfect square is not regular, without doing any new reasoning about $\equiv$-classes.

Hint. The regular languages are closed under intersection. Write A as the intersection of D with something you know to be regular.

153. NotIff

Lemma 12.10 is not an "if-and-only-if". That is, it does not assert that if $x \equiv_K y$ then $\hat{\delta}_M(s,x) = \hat{\delta}_M(s,y)$. Indeed, that statement is false in general.

Give a concrete example of a *DFA* M recognizing a language K and two strings x and y such that $x \equiv_K y$ but $\hat{\delta}_M(s,x) \neq \hat{\delta}_M(s,y)$.

Hint. You can use a simple K. Use a dumb *DFA* (that really is a hint. A smart *DFA* won't work!)

154. NoNoNFA

It is important that M be a *DFA*, not just an *NFA*, in Lemma 12.10. For one thing, the notation "$\hat{\delta}_M(s,X)$" doesn't make sense for an *NFA*, since there can be more than one run, or no runs, on a given string.

Still, one could imagine *NFA*-versions of Lemma 12.10 that at least make sense, such as the following.

1. *Let K be a regular language, and let M be an NFA recognizing K. If there is a run of M on x and a run of M on y that end in the same state, then $x \equiv_K y$.*
2. *Let K be a regular language, and let M be an NFA recognizing K. If for every run on x ending a state p there is a run on y that ends in p, then $x \equiv_K y$.*

For each of the above statements, decide if it is true; if so give a proof, if not, give a counterexample.

Chapter 13

Regular Decision Problems

Here we study some decision problems concerning automata. This means we will consider some questions one might want to ask about a given automaton (such as "does this automaton accept all strings?") and see if we can build algorithms to answer these questions.

13.1 Decidable and Undecidable

A *decision problem* is, informally, a set of "yes/no" questions. A decision problem is *decidable* if there is an algorithm to answer it. By "algorithm" we men simply a program executable by a computer, which halts and returns an answer on all inputs. That last part, "halts on all inputs," will be **crucial** as we proceed.

Now in fact what we just said above is not quite as precise as we will eventually be: we need some infrastructure before arriving at a careful definition. Without a careful definition we won't be able to prove anything to be **un**decidable. But for now we will be proving things to be decidable, by actually producing algorithms, and so we will not go wrong working with what we wrote above.

13.2 Encoding Automata

Computers don't manipulate abstract mathematical objects, they manipulate concrete objects. If we are going to write algorithms that answer questions about finite automata, we have to have a data structure to represent them.

A universal, uniform, way to represent finite objects is as strings over an alphabet. This isn't a very deep remark; essentially if you can talk about something, then you have a way to represent it as a string.

So this is our slogan:

> *When we manipulate finite objects, we are manipulating strings. In fact, since any alphabet can be encoded into the binary alphabet* $\{0,1\}$ *("bitstrings are universal"), without loss of generality we are manipulating bitstrings.*

You might like to go back and read the discussion at the beginning of Section 5.3 for more on this point.

So next we look at the problem of how to represent automata as bitstrings. This will be a funny section, because the particular details of how we do this *will never matter to us in the rest of our work!* The only thing that does matter is that it can be done in the first place (in a way that passes some sanity-checks). But in order to develop the right intuitions about encoding, and shrug off details later, you have to actually do some encoding at the start.

Example: Encoding Automata

The encoding scheme we define works exactly the same way for *DFAs* as for *NFAs*, so we will just speak of "automata" below.

It will easiest to understand the process if we think of it in two stages: define an encoding that is "human readable" for clarity, then finish the job by blasting everything down to bitstrings. This two-stage description of the process is just for pedagogical purposes.

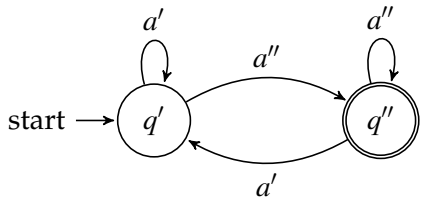
First we construct an alphabet Δ to talk about *DFA* states and *DFA* input alphabet. A crucial feature of our encoding is that we use **one** finite alphabet to encode all possible automata. The trick is to embrace the fact that we can name our alphabet symbols and states anything we like without changing the nature of the automaton as a machine.[1]. We agree to name every alphabet symbol (of any alphabet) as a string consisting of the symbol a followed by a number of tick-marks; and to name every state (of any automaton) as a string consisting of the symbol q followed by a number of tick-marks.

Since we can name states anything we like, it should be clear that there is no harm in assuming that every *DFA* will have s' (one tick) as the start state.

So for our encoding we will use the alphabet $\Delta = \{a, q, f, e, '\}$. Think of a as standing for "alphabet symbol", q as standing for "state", f as standing for "accept state." e as standing for "transition", and The last symbol in Δ is just a little tick-mark.

For example here is a *DFA* whose states and alphabet are named according to these conventions.

[1] It's just like re-naming the variables in a program: you haven't really changed the program's behavior



Pause for a moment and make sure it is clear to you that any automaton can be represented using such names for its input alphabet and states.

Then the first stage of our encoding is to concatenate following strings.

- We write the name of the start state as the first entry of the encoding.
- For each accept state $q\cdots$ (q followed by some number of tick-marks) we add to the encoding the string $fq\cdots$
- For each transition we add a fact to the encoding by writing the strings $e\quad q\cdots\quad a\cdots\quad q\cdots$ (spaces added here for readability) where qu and qv are strings encoding the source and target nodes respectively, and aw is the string encoding the alphabet symbol.
- The entire automaton will be encoded by concatenating the codes describing the accepting states, and with the transitions in that order.

  Note that there are many orders we might choose to concatenate our data, but we declare that all of them are legal encodings.

For the *DFA* immediately above, we have the following first-stage encoding (where we have inserted spaces for readability)

$$fq''\quad eq'a'q'\quad eq'a''q''\quad eq''a''q''\quad eq''a'q'$$

Now the second step is just to map each symbol of Δ to a bitstring, and transform the Δ-encodings into bitstring encodings in the obvious way.

Completely arbitrarily we use the following "dictionary."

- $' \mapsto 000$
- $a \mapsto 001$
- $q \mapsto 010$
- $e \mapsto 011$
- $f \mapsto 100$

So our final encoding is (again with spaces and a line break for readability)

100 010 000 000 011 010 000 001 000 010 000 011 010 000 001 000 000 010 000
000 011 010 000 000 001 000 010 000 000 011 010 000 000 001 000 000 010 000

The important takeaways here are

1. Every automaton gets at least one encoding.
2. Not every bitstring over arises as the encoding of an automaton; this is slightly annoying but doesn't cause any real harm either.
3. Encodings are unambiguous in the sense that no bitstring is the encoding of more than automaton.

Now we can say more about what we meant when we said that the specific details of our encoding "don't matter." Obviously we made a bunch of somewhat random design choices in the coding above. But what matters is that we satisfy the above "takeaways." Those properties are what will need for our future work.

Encoding Multiple Inputs

Lots of decision problems we want to solve will present (intuitively) more than one input. For example, we may want to decide whether two *DFAs* accept the same language. But we want to have a uniform formalism, in which problem instances are encoded as strings to be given to an algorithm. The solution is natural enough: if we have two "inputs" we just take the strings corresponding to these inputs and squash them together to make one input string. The only subtlety is that we can't use ordinary concatenation, since our algorithm will want to be able to unambiguously pull apart the big string into its two original pieces.

So here is what we do. The remarks below apply generally, to encoding any objects, not just automata.

- Isolate an alphabet Δ in which we can represent the data for the problem instances.
- Pick a new symbol \$ not in Δ: we will treat $\Delta \cup \{\$\}$ as a new alphabet.
- Represent the problem instance involving multiple Δ^*-strings $\langle a_1, a_2, \ldots, a_k \rangle$ as the single string $a_1\$a_2\$\ldots\$a_k$ This is a string over the alphabet $\Delta \cup \{\$\}$.
- Map down to bitstrings as before, after assigning bitstrings to each symbol in $\Delta \cup \{\$\}$

13.3 Some *DFA* Decision Problems

13.3.1 DFA Membership

DFA Membership

INPUT: (the encoding of) a DFA M, (the encoding of) a word w

QUESTION: $w \in L(M)$?

It is tedious to keep saying "the encoding of a *DFA M*" so from now on we will allow ourselves to simply say, for example, that the input to a problem is "a *DFA M*".

Anyway, finding an algorithm for this problem is not much of a challenge.

Algorithm 10: DFA Membership

Input: *DFA M* and string x
Decides: *does* $L(M)$ *contain* x*?*

just simulate the machine.

Complexity: $O(|w|)$, since we take one step per symbol in the word w.

13.3.2 DFA Emptiness

Here is the first interesting example.

DFA Emptiness

INPUT: DFA M

QUESTION: $L(M) = \emptyset$?

The most naive attempt at an algorithm we could make would look something like this:

> *Let* $x_0, x_1, x_2, \ldots$ *be the possible inputs to M. Run M on each of them in turn. If M accepts any of them, return NO. If M accepts* none *of them, return YES.*

But this is a very bad "algorithm," because there are infinitely many strings x_i to check. So there is no way we would every say "YES" in a finite time. In fact we will decline to even call this an algorithm.

A much better approach would recognize that if M is going to accept any string at all then it will accept some string whose length is bounded by the number of states on M. (Think about why.) So we could replace the original unbounded search by a finite search, and get a legitimate algorithm. We won't discuss this idea further here, since it is computationally very inefficient, and we can do better.

We simply observe that M will accept some string precisely if there is a way to reach some accepting state by following transitions—no matter what the labels are—from the start state. This is a well-known graph search problem. So :

Algorithm 12: *DFA* Emptiness

Input: *DFA* M
Decides: $L(M) = \emptyset$

view the *DFA* $M = (\Sigma, \delta, s, F)$ as a directed graph ;
do a depth-first search starting with s ;
if *any of the visited vertcies is in* F **then**
  return NO
else
  return YES

Complexity: the complexity of depth-first search in a graph is $O(n+e)$ where n is the number of nodes and e is the number of edges. The number nodes in our graph is $|Q|$. The number of edges is $k \times |Q|$ where $k = |\Sigma|$. Treating the size of Σ as being fixed as the DFA varies, we have that the complexity is $O(|Q| + k|Q|) = O(|Q|)$.

13.3.3 DFA Universality

How about the other extreme?

DFA Universality
INPUT: DFA M
QUESTION: $L(M) = \Sigma^*$?

We again stress that an exhaustive search is *not* a useful approach here. If we generate-and-test all possible inputs then—dually to the Emptiness problem—we could successfully discover NO answers, but we could never arrive at a YES answer.

Inspired by the trick in the Emptiness problem, we might consider viewing M as a direct graph, and doing some graph-theory magic to test a question like "is it true that *all* paths from the start state lead to an accepting state?". That sounds do-able, though perhaps a bit tricky to implement correctly.

But the key idea, which we will exploit all through the section, is to use the closure properties we have developed. Remember that we have an algorithm that will, given a *DFA* M, compute a *DFA* M' whose language is the complement of the language of M. So testing whether $L(M)$ is all strings is equivalent to testing whether $L(M')$ is empty. So we can use exactly the same graph-theoretic algorithm as for *DFA* Emptiness, as long as we do a little *DFA* magic at the start.

Algorithm 14: DFA Universality

Input: *DFA* M
Decides: $L(M) = \Sigma^*$

construct a DFA M' with $L(M') = \overline{L(M)}$;
call the algorithm *DFA Emptiness* on M';
return that answer

Complexity: The same as for the *DFA Emptiness*, since the construction of M' can be done in linear time (or constant time with a clever data structure). So the complexity is $O(|Q|)$ where Q is the set of states of M.

The next example is a decision problem about an ordered pair of *DFAs*. Recall our convention about how to encoded multiple conceptual inputs into a single string.

13.3.4 DFA Containment

DFA Containment

INPUT: DFAs M_1, M_2

QUESTION: $L(M_1) \subseteq L(M_2)$?

This is not an artificial problem. When automata are used for specification, we often have one automaton S representing a systems' specification, and automaton M representing a systems' implementation. To ask whether all behaviors of the system are consistent with its specification is to whether $L(M) \subseteq L(S)$. To be fair, when automata are used as specifications they are typically not simple *DFAs*; they will have some non-determinism and maybe slightly richer structure. But we use the same techniques for such automata, so view this remark as a teaser and an invitation to further study.

The idea: Use that fact that for any sets X and Y, $X \subseteq Y$ iff $X \cap \overline{Y} = \emptyset$. Use this fact and the constructions for complement and intersection of DFA-languages to reduce this

problem to DFA emptiness.

Algorithm 15: DFA Containment

Input: *DFAs* M_1 and M_2
Decides: $L(M_1) \subseteq L(M_2)$

construct a DFA M_2' with $L(M_2') = \overline{L(M_2)}$;
construct a DFA N with $L(M_2') \cap L(M_1)$;
call the algorithm *DFA Emptiness* on N;
return that answer

Complexity: The construction of M_2' can be done in constant time, and M_2' has the same number of states as M_2. The construction of N takes time $O(|Q_1| \times |Q_2|)$; the DFA emptiness test takes linear time. So the final complexity result is $O(|Q_1| \times |Q_2|)$.

13.3.5 DFA Equivalence

DFA Equivalence

INPUT: DFAs M_1, M_2

QUESTION: $L(M_1) = L(M_2)$?

In Section 14.7 we saw an algorithm for this based on $\approx$. Here is another, that uses the idea we are exploring in this section, of reducing problems (ultimately) to *DFA* Emptiness.

The idea: $X = Y$ iff $X \subseteq Y$ and $Y \subseteq X$.

Algorithm 17: DFA Equivalence

Input: *DFAs* M_1 and M_2
Decides: $L(M_1) = L(M_2)$

call *DFA Containment* on (M_1, M_2) ;
call *DFA Containment* on (M_2, M_1) ;
if *both of these return YES* **then**
  return YES
else
  return NO

Complexity: We make two calls to *DFA Containment*, which are each $O(|Q_1| \times |Q_2|)$, so the complexity of *DFA Equivalence* is $O(|Q_1| \times |Q_2|)$.

13.3.6 DFA Infinite Language

DFA Infinite Language

INPUT: DFA M

QUESTION: is $L(M)$ infinite?

The idea: use depth-first search to search for a path to an accepting state with a loop back to that state.

Algorithm 19: DFA Infinite Language

Input: *DFA M*
Decides: $L(M)$ *is infinite?*

view the DFA $M = (\Sigma, Q, \delta, s, F)$ as a directed graph ;
compute the set X of all states that are on a path from s to an accept state ;
call depth-first search starting with s ;
foreach *of the visited vertices* $q \in X$ **do**
    call depth-first search starting with q ;
    if *q is visited* **then** // this means there is a loop starting and ending at q
        return YES
return NO

Complexity: As described above, each depth-first search of the DFA takes $O(|Q|)$ time. The number of such searches is bounded by the number of states, so this yields an $O(|Q|^2)$ bound.

13.3.7 Other DFA Questions

With a little more cleverness we can answer richer questions about a *DFA*'s language. For example:

13.1 Example. *DFA Accept Any Even*

INPUT: DFA M

QUESTION: Does M accept any strings of even length?

The idea: It is easy to make *DFA* M_E that accepts precisely the strings of even length. So to ask whether the given *DFA* M accepts any strings of even length is to ask whether $L(M)$ and $L(M_E)$ have any strings in common. That's the same as asking whether $L(M) \cap L(M_E)$ is non-empty. We know how to make a *DFA* accepting the intersection of two languages, and we know how to test whether the language of a

DFA is non-empty....

Algorithm 20: DFA AcceptAnyEven

Input: a DFA M
Decides: *does* $L(M)$ *contain any even-length strings?*

construct a *DFA* M_E such that $L(M_E) =$ the strings of even length ;
construct a *DFA* P such that $L(P) = L(M) \cap L(M_E)$;
call Algorithm DFA Emptiness on P ;
if *this return YES* **then**
  return NO
else
  return YES

Here's another example.

13.2 Example. *DFA Even Equivalence*

INPUT: *DFAs* M and N

QUESTION: Do M and N agree on all even-length strings?

This problem seems easier once you actually write the question precisely, *i.e.*, mathematically! Letting E be the set of all even-length strings, we are asking whether

$$L(M) \cap E = L(N) \cap E$$

Here is an algorithm.

Algorithm 21: DFA AgreeOnEven

Input: *DFAs* M and N
Decides: $L(M) \cap E = L(N) \cap E$*?*

construct a *DFA* M_E such that $L(M_E) =$ E ;
construct a *DFA* P_1 such that $L(P_1) = L(M) \cap L(M_E)$;
construct a *DFA* P_2 such that $L(P_2) = L(N) \cap L(M_E)$;
call Algorithm *DFA* Equivalence on P_1 and P_2 ;
return that answer

You are asked to do more reasoning of this kind in the Problems at the end of the chapter.

Caution!

Please don't make the sloppy mistake of saying things like, "we have an algorithm to decide whether a regular language is empty." The input to decision problems is **never** a *language,* except in a trivial case where the language is a finite set of strings.

This wouldn't make any sense. The input to any decision problem must be a finite object, something that can be presented to a computer. Languages are typically *infinite* mathematical objects. So the inputs to decision problems above are (the encodings of) a DFA. Then the question that gets asked is about the language associated with the DFA or regular expression.

having said that ,we are not restricted to asking questions about *DFAs*: we can try to reason about anything that can be encoded as a finite string. Read on.

13.4 *NFA* Inputs

Naturally, we can consider asking questions about *NFAs* as well as *DFAs*. There is nothing really new to say here, except a caution that we need to be a bit careful about the basic constructions.

13.4.1 *NFA* Membership

NFA Membership

INPUT: *NFA* N, string w

QUESTION: $w \in L(M)$?

This is not quite as simple as the *DFA* Membership problem. We cannot simply say, "on input M and string w, just run M on w" because of the fact that an *NFA* can have more than one attempted run on an input. What we are testing for is whether any one of these attmpeted runs succeeds. The problem isn't hard though: we just need to observe that there are only finitely many runs of an *NFA* on a given string x: they form a finite-branching tree if you think about it, whose path lengths are the same as the length of x, hence the number of paths is finite, and we can test them all.

Another obvious algorithm is: (i) convert N to a DFA M, then (ii) run the DFA membership algorithm on M and w. But this can take time exponential in the number of states of N. We can do better; recall Problem 109.

13.4.2 *NFA* Emptiness

This submits to exactly the same algorithm as *DFA* Emptiness: view the *NFA* as a directed graph and decide whether there are any paths from the start state to an accepting state.

13.4.3 *NFA* Universality

Here we *cannot* use (exactly) the same algorithm as for *DFA* Universality. The main trick in the latter algorithm is to swap the accepting/non-accepting polarity of the states of the automaton and then call the Emptiness decision procedure. But as we have noted, swapping the accepting/non-accepting polarity of the states of the automaton does *not* have the effect of complementing the language accepted.

What we do is to convert the *NFA* to a *DFA* and then call the *DFA* Emptiness algorithm.

Algorithm 22: *NFA* Universality

Input: An *NFA* N
Decides: $L(N) = \Sigma^*$?
construct a DFA M with $L(M) = L(N)$;
call the algorithm *DFA Universality* on M;
return that answer

13.4.4 Other *NFA* Decision Problems

When confronted by a decision problem concerning *NFAs*, if it analogous to one already solved for *DFAs*, you can be confident that it is solvable in a similar way: sometimes you can use exactly the same algorithm, sometimes you may have to convert some *NFAs* to *DFAs* and then call the corresponding algorithm about *DFAs*.

13.5 Decision Problems about Regular Expressions

For decision problems when the inputs are regular expressions we can often use the technique developed in Section 13.5

13.3 Example. Consider the decision problem

> *RegExp Language Emptiness*
>
> INPUT: *RegExp* E
>
> QUESTION: does $L(E) = \emptyset$?

Here we show an algorithm Empty?, returning a boolean, to solve this problem. We use recursion on the structure of E.

Algorithm 23: *RegExp* Language Emptiness

Name : Empty?
Input : E
Decides : $L(E) = \emptyset$
if $E = \mathbf{O}$ **then**
  return yes
if $E = c$ *(for* $c \in \Sigma$*)* **then**
  return no
if $E = E_1 + E_2$ **then**
  if Empty?(E_1) = *yes and* Empty?(E_2) = *yes* **then**
    return yes
  else
    return no
if $E = E_1E_2$ **then**
  if Empty?(E_1) = *yes or* Empty?(E_2) = *yes* **then**
    return yes
  else
    return no
if $E = (E_1)^*$ **then**
  return no

13.4 Check Your Reading. *Trace through the algorithm step-by-step on some inputs. You might try*

- $\mathbf{O}^*$
- $\mathbf{cO}$
- $\mathbf{cO} + \mathbf{O}^*$
- $(\mathbf{cO})(\mathbf{O}^*)$

Lots of questions about regular expressions are very easy to write algorithms for using this technique. Several examples are given as problems for you to solve in the Problems at the end of this chapter.

Some *RegExp* decision problems don't lend themselves as readily to that syntax-directed technique. A key example is

> Regular Expression Equivalence: given regular expressions E_1 and E_2, is $L(E_1) = L(E_2)$?

Since this have obvious corresponding *DFA* problems we can solve it by translating the regular expressions into *DFAs* and using our *DFA* Equivalence algorithm. This is inefficient, though.

13.5.1 Regular Expression Membership

Regular Expression Membership

INPUT: a regular expression α, a word w

QUESTION: is $w \in L(\alpha)$?

Notice that there is no obvious way to *directly* look at a regular expression and decide whether a given word matches it. But, it is still true that there is an algorithm to solve the problem:

Algorithm 24: Regular Expression Membership

given α, build an NFA M such that $L(M) = L(\alpha)$;
call the NFA Membership algorithm on M and w

The worst-case complexity of this is not so good, since it will involve eliminating λ-transitions from NFAs, then simulating NFAs. The problem of finding an algorithm that is fast—both in theory and in practice—is interesting and has a long history... we won't go into it here.

13.5.2 Regular Expression Equivalence

Regular Expression Equivalence

INPUT: *RegExp* E and F

QUESTION: $L(E) = L(F)$?

The idea. We know how to translate from regular expressions to DFAs, and we know how to decide whether two DFAs accept the same language.

Algorithm 25: RegExpEquivalence

Input: two regular expressions E and F
Decides: $L(E) = L(F)$*?*

construct *DFA* M_E such that $L(M_E) = L(E)$ and *DFA* M_F such that $L(M_F) = L(F)$;
return the result of Algorithm DFAEquivalence on (M_E, M_F)

13.5.3 Other Regular Expression Decision Problems

Most other decision problems about *RegExp* can be solved in the same was as above: by translating to the automata world.

For some problems of course, we may have easier solutions by staying in the regular expression world, since we have algorithms that exploit the inductive structure of regular expressions.

13.6 Decision Problems are Languages

The following observation might make your head spin a bit. We won't really use it until later chapters but we mention it here because it is the logical place for it.

Consider any decision problem; for concreteness here, consider the *DFA* Emptiness problem. Once we have encoded our *DFAs* as bitstrings, the set of all *DFAs* that yield a "yes" answer to the *DFA* Emptiness question becomes a set of bitstrings. Since it is a set of bitstrings, it is a language over $\{0,1\}$. That is to say, the set of *DFAs* $M = \langle \Sigma, Q, \delta, s, F \rangle$ such that $L(M) \subseteq \Sigma^* \neq \emptyset$, is *itself* a language, over the alphabet $\{0,1\}^*$.

Similarly—considering the *DFA* Infinity decision problem— the set of *DFAs* whose language over the *DFA*'s input alphabet is infinite is a language over $\{0,1\}$.

In fact, once we defined a way to encode pairs of inputs as single bitstrings, the set of pairs (M,N) of *DFAs* such that $L(M) = L(N)$ is once again a language itself over $\{0,1\}$.

What all this means is that the question of finding an algorithm for a given decision problem is, really, the question of finding an algorithm for membership in a certain language over $\{0,1\}$. This gives more robustness to our slogan that "bitstrings are universal." For now you can view this as a curiosity but it will be technically crucial later, when we study decision problems which cannot be solved algorithmically.

13.7 Problems

When a problem asks for a decision procedure, your answer should be precise but non-fussy pseudocode. If you want to use algorithms for any of the following operations just call them, don't derive them.

- *DFA* Emptiness
- *DFA* Universality
- *DFA* Equivalence
- *DFA* Infinite Language
- the complement construction on *DFAs*
- the product construction on *DFAs*, for union and for intersection

155. EncodeNFA

Refer to Problem 106. Note that we phrased that problem anticipating this section's naming conventions about states and alphabets. So: using this section's encoding scheme, choose two of the *NFAs* with one state and write down their encodings.

156. AcceptEveryEvenDFA

Give an algorithm for the following decision problem.

> *DFA AcceptEveryEven*
> INPUT: A *DFA M*
> QUESTION: Does *M* accept every even length string?

(This is not the same as asking whether *M* accepts *only* the even-length strings.)

157. AcceptNoOddDFA

Give an algorithm for the following decision problem.

> *DFA AcceptNoOdd*
> INPUT: A *DFA M*
> QUESTION: Does *M* accept no odd-length strings?

This is the same as asking, when D is the set of all odd-length strings, whether $L(M) \cap D = \emptyset$.

158. RejectInfiniteDFA

Give an algorithm for the following decision problem.

DFA RejectInfinite
INPUT: A *DFA M*
QUESTION: Does M reject infinitely many strings?

159. DifferInfiniteDFA

Give an algorithm for the following decision problem.

DFA DifferInfinite
INPUT: *DFAs M* and *N*
QUESTION: Do M and N differ on infinitely many inputs?

Equivalently : is the symmetric difference between $L(M)$ and $L(N)$ infinite?

160. ContainmentEvenDFA

Give an algorithm for the following decision problem.

DFA ContainmentEven
INPUT: *DFAs M* and *N*
QUESTION: Does M accept all the even-length strings that N does?

This is the same as answering yes or no to the claim : *for every x of even length, if* $x \in L(N)$ *then* $x \in L(M)$.

Note that we do not care how M and N compare on odd-length strings.

161. ContainmentNFA

Give an algorithm for the following decision problem.

NFA Containment
INPUT: *NFAs* N_1 and N_2
QUESTION: Does $L(N_1) \subseteq L(N_2)$?

162. InfiniteNFA

Consider the decision problem

NFA Infinite Language

INPUT: *NFA N*

QUESTION: is $L(N)$ infinite?

Will this problem submit to exactly the same algorithm as for *DFA* Infinite Language, or do we need to convert N to a *DFA* M and *then* call *DFA* Infinite?

163. Has1RegExp

Let $\Sigma = \{0,1\}$. Write a recursive algorithm OneOccurs? to return a boolean as to whether a *RegExp* E matches any strings with a "1" occurring.

So OneOccurs?(E) = false iff $L(E) \subseteq 0^*$)

You may assume you have a function Empty? for testing whether a *RegExp* has an empty language.

164. HasNilRegExp

Consider the decision problem

RegExp HasNil

INPUT: *RegExp E*

QUESTION: is $\lambda \in L(E)$?

Write an algorithm Nullable?, returning a boolean, to solve this problem. Use recursion on the structure of E.

165. JustNilRegExp

Consider the decision problem

RegExp JustNil

INPUT: *RegExp E*

QUESTION: is $L(E) = \{\lambda\}$?

Write an algorithm JustNil?, returning a boolean, to solve this problem. Use recursion on the structure of E. Use the same approach as in Example 13.3.

Hint. At one point you will want to use the fact that there is an algorithm Empty? for deciding *RegExp* Language Emptiness. This was done in Example 13.3, so you can simply call Empty? when needed.

166. InfRegExp

Consider the decision problem

> *RegExp Infinite Language*
>
> INPUT: *RegExp E*
>
> QUESTION: is $L(E)$ infinite?

Write an algorithm Infinite?, returning a boolean, to solve this problem. Use recursion on the structure of E. Use the same approach as in Example 13.3.

Hint. At one point you will want to use the fact that there are algorithms Empty? and JustNil? for deciding whether a *RegExp* denotes $\emptyset$ or $\{\lambda\}$. Just call these algorithms when needed.

Chapter 14

DFA Minimization

Suppose A is a regular language. Can there be more than one *DFA* M with $L(M) = A$?

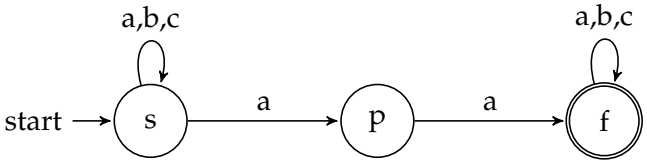
Let's first note that we need to be precise when we say "more than one *DFA*". Surely if we systematically rename the states of a *DFA* that shouldn't count as a different *DFA*, right? So whenever we speak *DFAs* being "the same" or not, we mean "up to renaming of states".

Next let's note that there is an easy uninteresting answer to the above question: if M is a *DFA* and we add one or more unreachable states to M we'll get a different *DFA* but we will not have changed the language accepted.

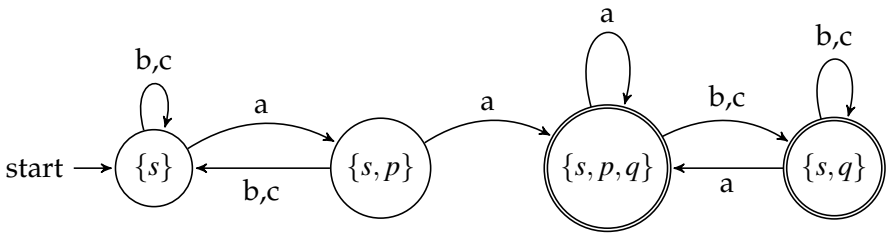
So the real question we want to ask is, *Suppose A is a regular language, can there be more than one DFA with no inaccessible states recognizing A, up to renaming of states?* The answer is yes; see several examples in this section.

That's not surprising, but the following good news is surprising. We can, algorithmically, optimize our *DFAs*, to eliminate unreachable states and to combine states that "do the same thing". Indeed we will see that for any *DFA* M there is a unique best optimized version of M. It will take some work to make that precise and show the optimization algorithm, but it will be worth it.

14.1 Example. Often the *NFA*-to-*DFA* construction will yield a *DFA* that is not minimal. Here is the obvious *NFA* over the alphabet $\{a, b, c\}$ that accepts those strings with *aa* as a substring.



If we use the subset construction to build a *DFA* recognizing this same language we get this:



But we could collapse the two accepting states into one and still have a *DFA* for the same language.

Having noted that, let's ask more ambitious questions. For any regular language A, consider all the *DFAs* that accept A. Certainly there is a smallest number of states that occur among all these *DFAs*. Let's call these *minimal DFAs* for the language A.

Here are the question we want to ask in this section.

1. Can we *construct* a minimal *DFA* for A if we start with an arbitrary *DFA* for A?
2. Are all the minimal *DFAs* for a language A essentially the same?

The answer to the first question is yes: we will show how we can construct a *DFA* M' which is equivalent to M and which has the smallest possible number of states among all *DFA* equivalent to M. The answer to the second question is also yes: given an arbitrary *DFA* M there is a *unique DFA* M' of minimal size equivalent to M.

14.1 Unreachable States

First note that if our given *DFA* M has some unreachable states, we can eliminate them immediately in our search for a minimal equivalent of M. And in fact the techniques below—or rather our analysis of its correctness—will rely on the assumption that the input *DFA* has no unreachable states.

It is algorithmically straightforward to eliminate unreachable states from a *DFA*: a depth-first search on the directed graph associated with the *DFA* will discover all the reachable states. **So henceforth we make that assumption: all *DFAs* we work with in this section have no inaccessible states.**

14.2 State Equivalence

In this section we define an equivalence relation $\approx$, on states of a given *DFA*.

Caution! In Section 12 we looked at the equivalence relation K-equivalence on strings. We will eventually that this relation and the relation $\approx$ that we are about to define are entangled in a deep and interesting way. But be careful not to mix these up, they are certainly not the same thing. Indeed *they don't even relate the same kind of objects:* K-equivalence compares strings, while $\approx$ compares machine states.

14.2 Definition. Let $M = (Q, \Sigma, \delta, q_0, F)$ be a *DFA*; let p and q be states of M. We say that states p and q are *M-equivalent*, written if $p \approx_M q$, if

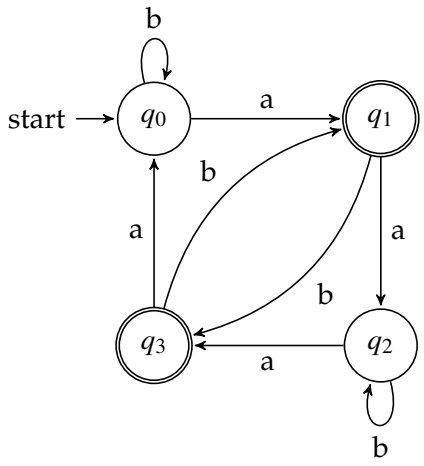
$$\text{for every } x \in \Sigma^* : \text{ if } p \xrightarrow{x}\!\!\!\!\!\rightarrow\ p' \text{ and } q \xrightarrow{x}\!\!\!\!\!\rightarrow\ q' \text{ then } p' \in F \text{ if and only if } q' \in F$$

When p and q are not M-equivalent we write $p \not\approx_M q$. Unfolding the definition, we have that

$$p \not\approx_M q \text{ if and only if}$$

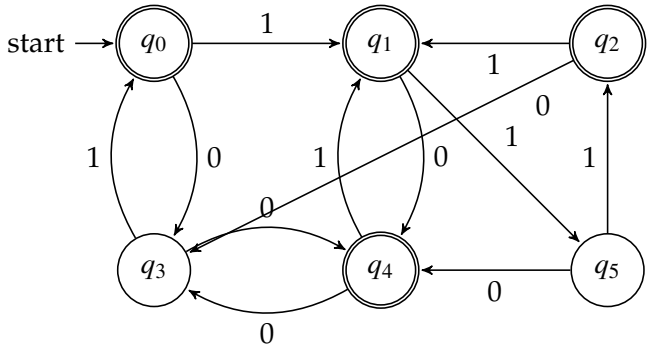
$$\exists x \in \Sigma^* : \text{ with } p \xrightarrow{x}\!\!\!\!\!\rightarrow\ p' \text{ and } q \xrightarrow{x}\!\!\!\!\!\rightarrow\ q' \text{ but exactly one of } p' \in F \text{ or } q' \in F$$

14.3 Example. Starting with



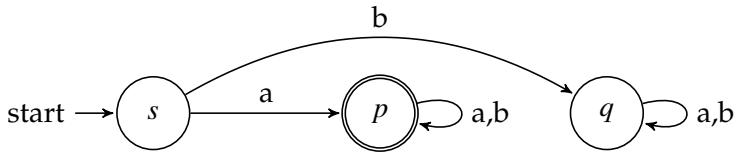
you can check that $q_0 \approx q_2$ and that $q_1 \approx q_3$.

14.4 Example. Let's start with this *DFA*.



Here it turns out that $q_0 \approx q_2$ and $q_0 \approx q_4$ and $q_3 \approx q_5$.

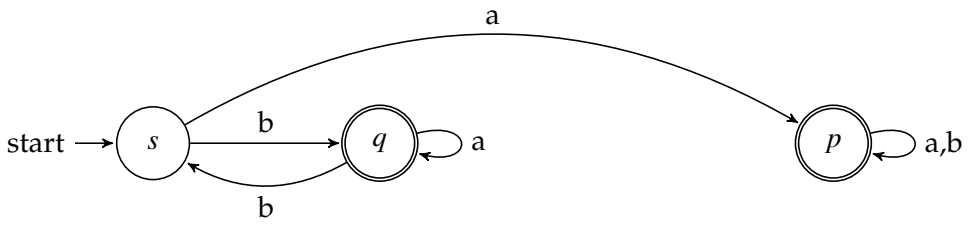
14.5 Example. In this *DFA*



no pair of distinct states is M-equivalent. For example, the reason that p and q are not M-equivalent is that the empty string distinguishes them. The reason that s and q are not equivalent is that a sends s to an accepting state, while a sends q to a non-accepting state.

The previous example makes it clear that we can never have two states be M-equivalent if one of them is accepting and the other is not. But we have to be careful:

14.6 Example. In this *DFA*



no pair of states are M-equivalent. For example, even though both p and q are accepting, the string b distinguishes them.

14.7 Example. Often when a *DFA* is constructed from an *NFA* by the subset construction, we construct different but equivalent states. For example, in the *DFA* constructed in Example 14.1 , the states labelled $\{s, p, q\}$ and $\{s, q\}$ are M-equivalent. The states labelled $\{s\}$ and $\{s, p\}$ are M-inequivalent: the string a distinguishes them.

Note that it is not clear at first glance how to test algorithmically whether $p \approx q$, since the definition seems to involve consideration of all strings in Σ^*. Put that question aside for a moment, and ask: why do we care about $\approx$?

The answer is that if $p \approx q$ then we can "collapse" p and q to get a smaller *DFA*, and this smaller one will accept precisely the same language as M. In fact we will do *all* such collapses simultaneously.

But first, some observations about $\approx$.

14.8 Lemma.

1. *The relation $\approx$ is an equivalence relation;*
2. *The relation $\approx$ respects transitions: if $p \approx q$ then for every $a \in \Sigma$, when $p \xrightarrow{a} p'$ and $q \xrightarrow{a} q'$, then $q \approx q'$.*

These observations will be important as we argue for the correctness of the construction in Section 14.4

14.3 Computing the $\approx$ relation

We did the examples so far "by inspection" but what if the *DFAs* had thousands of states? Is there an algorithm that, given M, computes the relation $\approx$? It's not obvious at first glance how to do so. To see why you should be suspicious, look carefully at the definition of $p \approx q$: to conclude that p and q are related in this way, according to the definition (cf Definition 14.2) we have to check that "for all $z \in \Sigma^*$, z doesn't distinguish p and q. That's infinitely many z to "test". We certainly can't do a naive exhaustive search through all candidate zs.

If we think about computing the *complement* of the $\approx$ relation (which is just as good as computing $\approx$ of course) we get a clue about how to proceed: to conclude that $p \not\approx q$, we have to check whether *there exists* some $z \in \Sigma^*$ such that z distinguishes p and q. And if you think about it a little you should expect that if such a z exists, there will be one whose length is no longer than the number of states.

14.9 Check Your Reading. *Before going further convince yourself that if $p \not\approx q$, then there exists some $z \in \Sigma^*$ such that z distinguishing p and q whose length is no longer than the number of states.*

You don't need a formal proof of this fact, it will emerge from our analysis of Algorithm 26. But understanding this fact intuitively will lead you to understand the algorithm.

Once you believe that bound on possible witnesses to $p \not\approx q$, then it is clear that in principle we could test $p \not\approx q$, by an exhaustive search. But in fact we can do better than naively searching for such a z; here is an efficient algorithm that does the job.

The following algorithm computes the $\approx$ relation for a given *DFA* M. During the course of the algorithm we actually compute the complement, D, of $\approx$, and at the end return $\approx$ itself.

The basic idea of the algorithm is this

1. If r and t are states such that one of them is accepting and the other one isn't, then clearly $r \not\approx t$.
2. If r' and t' are states such that for some alphabet symbol a, we have $r' \xrightarrow{a} r$ and $t' \xrightarrow{a} t$ where r and t are as in the previous sentence (exactly one of them is accepting), then $r' \not\approx t'$ as well...in fact
3. Using the same intuition, if p and q are states such that for some *string* x, we have $p \xrightarrow{x} r$ and $q \xrightarrow{x} t$ with exactly one of r and t being accepting, then $p \not\approx q$ as well.
4. Furthermore, if pair $\{p,q\}$ satisfies $p \not\approx q$, it because there is some x which witnesses that fact in the way described in the previous sentence. So the method above will find all the $\not\approx$ pairs for us.

With this intuition in place, the algorithm is presented formally as Algorithm 26.

A formal proof of the correctness of Algorithm 26 is omitted here. One potentially confusing point is worth noting: if p and q are distinct states in M but p and q happen to go to the same state n after reading a, then there is no transition starting $\{p,r\} \overset{a}{\rightsquigarrow}$ in the graph we build. We can't have a node of the form $\{n,n\}$; indeed $\{n,n\}$ is not an unordered pair at all, it is just $\{n\}$.

14.10 Remark. An efficient implementation of Algorithm 26 will not explicitly construct the graph described there. A better idea is to construct the set D_0 explicitly, and then construct the set D by working *backwards*. Initially D is set D_0, then we update D in a loop:

> Consider each $\{q_i, q_j\} \notin D$ in turn
>
> For each such $\{q_i, q_j\}$, consider each alphabet symbol $a \in \Sigma$ if $q_i \xrightarrow{a} q_m$ and $q_j \xrightarrow{a} q_n$ and $\{q_m, q_n\} \in D$, add $\{q_i, q_j\}$ to D.

Algorithm 26: *DFA* State Equivalence

Input: a *DFA* $M = (Q, \Sigma, \delta, q_0, F)$
Output: the relation $\approx$ (as a set of pairs of states)
Let $G = (V, E)$ be the following graph.
The nodes V of G are the *unordered* pairs of (distinct) states from Q.
The edges E are directed and labeled with symbols from Σ; there will be an edge

$$\{p, q\} \overset{a}{\rightsquigarrow} \{r, t\}$$

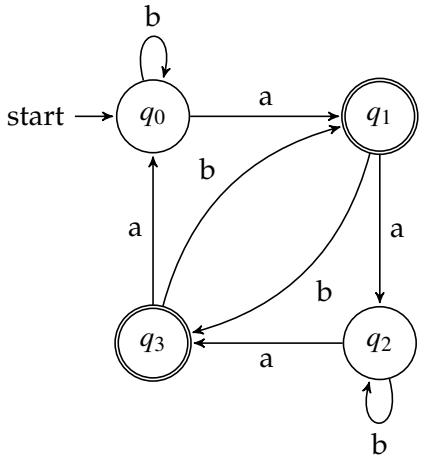
precisely if $p \xrightarrow{a} r$ and $q \xrightarrow{a} t$ are transitions in M.
Let $D_0 \subseteq V$ be this subset of nodes:

$$D_0 := \{\{r, t\} \mid (r \in F \text{ and } t \notin F) \text{ or } (r \notin F \text{ and } t \in F)\}.$$

Then let $D \subseteq V$ be the set of nodes $\{p, q\}$ such that there is a path in G from $\{p, q\}$ to some node in D_0.
return the complement of D: the set of all those pairs $\{u, v\}$ of states such that the node $\{u, v\}$ is *not* in the set D.

If we do this until there is no change in D, we have computed the complement of the $\approx$ relation.

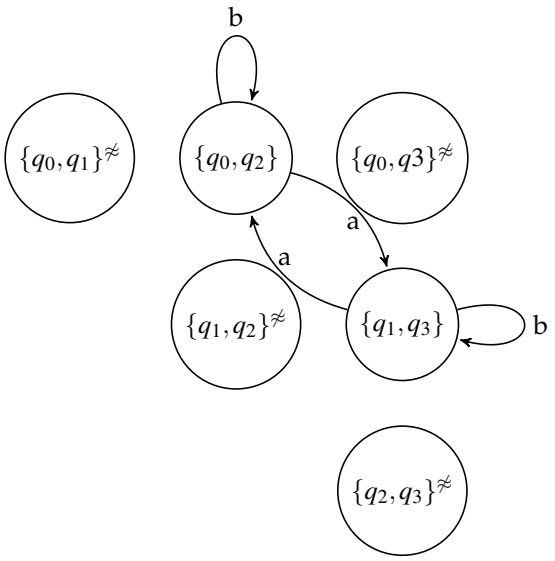
14.11 Example. Starting with



we observed in the last section that $q_0 \approx q_2$ and that $q_1 \approx q_3$. Let's check that this is what Algorithm 26 computes.

Here is our graph. We have indicated the initial set D_0, those pairs with exactly one accepting state in them, by tagging them with "$\not\approx$."

We have not bothered to indicate any edges out of nodes that are in D_0, since the whole point to the graph is discover nodes that have paths into D_0, and a D_0-node already has that: the empty path!



We can see that there are no paths leading from nodes which are not in the original D_0 (there are only two such!) leading into D_0.

So, the nodes that *do not* have paths into D_0 tell us the state-pairs that *are* $\approx$: $\{q_0, q_2\}$ and $\{q_1, q_3\}$.

14.12 Remark. If you are asked (for example in a homework problem!) to compute $\approx$ by hand for some moderate-size *DFA*, here is a suggested workflow, if you want to avoid writing down the whole pairs-of-nodes graph.

Certainly every node p is $\approx$ with itself, so we keep track of only the unordered pairs of *distinct* nodes.

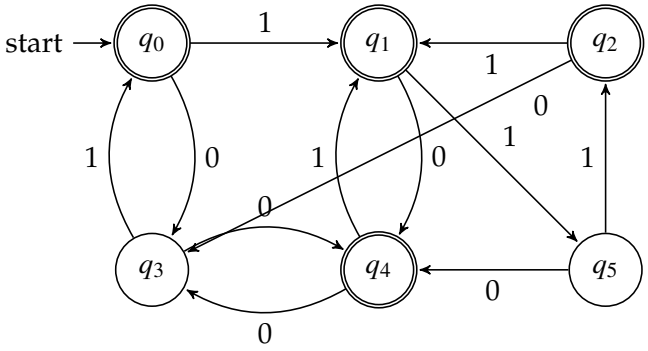
1. Write down the set X of pairs comprising the complement of the set D_0. This is the set of pairs $\{p, q\}$ such p and q are either both accepting or both non-accepting. Pairs in X have a chance of eventually being declared $\approx$.

2. Start building the $\{p, q\} \overset{a}{\rightsquigarrow}$ arrows out that set. If any such arrow would lead *outside* the current set X, eliminate $\{p, q\}$ from X. Such a pair is definitely not $\approx$.

   If you eliminate such a $\{p, q\}$ this may have a domino effect of eliminating other pairs, those that might point into $\{p, q\}$.

When you have finished this process you will have a maximal set X such that all the $\overset{a}{\rightsquigarrow}$ arrows stay inside X. This is our $\approx$ relation.

14.13 Example. Let's run the algorithm on the *DFA* from Example 14.4.



I'll ask you to draw the pictures. There are 6 states, so there are 15 pairs of distinct states, that is, 15 nodes in our graph:

$\{q_0,q_1\}$	$\{q_0,q_2\}$	$\{q_0,q_3\}$	$\{q_0,q_4\}$	$\{q_0,q_5\}$
	$\{q_0,q_2\}$	$\{q_0,q_3\}$	$\{q_0,q_4\}$	$\{q_0,q_5\}$
		$\{q_0,q_3\}$	$\{q_0,q_4\}$	$\{q_0,q_5\}$
			$\{q_0,q_4\}$	$\{q_0,q_5\}$
				$\{q_0,q_5\}$

The initial set D_0 is :

$$\{q_0,q_3\}\ \{q_0,q_5\}\ \{q_1,q_3\}\ \{q_1,q_5\}\ \{q_2,q_3\}\ \{q_2,q_5\}\ \{q_3,q_4\}\ \{q_4,q_5\}$$

Now we record the edges out of the non-D_0 nodes. Taking them in turn:

- Out of $\{q_0,q_1\}$ we have an edge labeled 0, to $\{q_3,q_4\}$.
- Out of $\{q_0,q_2\}$ we have no edges, since 0 takes these both to q_3 and 1 takes these both to q_1.
- Out of $\{q_0,q_4\}$ we have no edges (why?)
- Out of $\{q_1,q_2\}$ we have an edge labeled 0, to $\{q_3,q_4\}$.
- Out of $\{q_1,q_2\}$ we have an edge labeled 1, to $\{q_1,q_5\}$.
- Out of $\{q_1,q_4\}$ we have an edge labeled 0, to $\{q_3,q_4\}$.
- Out of $\{q_1,q_4\}$ we have an edge labeled 1, to $\{q_1,q_5\}$.
- Out of $\{q_2,q_4\}$ we have no edges
- Out of $\{q_3,q_5\}$ we have an edge labeled 1, to $\{q_0,q_2\}$.

That's our graph. Now we look to see which nodes have *paths* leading into D_0, and it turns out that the only nodes which do not are $\{q_0,q_2\}$, $\{q_0,q_4\}$, $\{q_2,q_4\}$, and $\{q_3,q_5\}$. So that is our $\approx$ relation, as we computed by cleverness originally.

14.4 The Collapsing Quotient of a *DFA*

We have seen in Section 14.2 how to compute the $\approx$ relation for a *DFA*. Now we'll use it to build a minimization of that *DFA*.

14.14 Definition (The Quotient Construction)**.** Let $M = (Q, \Sigma, \delta, q_0, F)$ be a *DFA*. For each $q \in Q$ let $[q]$ denote the equivalence class of q with respect to the relation $\approx$. We define the *DFA* $M_\approx$ as follows.

- the states of $M_\approx$ are the equivalence classes of states of M;
- the start state of $M_\approx$ is $[q_0]$;
- a state $[q]$ of $M_\approx$ is accepting in $M_\approx$ precisely if q is accepting in M;
- the transition function $\delta_\approx$ of $M_\approx$ is given by $[p] \xrightarrow{a} [p']$ just in case $p \xrightarrow{a} p'$ in M.

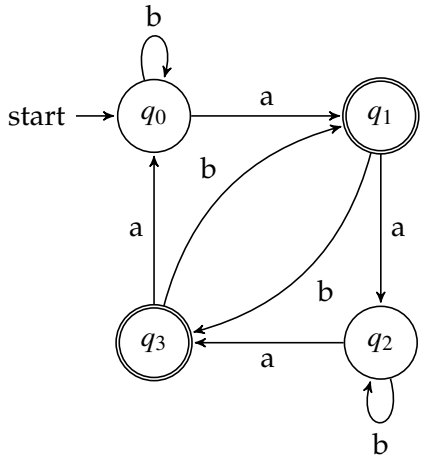
The problem of "well-definedness"

There is a technical matter to dispose of first. It is important to understand that the construction in Definition 14.14 makes sense only in light of the properties of $\approx$ in Lemma 14.8. Specifically, we need $\approx$ to be an equivalence relation in order to define the states of $M_\approx$, and we need the second property in Lemma 14.8 in order for the definition of $\delta_\approx$ to be sensible. To understand this second remark, suppose—to the contrary—that we could have two states $p \approx q$ *but* there were an $a \in \Sigma$ with $p \xrightarrow{a} p'$ in M, and $q \xrightarrow{a} q'$ in M, but $p' \not\approx q'$ in M.

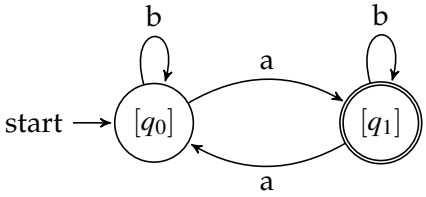
Then, in the collapse-*DFA* we are defining, $[p]$ and $[q]$ would be the same state (since $p \approx q$), but where would we transition to from this state on input a? One the one hand, the definition says we should go to $[p']$, because $p \xrightarrow{a} p'$, but the definition also says we should go to $[q']$, because $q \xrightarrow{a} q'$, If $[p']$ and $[q']$ are not the same state in the collapse automaton, things wouldn't make sense. But Lemma 14.8, part 2, is precisely what assures us that this doesn't happen.

This kind of issue comes up all the time whenever one is working with equivalence classes, that is, whenever one wants to make some sort of definition concerning classes, where there can be more than one "name" for a given class. One needs to be sure that the definition does not say something different if one chooses different names for the same class. People speak of making sure that a notion is *well-defined.*

14.15 Example. Starting with

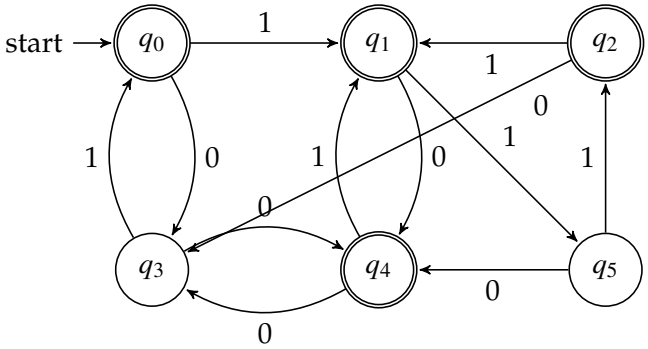


we observed in the last section that $q_0 \approx q_2$ and that $q_1 \approx q_3$. So when we collapse we get



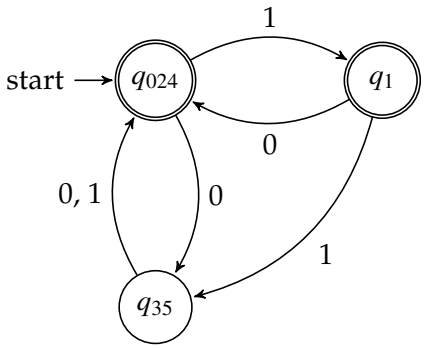
Important. In the picture we labelled the left state as "$[q_0]$" but we could just as easily have written "$[q_2]$"; the same goes for "$[q_1]$" versus "$[q_3]$". Make sure you understand this: $[q_0]$ and $[q_2]$ are different names for *exactly the same thing!*

14.16 Example. Let's look once again at the *DFA* from Example 14.4.



In Example 14.13 we computed the pairs $\{i, j\}$ of $\approx$ states. Namely: $q_0 \approx q_2$ and $q_0 \approx q_4$ and $q_3 \approx q_5$.

Notice that when we collapse these states, since q_0 is equivalent to both q_2 and q_4, they collapse down to a single state. Here is the *DFA* we get.



14.5 $M_\approx$ Does the Right Thing

The next Lemma says that when we collapse we don't change the ultimate behavior of a *DFA*.

14.17 Lemma. $L(M_\approx) = L(M)$

Proof. Proving this is equivalent to proving, when s is the start state, that if $s \xrightarrow{x}\!\!\!\!\rightarrow p$ in M, then $[s] \xrightarrow{x}\!\!\!\!\rightarrow [p]$ in $M_\approx$.

Now, this looks just like the definition of the transition relation in $M_\approx$ except that it speaks about strings x instead of single letters a. So it is natural to use induction over x.

When $x = \lambda$, things are easy, since $s \xrightarrow{\lambda}\!\!\!\!\rightarrow s$in M and indeed $[s] \xrightarrow{\lambda}\!\!\!\!\rightarrow [s]$in $M_\approx$.

When $x = ya$, for $y \in \Sigma^*, a \in \Sigma$, the computation in M looks like $s \xrightarrow{y}\!\!\!\!\rightarrow p' \xrightarrow{a} p$ for some p'. By induction hypothesis, $[s] \xrightarrow{y}\!\!\!\!\rightarrow [p']$ in $M_\approx$. By the definition of transition in $M_\approx$ we have $[p'] \xrightarrow{a} [p]$. Putting these together we conclude $[s] \xrightarrow{x}\!\!\!\!\rightarrow [p]$ in $M_\approx$. ///

14.18 Check Your Reading. *Why did we choose to write x in the induction step as ya instead of ay?*

14.19 Check Your Reading. *Explain why it is* not *true that $s \xrightarrow{x}\!\!\!\!\rightarrow p$ in M, if and only if $[s] \xrightarrow{x}\!\!\!\!\rightarrow [p]$ in $M_\approx$ Having seen that, go back and make sure you believe that proving just the one direction, as we did in the proof above, is sufficient for proving $L(M_\approx) = L(M)$.*

14.6 $M_\approx$ is special

Some natural questions come up now. Let M be any *DFA*, with no unreachable states. Let K be $L(M)$. And suppose that we compute $M_\approx$.

> *What happens if we collapse $M_\approx$ itself?*
>
> *Is $M_\approx$ a smallest possible DFA for K?*
>
> *Suppose we had started with a different DFA N recognizing K. How are the respective collapses $M_\approx$ and $N_\approx$ related?*

Amazingly, the answer to these questions are the nicest ones we could imagine. For all the *DFAs* that accept K, their collapses all have the same number of states, namely the number of $\equiv_K$ equivalence classes. And this is the minimum number of states that a *DFA* for K could have. Both of those remarks follow from Corollary 14.21.

The *DFA* $M_\approx$ has the following nice property, which can be stated as: $\equiv_K$-equivalent strings end up in $\approx$-equivalent states. Note that this is a converse to the K-Equivalence Lemma 12.10, but it only holds for collapsed *DFAs*!

14.20 Lemma. *Let $M_\approx$ be the output of Algorithm 26 for some DFA M. For any strings x and y, if $x \equiv_K y$ then $\hat{\delta}_{M_\approx}(s,x) = \hat{\delta}_{M_\approx}(s,y)$*

Proof. We want to show that if $x \equiv_K y$ and $s \xrightarrow{x}\!\!\!\!\rightarrow p$ and $s \xrightarrow{y}\!\!\!\!\rightarrow q$ then $p \approx q$.

Let $z \in \Sigma^*$; we have to show that if $p \xrightarrow{z}\!\!\!\!\rightarrow r$ and if $q \xrightarrow{z}\!\!\!\!\rightarrow r'$ then $r \in F$ iff $r' \in F$.

But since $s \xrightarrow{x}\!\!\!\!\rightarrow p \xrightarrow{z}\!\!\!\!\rightarrow r$, $xz \in K$ iff $r \in F$. And since $s \xrightarrow{y}\!\!\!\!\rightarrow q \xrightarrow{z}\!\!\!\!\rightarrow r'$, $yz \in K$ iff $r' \in F$. Since we are assuming $x \equiv_K y$, we have $xz \in K$ iff $yz \in K$, so indeed $r \in F$ iff $r' \in F$. ///

14.21 Corollary. *Suppose M is a DFA with no inaccessible states. Let K be $L(M)$. Then the number of states of $M_\approx$ is equal to the number of $\equiv_K$ equivalence classes.*

Proof. A way to think about $\hat{\delta}_{M_\approx}(s,-)$ is that it is a function from strings to states of $M_\approx$. When there are no inaccessible states in M this function is surjective.

But Lemma 14.20 says that this really makes a map from the $\equiv_K$ equivalence classes to the states of $M_\approx$, since strings in the same class will go to the same $M_\approx$-state.

So we have a surjective function from $\equiv_K$ equivalence classes to the states of $M_\approx$, so the number if states is no larger than the number of $\equiv_K$ classes.

Since $L(M_\approx) = K$ we can use Corollary 12.11 to conclude that the number states is actually equal to the number of classes.

///

It follows that $M_{\approx}$ cannot be collapsed.

14.22 Corollary. *The collapse of $M_{\approx}$ is $M_{\approx}$, that is $(M_{\approx})_{\approx} = M_{\approx}$*

Proof. If $(M_{\approx})_{\approx}$ were not equal to $M_{\approx}$, it would have fewer states than $M_{\approx}$, but would contradict that every *DFA* for K must have at least as many states as there are $\equiv_K$-classes. ///

All Minimal *DFAs* for K Are the Same

We can say something stronger. Namely, if you take any two *DFAs* for K and collapse them, you will always get the *same* resulting *DFA*. Of course the states might have different names, but that is a superficial difference that we won't care about. What we are claiming is that any two such *DFAs* can be made identical just by renaming states. This is a stronger statement that saying that the two have the same number of states: we are saying that there is a bijection between their states that preserves the structure of the transitions.

Rather than defining abstractly what that last notion means we will just describe the correspondence (if you know what an "isomorphism" is, that is what we will establish).

We will use the idea from Section 14.7. Let $M_1 = (\Sigma, Q_1, \delta_1, s_1, F_1)$ $M_2 = (\Sigma, Q_2, \delta_2, s_2, F_2)$ each be the result of having done a minimization construction, starting from some *DFAs* recognizing K. As usual, to avoid silliness, we will assume that $Q_1 \cap Q_2 = \emptyset$. This will be true as long as the original *DFAs* we started with had disjoint states.

Let $\approx$ be the stateEquivalence relation on $Q_1 \cup Q_2$ as defined in Section 14.7. Let's note a few things:

1. No two distinct states from the same M_i can be $\approx$ to each other. Since: otherwise, that M_i could be collapsed further.
2. No state in M_1 can be $\approx$ to more than one state in M_2 (and vice versa). Since: otherwise, those two states in M_2 would be $\approx$ to each other, contradicting the previous remark.
3. So every state in M_1 is $\approx$ to a unique state in M_2, thus the following function f defines a bijection from the states of M_1 to the states of M_2 :

$$f(q_1) = \text{ the state } q_2 \in Q_2 \text{ such that } q_1 \approx q_2 \ .$$

4. That's our bijection; to see the sense in which is preserves the structure of the *DFAs*, notice that

(a) The original start states are $\approx$: $s_1 \approx s_2$. Since: M_1 and M_2 recongnize the same language.

(b) If $q_1 \approx q_2$ and $q_1 \xrightarrow{a} r_1$ in M_1 and $q_2 \xrightarrow{a} r_2$ in M_2, then $r_1 \approx r_2$. Since: if r_1 and r_2 could be distinguished by some word z then q_1 and q_2 would be distinguished by az.

Putting all that together we see that M_1 and M_2 have exactly the same structure, based only on the fact that they are collapsed *DFAs* for the same language.

14.7 Application: Testing Equivalence of *DFAs*

Let's see how $\approx$ can be used to test whether two *DFAs* accept the same language. This subsection is adapted from Section 4.4 of [HMU06].

Suppose M_1 and M_2 are *DFAs* over the same alphabet Σ. How might we test whether $L(M_1) = L(M_2)$? Given any single string x we can test whether M_1 and M_2 agree on x by just running the machines on x. But we certainly can't decide $L(M_1) =^? L(M_2)$ by exhaustively testing all strings, since there are infinitely many.

We will see in a later section an important algorithm to answer this question using the product construction and an Emptiness Test for *DFAs*. But it is interesting to see that our algoritm for computing $\approx$ also gives an algorithm, as follows.

Write $M_1 = (\Sigma, Q_1, \delta_1, s_1, F_1)$ $M_2 = (\Sigma, Q_2, \delta_2, s_2, F_2)$ and assume—without loss of generality—that the states of M_1 and M_2 are disjoint from each other. Notice that, conceptually, the definition of state equivalence in Definition 14.2 makes sense as a binary relation between any two states q and q' from $Q_1 \cup Q_2$, even though the aren't from the same *DFA*. To be careful and obey the rules though, we proceed as follows.

Define a new *DFA* $M_1 = (\Sigma, (Q_1 \cup Q_2), (\delta_1 \cup \delta_2), s_1, (F_1 \cup F_2))$ by just bundling the two original *DFAs* together, and declaring s_1 to be the start state. It doesn't matter that we chose s_1 rather than s_2, and of course the M_2-part of our new automata is inaccessible, but still, this M is a legal *DFA*. And now the $\approx$-relation makes perfect sense on this M. The punch line is this: the two machines will accept the same language precisely if the two start states s_1 and s_2 satisfy $s_1 \approx s_2$.

14.23 Lemma. *Let $M_1 = (\Sigma, Q_1, \delta_1, s_1, F_1)$ $M_2 = (\Sigma, Q_2, \delta_2, s_2, F_2)$ be as just described, and let $\approx$ be the state equivalence relation as just described. Then $s_1 \approx s_2$ if and only if $L(M_1) = L(M_2)$.*

14.8 Collapsing *NFAs*?

A natural question at this point is: what happens if we try to do the collapse construction on nondeterministic automata? The first thing to observe is that it is not obvious what the definition of stateEquivalence might be in the presence of nondeterminism: think about it.

The next observation is the killer, though: it is not true that minimal-sized *NFAs* for a given language are unique. That is, there can be two NFAS N_1 and N_2 such that $L(N_1 = L(N_2)$ and each of N_1 and N_2 have a minimal number of states for their language, yet N_1 and N_2 are not isomorphic.

14.24 Example.

Once we notice this we can see that the collapsing construction for *DFAs* cannot possibly carry over to NFAs without some changes.

It turns out that there is quite a nice generalization of the idea of collapsing that works for NFAs, based on the idea of *bisimulation.* This is an important idea in the theory of concurrent processes. But we cannot go into it here.

14.9 Problems

167. EquivStates1

(From [Koz97])

For the *DFAs* below, given in table form, list the equivalence classes for the equivalence relation $\approx$.

a)

		a	b
start	1	1	4
	2	3	1
accepting	3	4	2
accepting	4	3	5
	5	4	6
	6	6	3
	7	2	4
	8	3	1

b)

		a	b
start, accepting	1	3	5
accepting	2	8	7
	3	7	2
	4	6	2
	5	1	8
	6	2	3
	7	1	4
	8	5	1

c)

		a	b
start, accepting	1	2	5
accepting	2	1	4
	3	7	2
	4	5	7
	5	4	3
	6	3	6
	7	3	1

d)

		a	b
start, accepting	1	2	6
accepting	2	1	7
	3	5	2
	4	2	3
	5	3	1
	6	7	3
	7	6	5

e)

		a	b
start	1	1	3
accepting	2	6	3
	3	5	7
accepting	4	6	1
	5	1	7
accepting	6	2	7
	7	3	3

f)

		a	b
start, accepting	1	2	5
accepting	2	1	6
	3	4	3
	4	7	1
	5	6	7
	6	5	4
	7	4	2

g)

		a	b
start, accepting	1	3	5
accepting	2	8	7
	3	7	2
	4	6	2
	5	1	8
	6	2	3
	7	1	4
	8	5	1

h)

		a	b
start, accepting	1	2	5
accepting	2	1	4
	3	7	2
	4	5	7
	5	4	3
	6	3	6
	7	3	1

i)

		a	b
start, accepting	1	2	6
accepting	2	1	7
	3	5	2
	4	2	3
	5	3	1
	6	7	3
	7	6	5

j)

		a	b
start	1	1	3
accepting	2	6	3
	3	5	7
accepting	4	6	1
	5	1	7
accepting	6	2	7
	7	3	3

k)

		a	b
start, accepting	1	2	5
accepting	2	1	6
	3	4	3
	4	7	1
	5	6	7
	6	5	4
	7	4	2

168. EquivStates2

(From [Koz97])

Consider the *DFAs* from Problem 167 for which you computed the $\approx$-relation. For each machine, tell which states are accessible, then build the automaton obtained by removing inaccessible states and collapsing equivalent states.

169. ComputeDistinguisher

Make an adjustment to Algorithm 26 so that at the end of the algorithm, if states q_i and q_j are $\not\approx$, we can compute (quickly) a particular string x such that x distinguishes them.

Hint. Record more information in the table.

170. ConstructMinimal

For each language E, construct the minimal *DFA* M such that $L(M) = E$.

1. $E = (a+b)^*ab$.
2. $E = ((a+b)(a+b))^* + (a+b)^*b$
3. $E = (a+b)^*aba(a+b)^*$
4. $E = a^*b^* \ + \ b^*a^*$

Don't be clever and invent a minimal *DFA* from scratch - treat these as problems in finding minimal *DFAs* systematically. That is, make a naive *NFA*, refine it to a *DFA* and use the minimization algorithm. Remember that you cannot do the collapse construction on *NFAs* directly.

171. WellDefinedness

This problem is designed to help you understand the well-definedness requirement, used crucially in Definition 14.14.

a) Define the following relation $\equiv_5$ on the set $\mathbb{Z}$ of integers:

$$m \equiv_5 n \quad \text{if } (m-n) \text{ is divisible by } 5.$$

Show that $\equiv_5$ is an equivalence relation.

Having done that, let's write $[n]$ for the equivalence class that n belongs to. Let $\mathbb{Z}_5$ be the set of equivalence classes.

Note that, for example, $[2] = [7] = [-13] = [257] = \ldots$ All of these are the same class, that is, the same element of $\mathbb{Z}_5$.

Now let us (try to) define a binary relation $\preceq$ on $\mathbb{Z}_5$ by the rule:

$$[m] \preceq [n] \quad \text{if } m \leq n \text{ in } \mathbb{Z}.$$

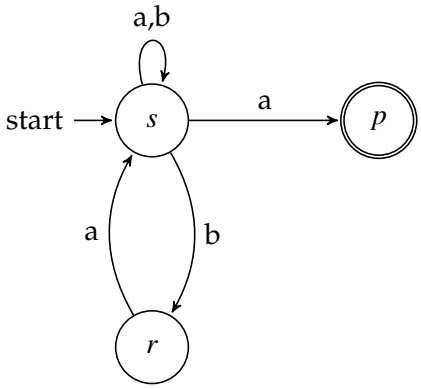
What goes wrong? Relate this to the construction of $M_\approx$.

b) Write down a *DFA* $M = (\Sigma, Q, \delta, S, F)$ (at random). Define the binary relation $\cong$ on states on M that holds between q_1 and q_2 if they are the same distance (the length of a minimal path) from the start state. Prove that $\cong$ is an equivalence relation. So it partitions Q. Explain why it does not satisfy property 2 of Lemma 14.8. (Actually, it is possible that it *does* satisfy it, by accident, if so, start again with a different *DFA* ...)

Assuming it does not satisfy property 2: try to build a *DFA* by collapsing states according to $\cong$: What goes wrong? Relate this to the construction of $M_{\approx}$.

172. StateEquivNFA?

To help drive home the idea that we cannot easily generalize the minimization idea to *NFAs*, consider the following *NFA* N.



1. Show that N is *not* a minimal-size *NFA* for $L(N)$, but
2. Show that for any two states q_1, q_2 of N there is a string z that distinguishes them.

This is whay we have only explored the $\approx$ relation for *DFAs*, not *NFAs*.

Chapter 15

The Myhill-Nerode Theorem

Our work in Sections 14.2 and 14 has told us, so far, that if K is a regular language then

1. the number of $\equiv_K$-equivalence classes for K is finite,
2. if M is any *DFA* recognizing K then we can collapse M to get a *DFA* $M_{\approx}$ recognizing K with the same number of states as there are $\equiv_K$-equivalence classes, and
3. if M and N are any *DFAs* recognizing K then $M_{\approx}$ and $N_{\approx}$ are the same up to renaming of states.

This is a very pretty picture. There is one question remaining, though. Suppose the number of $\equiv_K$-equivalence classes is finite. Is K necessarily regular? In this chapter we prove this (the converse of the main result in Section 12).

This will show, by the way, that the technique of finding an infinite distinguishable collection will always work to show languages not regular. That is, if K is not regular there will always be such an infinite distinguishable collection. (Whether you can be clever enough to find it is another question, of course).

To prove our result we need to look at the distinguishability relation a little harder. We start by recalling that $\equiv_K$ is an equivalence relation on Σ^*, that is, it is reflexive, symmetric, and transitive. Since $\equiv_K$ is an equivalence relation, it partitions Σ^* into classes. Recall that we write $[x]_K$ for the equivalence class of a string x. That is,

$$[x]_K = \{y \mid x \equiv_K y\}$$

Do yourself a favor and do the problems Problem a) through Problem g) at the end of this chapter that ask you count equivalence classes before reading further. After you do that you should be able to make a conjecture about what's coming next.

15.1 Finite index languages are regular

Since we will be focusing now mainly on languages K that have finitely many $\equiv_K$ classes, it will be convenient to introduce the notation "Index(K)" for the number of classes If K happens not to have finitely many classes we just agree that Index(K) stands for "infinity".[1]

What we will show is that if Index(K) is finite then K is regular. Even more, such a K has an recognizing DFA with Index(K)-many states.

Now here is the beautiful idea. When Index(K) is finite then we will show explicitly how to build a DFA recognizing K. The trick is to let the states of the DFA be the equivalence classes of $\equiv_K$ themselves!

There are two times before that we have built automata by taking a novel notion of what a "state" is. In the product construction we build an *FA* whose states are ordered pairs of states from two given *FAs*. In the subset construction we build a *DFA* whose states are subset of a given *NFA*. Now we are going to build a *DFA* whose states are certain (often infinite) sets of strings. This is a bit wilder than the other two constructions, but the essential point is the same: states are a mathematical abstraction and you can construct them out of anything you like.

15.1 Definition (The Minimal *DFA* D_K)**.** Suppose $K \subseteq \Sigma^*$ is a language with finitely many $\equiv_K$-classes. Define $D_K = (\Sigma, Q, \delta, s, F)$ as follows.

- $Q = \{[x]_K \mid x \in \Sigma^*\}$
- The transition relation δ is given by:
  for each state $[x]_K$ and each input symbol a, $[x]_K \xrightarrow{a} [xa]_K$.
- $s = [\lambda]_K$
- $F = \{[x]_K \mid x \in K\}$

Our goal is to show that this definition makes a *DFA*, and that $L(D_K) = K$.

The problem of "well-definedness" again

Once again we have that issue of well-definedness, as introduced in Section 14.4. We wrote the equivalence classes of $\equiv_K$ as

$$[x_1]_K,\ [x_2]_K,\ \ldots,\ [x_n]_K$$

But the choice of these x_i is not canonical, that is, there are usually many different "names" for the same equivalence class. Indeed, whenever $x \equiv_K y$ we have that $[x]_K$

[1] It turns out that there are always countably many classes, but that will never matter to us.

is the same set as $[y]_K$.

15.2 Check Your Reading. *Go back to the problems above where you generated the equivalence classes for various languages. Start with language A there and write the classes as*

$$[x_1]_A,\ [x_2]_A,\ \ldots$$

for several different choices of x_1, x_2, etc. Do the same for languages $B, C, D, \ldots$ until you really understand this idea or until you get tired.

Having noticed the fact that any given equivalence class can have many different representatives (or "names") it then becomes important to be sure that the definition of δ, *which ostensibly depends on the choice of names* for the equivalence classes does not *really* depend on the names. In other words, we must show that if $[x]_K$ and $[y]_K$ are the same state, then our definition of the δ-function treats them the same. This amounts to proving that in writing

$$[x]_K \xrightarrow{a} [xa]_K.$$

if we replace the name x by some y which also names $[x]_K$, that is, $[x]_K = [y]_K$, then the result $[ya]_K$ is the same state as $[xa]_K$.

This isn't hard to prove, though. Here is the Lemma.

15.3 Lemma. *If $[x]_K = [y]_K$ then for every $a \in \Sigma$, $[xa]_K = [ya]_K$.*

Proof. To say that $[xa]_K = [ya]_K$ is to say that $xa \equiv_K ya$. So to prove that we consider an arbitrary $z \in \Sigma^*$ and argue that $xaz \in K$ iff $yaz \in K$. But this follows from the assumption that $x \equiv_K y$ since az is a test string for x and y. ///

Having proved Lemma 15.3 we can be confident that Definition 15.1 really does define a *DFA*.

Correctness of the Construction

Ok, now we know that we really defined a *DFA*. Let's know show that the *DFA* does the right job.

15.4 Theorem. *Let K and D_K be as in Definition 15.1. Then $L(M) = K$.*

Proof. To avoid notational clutter let's just write δ for δ_{D_K} and $\hat{\delta}$ for $\hat{\delta}_{D_K}$.

We want to show that for all strings x,

$$\hat{\delta}_{D_K}(s,x) \in F \text{ if and only if } x \in K$$

By the definition of F in D_K it suffices to show that for all strings x,

$$\hat{\delta}(s,x) = [x]$$

We prove that fact by induction on x.

When $x = \lambda$: this is immediate from the fact that the start state of D_K is $[\lambda]$

When $x = ya$ for $a \in \Sigma$: we compute

$$\begin{aligned} \hat{\delta}(s,ya) &= \delta(\hat{\delta}(s,y),a) && \text{definition of } \hat{\delta} \\ &= \delta([y],a) && \text{induction hypothesis} \\ &= [ya] && \text{definition of } \delta \text{ in } D_K \end{aligned}$$

///

Summarizing our work, we've proved the following

15.5 Theorem. *A language K is regular if and only if Index(K) is finite.*

Proof. One direction is just the contrapositive of Corollary 12.12. The other direction is the content of Definition 15.1 and Theorem 15.4. ///

15.6 Theorem. *The DFA D_K has a minimal number of states among all DFAs recognizing K.*

Proof. It suffices to show that no *DFA* recognizing K can have fewer than Index(K) states. But this follows from Corollary 12.11 ///

15.2 Relating Myhill-Nerode and Minimization

If we were to minimize the *DFA* D_K it would not change, since it already has the minimum possible number of states. We showed in Section 14 that all minimized *DFAs* for a language were the same, so this means that in fact all minimized *DFAs* for K are precisely D_K .

What's nice about this observation is that, for any language K with finitely many $\equiv_K$-classes, we have a canonical notion of a minimal *DFA* for K, without having to *start out* with some *DFA* in the first place. Keep in mind that the structure of D_K is determined purely combinatorial, as a quotient of Σ^* defined in terms of membership in K. So here we defined a machine "organically" out of Σ^* using K.

So let us collect all of our work on collapsing *DFAs*, indistinguishability, etc, into one place.

15.7 Theorem (Myhill-Nerode)**.** *A language K is regular if and only if Index(K) is finite. When Index(K) is finite, the unique minimal DFA D_K for K is obtained by the construction in Definition 15.1: this DFA has Index(K)-many states. If we start with any DFA M recognizing K, the collapsing quotient $M_\approx$ is a DFA isomorphic to D_K.*

15.3 Problems

173. MHExamples

a) Let $A = \{x \in \{a\}^* \mid x \text{ has length divsible by 3 }\}$.

Language A is regular. So it has finitely many $\equiv_K$-equivalence classes. List at least three strings in each class. Then construct a *DFA* for A. Finally—this is the most important part—compare the equivalence classes for the language with the states of your *DFA*.

b) Let $G = \{a^n b^m \mid n,m \geq 0 \text{ and } n+m \text{ is divisible by 3 }\}$. List at least three strings in each $\equiv_G$-equivalence classes. class. Then construct a *DFA*. Compare the equivalence classes for the language with the states of your *DFA*.

c) Let $D = \{a^n b^m \mid n,m \geq 0\}$. List at least three strings in each $\equiv_D$-equivalence classes. class. Then construct a *DFA*. Compare the equivalence classes for the language with the states of your *DFA*.

d) For fixed k let $H_k = \{w \in \{a,b\}^* \mid \text{the } k\text{th symbol from the end of } w \text{ is } a\}$. List at least three strings in each $\equiv_H$-equivalence classes. class. Then construct a *DFA*. Compare the equivalence classes for the language with the states of your *DFA*.

e) Let $B = \{x \in \{a,b\}^* \mid x \text{ has length divsible by 3 }\}$.

List at least three strings in each $\equiv_B$-equivalence classes. class. Then construct a *DFA*. Compare the equivalence classes for the language with the states of your *DFA*.

f) Let $J = \{x \mid x \text{ has no consecutive repeated symbols}\}$ List at least three strings in each $\equiv_J$-equivalence class. Then construct a *DFA*. Compare the equivalence classes for the language with the states of your *DFA*.

g) Let $K = \{x \in \{a,b\}^* \mid x \neq \lambda \text{ and } x \text{ begins and ends with the same symbol}\}$. List at least three strings in each $\equiv_K$-equivalence class. Then construct a *DFA*. Compare the equivalence classes for the language with the states of your *DFA*.

174. ClassesRegular

Let K be a language such that $\equiv_K$ has finite index.

a) Consider one of the equivalence classes of $\equiv_K$, call it C. Note that C is a set of strings, so it is a language in its own right. Prove that C is regular.

Hint. Problem 135 will help.

b) Now suppose you are given an arbitrary *DFA* recognizing K. Explain how to algorithmically compute regular expressions for each of the $\equiv_K$ equivalence classes.

c) For each of the languages L in problems Problem a) through Problem g) do the following:

1. Write a little box corresponding to each equivalence class, and put at least three representatives from that class in your box;
2. Using those boxes as states, draw a picture of the DFA obtained by Definition 15.1.
3. For each of the equivalence classes C, find a regular expression denoting C.

Part III

Context-Free Languages

Chapter 16

First Examples : Context-Free

16.1 Context-Free Grammars

Grammars are systems for generating strings of symbols. We define a *start symbol* and some *rules* that describe how to transform strings into new strings. More than one rule might apply to an string; if so we just choose one. By iterating the rules (this is called a *derivation*) until we can't go any further we end up with a final string. Because we have a choice in which rules to apply in a given "session" there are usually lots of different strings that can be derived.

16.1.1 Natural Language

Here is a very simple grammar.

16.1 Example. Here are the rules:

$$\begin{aligned} S &\to N\,V \\ N &\to \text{Kim} \\ N &\to \text{Sandy} \\ V &\to \text{walks} \\ V &\to \text{talks} \end{aligned}$$

The convention (though it is not required) is to use S as the start symbol.

Here is a derivation:

$$S \Longrightarrow N\,V \Longrightarrow \text{Kim}\,V \Longrightarrow \text{Kim talks}$$

In the first step we replaced S by the right-hand side of the only S-rule, arriving at "$N\,V$." In the first step we replaced N by the right-hand side of the first N-rule,

arriving at "Kim V." Note that the V just went along for the ride. In the third step we replaced V by the right-hand side of the second V-rule, arriving at "Kim talks."

For practice, make derivations of "Kim walks", "Sandy walks" and "Sandy talks".

It is intuitively clear that by adding more vocabulary and rules one can generate a rich fragment of natural language. Welcome to the world of linguistics.

In fact the entire field of grammars originated with the linguist Noam Chomsky and was borrowed by computer scientists in the 1960's. It's worth doing an internet search for Chomsky (*e.g.* have a look at his Wikipedia page): he's a fascinating guy.

The grammars we will study have a property that hits a sweet spot in the tradeoff between expressive power and manageability. The property is that every rule has a single symbol as the left-hand side. These grammars are called *Context-Free* grammars, abbreviated *CFGs*.

16.1.2 Arithmetic Expressions

(By the way, when "arithmetic" is used as an adjective as above, the stress is on the third syllable.)

Here is an example that feels different

16.2 Example (Simple Arithmetic Expressions)**.** Here is a context-free grammar G generating arithmetic expressions. For our first pass we assume that numbers are just single digits.

- The terminal alphabet Σ is $\{+,*,0,1,2,3,4,5,6,7,8,9\}$
- The variable alphabet V is $\{E,I\}$.
- The start symbol is E.
- The rules are

$$\begin{aligned} E &\to E+E \mid E*E \mid I \\ I &\to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{aligned}$$

Here is a derivation, deriving the string $2+5*3$.

$$\begin{aligned} E &\Longrightarrow E+E \Longrightarrow E+E*E \Longrightarrow I+E*E \Longrightarrow 2+E*E \\ &\Longrightarrow 2+E*I \Longrightarrow 2+E*3 \Longrightarrow 2+I*3 \Longrightarrow 2+5*3 \end{aligned}$$

For this grammar—in contrast to the first example—there are infinitely different derivations possible. Do a few more for our practice.

Here's a good question: is the set of strings generated by this grammar a regular language? The answer is no. You should be able to prove that after your work on regular languages.

We'll have a lot to say about variants of this grammar as we go.

16.1.3 Grammars Can Do More Than Finite Automata

What does this grammar generate?

16.3 Example.

$$S \to aSb$$
$$S \to \lambda$$

Don't be freaked out by having λ on the right side of a rule. It just means "replace the left-hand side by λ" which is just to say, "erase an occurrence of the left-hand side."

Let's try a few derivations

$$S \Longrightarrow \lambda$$
$$S \Longrightarrow aSb \Longrightarrow ab$$
$$S \Longrightarrow aSb \Longrightarrow aaSbb \Longrightarrow aabb$$

It shouldn't take long for you to realize that this grammar will generate our old friend $\{a^n b^n \mid n \geq 0\}$, which we know is not regular.

So context-free grammars can generate some languages that finite automaton cannot recognize. Are they *strictly* more powerful? That is, can every regular language be generated by a CFG? The answer is yes; we'll show this in Section 18.

16.2 Pushdown Automata

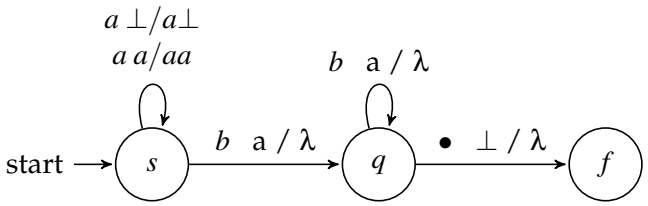
Go back to the $a^n b^n$ example. The obstacle to building a *DFA* for this language is that, as we read an input string, we cannot count how many as we've read in order to compare that to the number of subsequent bs.

But what if we allowed our automaton to have a stack memory? Then we could do the following.

- As we read as we push them onto the stack.

- Once we start reading bs, we pop as off of the stack.
- If we run out of as on the stack before finish reading bs, then there are more bs than as in the input, so we reject.
- If we have as left on the stack when as finish reading bs, then there are more as than bs in the input, so we reject.
- If we see any as after we have started reading bs then we reject.
- If none of that happens, and we run out of as on the stack just when we run out of input, then we accept.

16.4 Example. Here is a picture for a *PDA* P recognizing $\{a^n b^n \mid n \geq 1\}$. The notation decorating arcs is more complex than a *DFA* because we have to talk about the stack. If you can't easily match the notation with the prose description above, don't worry about it for now.



There is some work to be done to make this more formal, of course. We do that work in Section 24. We re-do the above example formally there (Example 24.2).

16.3 Context-Free Grammars Can't Do Everything

Here is an example of a language that cannot be generated by a context-free grammar (and cannot be recognized by a *PDA*):

$$\{a^n b^n c^n \mid n \geq 0\}$$

We prove this in Section 26.

For what it is worth, here is a grammar that generates this language. This grammar is *not* context-free.

$$\begin{aligned} S &\to abc \\ S &\to aXbc \\ Xb &\to bX \\ Xc &\to Ybcc \\ bY &\to Yb \\ aY &\to aaX \\ aY &\to aa \end{aligned}$$

16.4 Looking Ahead

Here are some important themes about this part of the course.

1. There are algorithmic techniques for building grammars, we don't have to be smart all the time.
2. There are languages that *cannot* be generated by context-free grammars.
3. Every pushdown automaton P has a context-free grammar G that generates the same language that P recognizes, and

   every context-free grammar G has a pushdown automaton P that recognizes the same language that G generates, and

   we can algorithmically translate back-and-forth between pushdown automata and context-free grammars.
4. There are algorithmic procedures for testing whether **certain** properties of context-free grammars hold. Context-Free grammars are not as nice as finite automata is this regard. (But still better than arbitrary programs.)
5. If we are given a context-free grammar G that generates a language we are interested in, for example, a programming language, we can algorithmically decide whether some string p is generated by G. This is the *parsing problem*. It is the first main part of a compiler.
6. If we are given a context-free grammar G that generates a language we are interested in, for example, a programming language, G be used to help assign *meanings* to the strings generated by G. For example we may want to evaluate expressions, or generate code for statements.

16.5 Problems

175. NLPlay

Play around with enriching the natural-language example, to get more interesting sentences.

Chapter 17

Context-Free Grammars

A context-free grammar (typically abbreviated CFG) is a formalism for generating strings. It does so using *rules* that rewrite strings, starting with a special *start symbol*. It is useful to allow other, auxiliary symbols to hierarchically structure the rewriting. So we make a distinction between this auxiliary alphabet, of *variables* and the alphabet of *terminals* over which we build our output strings. Here is the formal definition.

17.1 Definition. A *context-free grammar* is a 4-tuple

$$G = \langle \Sigma, V, P, S \rangle$$

where

- V is a finite set, called the set of *variables*[a]
- Σ is a finite set, called the *terminal alphabet*
- $S \in V$ is distinguished variable, called the *start symbol*
- P is a set of *rules*[b], of the form

$$X \to \beta$$

  where $X \in V$ and β is an arbitrary string over $V \cup \Sigma$.

[a] Some authors call variables *nonterminals*
[b] Some authors call rules *productions*

17.2 Example. Here is a tiny grammar:

- The terminal alphabet Σ is $\{a, b\}$

- The variable alphabet V is $\{S\}$.
- The start symbol is S.
- The rules are

$$\begin{aligned} S &\to aS \\ S &\to Sb \\ S &\to \lambda \end{aligned}$$

Usually we allow ourselves to present grammars less pedantically, by just giving the rules. The variable alphabet V can be inferred: it's just the set of symbols that show up as the left hand sides of rules (unless the grammar-writer has, perversely, put a variable in the grammar with no defining rules). The alphabet Σ can be inferred: it's just the set of symbols in the grammar that are not variable symbols. The start symbol needs to be declared explicitly. But, if the grammar writer uses "S" as one of the variables, and doesn't say otherwise, they will typically not bother to say out loud that S is the start symbol.

A very convenient shorthand is to communicate more than rule with the same left-hand side by writing them on a single line with the right-hand sides separated by "|". So another way to write down this grammar would be just to write the rules as

$$S \to aS \mid Sb \mid \lambda$$

We had a first look at this next example in Section 16.1.2.

17.3 Example (Simple Arithmetic Expressions)**.** Here (again) is a context-free grammar G generating arithmetic expressions. For our first pass we assume that numbers are just single digits.

- The terminal alphabet Σ is $\{+,*,0,1,2,3,4,5,6,7,8,9\}$
- The variable alphabet V is $\{E,I\}$.
- The start symbol is E.
- The rules are

$$\begin{aligned} E &\to E+E \mid E*E \mid I \\ I &\to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{aligned}$$

17.1 Derivations

The purpose of grammars is to generate strings of terminals. In order to so we generate, along the way, strings of mixed terminals and variables.

17.4 Definition (Derivation in a CFG)**.** Let G be a CFG, and let σ be a string over $(V \cup \Sigma)$. If there is a rule $X \to \beta$ in G, and σ is of the form $\sigma_1 X \sigma_2$ then we may make a *derivation step* from X, obtaining the string $\sigma_1 \beta \sigma_2$. We write

$$\sigma_1 X \sigma_2 \Longrightarrow \sigma_1 \beta \sigma_2$$

If σ and τ are strings over $(V \cup \Sigma)$, a derivation from σ to τ, written $\sigma \Longrightarrow^* \tau$ is a finite sequence of 0 or more derivation steps from σ to τ. We sometimes say that τ can be *generated* from σ.

Here is a derivation of the string *aab* in the grammar of Example 17.2.

$$S \Longrightarrow aS \Longrightarrow aaS \Longrightarrow aaSb \to aab$$

Note carefully that we use a "$\to$" arrow to indicate a *rule* of a grammar, and we use a "$\Longrightarrow$" arrow to indicate a *step* in a derivation. The relationship between them is this: it is the rules that *justify* derivation steps. In the above example, we have the derivation step $aaS \Longrightarrow aaSb$ because of the rule $S \to Sb$. applied at the occurrence of S in aaS. You can think of the rules as commands in a program, and derivation steps as the progress of the state of a machine as it executes the program.

Note that although we defined derivations using strings that can be made up both variables and terminals, some derivations happen to generate *terminal* strings as a special case. The set of such terminal strings is the language of the grammar.

17.5 Definition (Language of a Grammar)**.** The language $L(G)$ generated by G is the set of *terminal* strings that can be generated from the start symbol:

$$L(G) \stackrel{\text{def}}{=} \{x \in \Sigma^* \mid S \Longrightarrow^* x\}$$

Go back our example derivation above

$$S \Longrightarrow aS \Longrightarrow aaS \Longrightarrow aaSb \Longrightarrow aab$$

Note that along the way we derive, for example, the string *aaS*. But we do *not* say that $aaS \in L(G)$. As we said above, the set $L(G)$ is by definition, that set of *terminal strings* that can be derived.

It is easy to see that the language generated by our simple example grammar is the set $\{a^n b^m \mid n, m \geq 0\}$.

17.6 Example. The order of the symbols in a rule matters! Suppose we changed the previous example to be

$$S \to aS \mid bS \mid \lambda$$

Convince yourself that this grammar generates *all* strings over $\{a,b\}$.

17.7 Definition (Context-Free Language)**.** A language L is *context-free* if there is some *CFG* G such that $L = L(G)$.

17.8 Examples. Based on the two previous *CFGs* we looked at the languages $\{a^n b^m \mid n,m \geq 0\}$ and $\{a,b\}^*$ are both context-free languages.

A natural question at this point is: what is the relationship between the context-free languages and the regular languages? The answer is: every regular language is context-free, but there are some context-free languages that are not regular. We prove these facts soon.

17.9 Example (Base-10 Numbers)**.** The set of base-10 representations of natural numbers is a context-free language. Here is a grammar them. We don't want them to start with a 0, so we use an auxilliary variable M, to stand for the part of the number after the leftmost digit.

$\Sigma = \{0,1,\ldots,9\}$

$V = \{N,M\}$

The start symbol is N. The rules are:

$$\begin{aligned} N &\to 1M \mid 2M \mid \ldots \mid 9M \\ M &\to 0M \mid 1M \mid 2M \mid \ldots \mid 9M \mid \lambda \end{aligned}$$

17.10 Example. Here is a derivation in the grammar of Example 17.3.

$$\begin{aligned} E &\Longrightarrow E+E \Longrightarrow E+E*E \Longrightarrow I+E*E \Longrightarrow 2+E*E \\ &\Longrightarrow 2+E*I \Longrightarrow 2+E*3 \Longrightarrow 2+I*3 \Longrightarrow 2+5*3 \end{aligned}$$

This derivation derives the string $2+5*3$. So we say that $2+5*3 \in L(G)$.

17.11 Example (Better Arithmetic Expressions)**.** In Example 17.9 we focused on how to generate decimal numbers. In Example 17.3 we focused on how to generate expressions using $+$ and $*$; and we just used single digits in the "base case" because we wanted to focus on the operators ($+$ and $*$). Context-free grammars are great at letting us modularize our concerns in this way: we can build a reasonably realistic grammar for arithmetic expressions by grafting the two previous grammars

together. Instead of the somewhat lame use of 0 through 9, embodied by I, we replace I by N, together with the rules defining N. Now we have all the infinitely many decimal numerals as part of our arithmetic expressions. Here is the resulting grammar (the start symbol is still E).

$$\begin{aligned} E &\to E+E \mid E*E \mid N \\ N &\to 1M \mid 2M \mid \ldots \mid 9M \\ M &\to 0M \mid 1M \mid 2M \mid \ldots \mid 9M \mid \lambda \end{aligned}$$

We will systematically explore this idea of building grammars in a modular way in Section 18.1.

17.2 Parse Trees

Grammars afford a completely different perspective from automata. Automata *accept* strings: they are input-only. Grammars *generate* strings: they are output-only. An additional important point is that grammars don't just generate strings, they impose a *structure* on strings.

17.12 Example. Go back to the derivation in Example 17.10. Here is another derivation in this same grammar.

$$\begin{aligned} E &\Longrightarrow E+E \Longrightarrow E+E*E \Longrightarrow E+E*I \Longrightarrow E+E*3 \\ &\Longrightarrow E+I*3 \Longrightarrow I+I*3 \Longrightarrow 2+I*3 \Longrightarrow 2+5*3 \end{aligned}$$

Note that this is a different *derivation* from the one in Example 17.10, even though it generates the very same terminal string.

So the same string can be generated by different derivations. Does this matter? Our resounding answer is: "sometimes". To make sense of that last sentence, we first define a new way to look at derivations: parse trees.

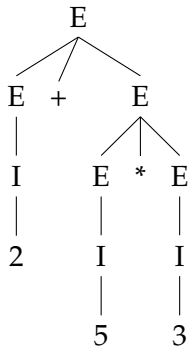
Parse trees, sometimes called derivation trees, are similar to derivations in that they witness the fact that a grammar can derive a string, but they are much more informative.

Fix a grammar $G = (\Sigma, V, P, S)$. A parse tree is a tree whose interior nodes are elements of V, whose root is S, whose leaves are elements of Σ, which is built according to P. Namely, the children of each interior node X of the tree correspond to an application of one of the rules of the grammar with left-hand-side X. Each derivation determines a (unique) parse tree.

It is a little tedious to give a formal definition of this but the idea will be totally clear once you see an example.

17.13 Example. Refer to the grammar in Example 17.3.

Here is a parse tree corresponding to the derivation given there, yielding $2+5*3$.



Think of the tree as "growing" downwards, guided by the derivation. It is not a coincidence that there are eight steps in the derivation in Example 17.3 and there are eight interior nodes of this tree!

17.14 Check Your Reading. *Make sure you see how this parse tree is gotten from the derivation.*

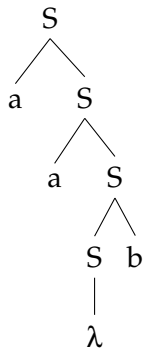
The reason that parse trees are important is that they impose a *structure* on a derived string. That is, when we have a parse tree we know more than just the fact that a string is derivable from a grammar: we know *how.*

Most of all, parse trees allow us to attach *meanings* to strings. When the expressions we are working with are code or computations, the parse trees tell us how to evaluate to expression. Look at the tree above: the natural way to evaluate this is to work bottom-up evaluating subtrees as we go, to arrive at 17.

17.15 Example. Refer to the grammar in Example 17.2, and the derivation

$$S \Longrightarrow aS \Longrightarrow aaS \Longrightarrow aaSb \rightarrow aab$$

Here is the corresponding parse tree



There is a slight awkwardness concerning the erasing rule $S \to \lambda$ that was used at the last step of the derivation. In writing down a derivation we can indicate the erasing rule by just erasing the symbol S. But when we draw a parse tree, it looks dumb to have a line coming down from S with nothing at the end. But if we don't draw a line from S there is nothing to indicate that invoked a rule (it looks like maybe we aren't finished). So what people do is draw a line ending in λ, even though λ isn't an alphabet symbol. So when you read off the "yield" of the parse tree, you just skip over the λ.

Parse Trees vs Derivations

Given any derivation, we can associate a unique parse tree in a straightforward way. But given a parse tree, there can be many corresponding different derivations. Derivations impose a linear ordering on "events" that is lost in displaying the parse tree.

17.16 Check Your Reading. *For the parse tree above for the string $2+3*5$, build several derivations, each of which would yield that tree.*

Is there a natural way to get a one-to-one correspondence between parse trees and derivations? Yes:

17.17 Definition. A *leftmost* derivation is one is which, at every step, the variable symbol that is leftmost in the current string is the one that is rewritten.

It is not hard to see that for each parse tree T there is a unique leftmost derivation yielding T. We don't want detour by proving this (a proof would only obfuscate the very simple idea). But these points are so important that we want to emphasize them clearly:

Facts.

- For a given derivation there is **one and only one** corresponding parse tree.
- For a given parse tree there can be **several** corresponding derivations.
- For a given parse tree there is **one and only one** leftmost derivation.

17.18 Check Your Reading. *The derivation shown in Example 17.3 is* not *leftmost. Build a leftmost derivation for $2*3+5$, one that yields the parse tree we showed.*

Now build a different *leftmost derivation for $2*3+5$. Build its parse tree. Since this leftmost derivation you just built is different from the first one you built, it will yield a different tree than the one we showed in the example.*

17.19 Check Your Reading. *Write down the obvious definition of "rightmost derivation" then satisfy yourself that for every parse tree T there is a unique rightmost derivation yielding T.*

Some authors prefer to treat derivations as primary. So in reading about context-free grammars in other places you may hear a lot of talk about leftmost derivations. But the parse trees are where the action is.

17.3 Lots of *CFG* Examples and Non-Examples

17.20 Example. Consider the language $\{a^n b^n \mid n \geq 0\}$

This is our standard example of a non-*regular* language. But it is easily seen to be context-free:

$$S \to aSb \mid \lambda$$

17.21 Example (Palindromes, Doubles, *etc.*)**.** A *palindrome* is a string x that reads the same backwards and forwards, in other words x is equal to its reversal: $w = w^R$. The set of palindromes is another example of a language that is context-free though it is not regular.

Here is a *CFG* generating the set of palindromes over the lower case letters $\Sigma = \{a, b, c, \ldots, z\}$:

$$S \to \lambda \mid aSa \mid bSb \mid cSc \mid \ldots \mid zSz$$
$$S \to a \mid b \mid c \mid \ldots \mid z \mid \lambda$$

The *even-length* palindromes can be described as the set of all strings of the form xx^R as x ranges over all strings.

$$EPal = \{xx^R \mid x \in \Sigma^*\}$$

It is easy to tweak the grammar above to get just the even-length palindromes (Problem 180). Thus $EPal$ is a context-free language.

Interestingly, the set of "doubled strings"

$$D = \{xx \mid x \in \Sigma^*\}$$

is *not* a context-free language, even though it seems closely related to $EPal$. This is tricky to prove. But it is a fact that there can be no *CFG* generating D.

And yet, the complement of D,

$$\overline{D} = \{w \mid w \text{ cannot be written as } xx \text{ for any } x \in \Sigma\}$$

is a context-free language. (This is not so easy to prove either ...)

On the other hand:

17.22 Example (A Non-Context-Free Language)**.** Consider the language $\{a^n b^n c^n \mid n \geq 0\}$ It can be shown that *there is no content-free grammar generating this language*. We give a proof in Section 26.1.

17.23 Example (Another Non-Context-Free Language)**.** The following language is *not* context-free. $\{a^n b^m c^n d^m \mid n,m \geq 0\}$ We omit the proof here.

17.24 Example (Arithmetic Expressions)**.** Here is a *CFG* generating the legal infix expressions over $+$ and $*$ and decimal numbers. The terminal alphabet is $\Sigma = \{+,-,*,/,0,1,\ldots,9\}$ The variable alphabet is $V = \{E,N,M\}$; the start symbol is E.

You will recognize that we have "inlined" the earlier grammar for decimal numbers. This is a point worth noting: in that earlier grammar N was the start symbol. Here it plays an auxiliary role. And in a larger grammar, say for an entire programming language, the expressions generated here will be auxiliary, so the E will not be a start symbol there.

$$\begin{aligned} E &\to E+E \mid E-E \mid E*E \mid E/E \mid N \\ N &\to 1M \mid 2M \mid \ldots \mid 9M \\ M &\to 0M \mid 1M \mid 2M \mid \ldots \mid 9M \mid \lambda \end{aligned}$$

The next two examples illustrate an important point. One can declare any collection of things an alphabet: they don't have to be things that you think of as individual "letters." In particular, when parsing a natural language it is common to treat *dictionary words* as comprising the terminal alphabet.

17.25 Example (A Programming Language)**.** Let's write a grammar for a little programming language. As in the baby grammar above for a fragment of English, we are going to take our terminal alphabet to consist of symbols that might, in another context, be thought of as strings. In fact what happens in a compiler is that there is an initial phase in which the input string of ASCII symbols is converted into a string of symbols over a different alphabet in which something like "while", which is a string of length 5 over the ASCII alphabet gets converted into a *single symbol* over a richer alphabet of "tokens".

In our little example we take our alphabet Σ of tokens to be

$$\Sigma = \{\texttt{while},\texttt{do},\texttt{if},\texttt{then},\texttt{else},:=,;,a,b,\ldots z,0,1,\ldots 9\}$$

This is a set with 7 + 26 + 10 elements in it.

We take the alphabet of variables to be

$$V = \{S, A, C, W\}$$

And here are the rules. Let us assume that you successfully write a grammar for programming-language identifiers in Problem d).

$$\begin{aligned}
S &\to A \mid C \mid W \mid S;S \\
A &\to I := E \\
W &\to \texttt{while}\, E\, do\, S \\
C &\to \texttt{if}\, E\, \texttt{then}\, S\, \texttt{else}\, S \\
C &\to \texttt{if}\, E\, \texttt{then}\, S \\
E &\to \text{[the rules for expressions Example 17.24]} \\
E &\to I \\
I &\to \text{[the rules for identifiers from Problem d)]}
\end{aligned}$$

17.26 Check Your Reading. *Make a derivation of the following program.*

$$\texttt{while}\, x\, do\, y := y+1;\, x := x-y$$

17.27 Example (Natural Language)**.** Let us declare the set of terminal symbols to be

$$\Sigma = \{a, the, boy, girl, dog, chased, heard, saw\}$$

and the set of variables to be

$$V = \{S, NP, VP, Det, N, Vrb\}$$

Think of NP as standing for "Noun Phrase", VP as standing for "Verb Phrase", Det as standing for "Determiner", N as standing for "Noun", and Vrb as standing for "Verb".

Now let's consider the rules

$$\begin{aligned}
S &\to NP \quad VP \\
NP &\to Det \quad N \\
VP &\to Vrb \quad NP \\
Det &\to a \mid the \\
N &\to boy \mid girl \mid dog \\
Vrb &\to ate \mid saw \mid heard
\end{aligned}$$

Here's a derivation

$$\begin{aligned}
S &\Longrightarrow NP \quad VP \\
&\Longrightarrow Det \quad N \quad VP \\
&\Longrightarrow the \quad N \quad VP \\
&\Longrightarrow the \quad boy \quad VP \\
&\Longrightarrow the \quad boy \quad V \quad NP \\
&\Longrightarrow the \quad boy \quad chased \quad NP \\
&\Longrightarrow the \quad boy \quad chased \quad the \quad N \\
&\Longrightarrow the \quad boy \quad chased \quad the \quad dog
\end{aligned}$$

17.28 Check Your Reading. *Make a derivation of the sentence*

$$the \quad dog \quad ate \quad a \quad boy$$

Are there finitely many sentences derivable in this grammar, or infinitely many?

17.29 Check Your Reading. *Continuing this example: Add the symbol* and *to* Σ *and add this rule to the grammar:*

$$S \rightarrow S \; and \; S$$

What new sentences can you derive?

Are there finitely many sentences derivable in this grammar, or infinitely many?

17.4 Problems

176. EasyStuff

a) What's the difference between a grammar and a language?

b) Can a grammar be infinite?

c) Does it make sense to talk about taking the complement of a grammar?

d) Does it make sense to talk about a grammar accepting a string?

177. DerivationPractice

Construct a derivation of the strings *racecar* and *kayak* in the grammar of Example 17.21.

178. MembershipPractice

(from Kozen) Consider the following grammar G:

$$\begin{aligned} S &\to ABS \mid AB, \\ A &\to aA \mid a \\ B &\to bA \end{aligned}$$

Which of the following strings are in $L(G)$? Prove your answers.

1. *aabaab*
2. *aaaaba*
3. *aabbaa*
4. *abaaba*

179. WhatLang

a) What language does the following grammar generate?

$$S \to aSbS \mid bSaS \mid \lambda$$

b) What language does the following grammar generate?

$$\begin{aligned} S &\to aB \mid bA \\ A &\to bAA \mid aS \mid a \\ B &\to aBB \mid bS \mid b \end{aligned}$$

Hint. Don't try to do problems like this by staring at the rules. Do a bunch of derivations, and the pattern will emerge for you.

180. MakeEasyCFGs

a) Make a CFG generating the even-length palindromes over $\Sigma = \{a, b\}$. *Hint:* modify the grammar in Example 17.21.

b) Construct a context-free grammar G generating $\{a^i b^j \mid i = 2j\}$

Hint. Start with a grammar for $\{a^i b^i \mid i \geq 0\}$, then make a little tweak.

c) Construct a context-free grammar G generating $\{a^i b^j c^k \mid i + j = k\}$

d) Imagine a programming language where the legal identifiers are: a string of lowercase letters and numbers, but which cannot start with a number. Make a regular CFG generating these strings.

181. PrefixArith

Let $\Sigma = \{+, *, a, b\}$. Let K be the language of prefix arithmetic expressions over Σ. Here are some strings in K.

$+\,a\,a$ $\qquad$ $+\,+\,a\,b\,*\,b\,a$

$+\,a\,*\,a\,b$ $\qquad$ $+\,a\,*\,a\,+\,b\,b$

Write a context-free grammar to generate K. Show parse trees and leftmost derivations for the strings bove.

Chapter 18

More on Context-Free Grammars

18.1 Closure Properties, or How to Build Grammars

With *NFAs* one of our themes was how to build complex machines out of simpler ones. We can play the same games with grammars. In fact the constructions themselves are easier. On the other hand, as we will see in Section 18.1, we can't do everything that we might like to do.

Union

Let two CFGs $G1 = (\Sigma, V_1, S_1, P_1)$ and $G1 = (\Sigma, V_2, S_2, P_2)$ be given, with disjoint variable alphabets.

We want to build a grammar that generates the union of the languages generated by G_1 and G_2.

Algorithm 27: CFG Union

Input: two CFGs $G_1 = (\Sigma, V_1, S_1, P_1)$ and $G_1 = (\Sigma, V_2, S_2, P_2)$ with $V_1 \cap V_2 = \emptyset$
Output: CFG G with $L(G) = L(G_1) \cup L(G_2)$

Let S be a symbol not in $V_1 \cup V_2$;
the set of variables of G is $(V_1 \cup V_2 \cup \{S\})$;
S is the start symbol of G ;
The rules of G are those of $P_1 \cup P_2$ together with the new rules $S \to S_1$ and $S \to S_2$

Proof of Correctness. To prove $L(G_1) \cup L(G_2) \subseteq L(G)$: let $w \in L(G_1) \cup L(G_2)$. Without loss of generality we may assume $w \in L(G_1)$, let $S_1 \Rightarrow^* w$ be a derivation of w in G_1. Then $S \Rightarrow S_1 \Rightarrow^* w$ is a derivation of w in G.

To prove $L(G) \subseteq L(G_1) \cup L(G_2)$: let $w \in L(G)$, via $S \Rightarrow^* w$.

The first step of this derivation is either $S \Rightarrow S_1$ or $S \Rightarrow S_2$; without loss of generality let us assume it is $S \Rightarrow S_1$. So we have $S \Rightarrow S_1 \Rightarrow^* w$ in G, and now *since we assumed that the variables of G_1 and G_2 were disjoint,* the part $S_1 \Rightarrow^* w$ of this derivation is actually a derivation of G_1. This means that $w \in L(G_1)$, so that $w \in L(G_1) \cup L(G_2)$ as desired. ///

What if the original grammars G_1 and G_2 did not have disjoint alphabets in the first place? It is not hard to see that we can always systematically rename the variables in a grammar without changing the language generated (remember that by definition the language generated by a grammar is a set of *terminal* strings).

Concatenation and Kleene star

Let two CFGs $G_1 = (\Sigma, V_1, S_1, P_1)$ and $G_2 = (\Sigma, V_2, S_2, P_2)$ be given, with disjoint variable alphabets.

We can build *CFGs* to capture the union and Kleene star of the languages generated by these grammars pretty easily. We won't be as formal as we just were for union; giving the intuition should be enough.

- To build a grammar G such that $L(G) = L(G_1)L(G_2)$: add a new start symbol S as above and add the single new rule $S \to S_1 S_2$
- To build a grammar G such that $L(G) = L(G_1)^*$ add a new start symbol S as above and add the new rules $S \to S_1 S$ and $S \to \lambda$

18.1 Check Your Reading. *Make proofs of correctness of those two procedures, verifying that they do indeed construct grammars for concatenation and Kleene star. Use the correctness proof for the union construction as a guide.*

We have now done all the work required to prove the following

18.2 Theorem. *Let A_1 and A_2 be context-free languages. Then*

1. *$A_1 \cup A_2$ is context-free.*
2. *$A_1 A_2$ is context-free.*
3. *$(A_1)^*$ is context-free.*

Proof. In each case the proof is: let G_1 and G_2 be context-free grammars with $L(G_1) = A_1$ and $L(G_2) = A_2$. By renaming if necessary, arrange that G_1 and G_2 have disjoint sets of variables. Use the appropriate algorithm described about to construct a new grammar generating the language desired. ///

Building Grammars Systematically

The algorithms we gave showing the set of context-free languages to be closed under union, concatenation and Kleene star are really design strategies for building complex grammars.

18.3 Example. Let's make a grammar for

$$K = \{a^i b^j c^k \mid i \neq j \text{ or } j \neq k\}$$

The strategy is: we recognize K as a union of $K_1 = \{a^i b^j c^k \mid i \neq j, k \geq 0\}$ and $K_2 = \{a^i b^j c^k \mid i \geq 0, j \neq k\}$ so that if we have grammars for each of these we can know how to combine.

Look at K_1. We can recognize that as a concatenation of $K_{11} = \{a^i b^j \mid i \neq j\}$ and $K_{12} = \{c^k \mid k \geq 0\}$ So we will build grammars for each of these, then combine them.

Now look at K_{11}. We can recognize that as a union, of the two cases K_{111} and K_{112} where $i > j$ and $i < j$.

Here is a grammar for $K_{111}, \{a^i b^i \mid i > j\}$. The variable A is there to generate non-nil strings of as.

$$\begin{aligned} S_{111} &\to aS_{111}b \mid A \\ A &\to aA \mid a \end{aligned}$$

A grammar for K_{112} is similar, using a variable B to generate non-nil strings of bs.

So then here is a grammar for K_{11}.

$$\begin{aligned} S_{11} &\to S_{111} \mid S_{112} \\ S_{111} &\to aS_{111}b \mid A \\ S_{112} &\to aS_{112}b \mid B \\ A &\to aA \mid a \\ B &\to bB \mid b \end{aligned}$$

Thus here is a grammar for K_1

$$\begin{aligned} S_1 &\to S_{11}\, S_{12} \\ S_{11} &\to S_{111} \mid S_{112} \\ S_{111} &\to aS_{111}b \mid A \\ S_{112} &\to aS_{112}b \mid B \\ A &\to aA \mid a \\ B &\to bB \mid b \\ S_{12} &\to cS_{12} \mid \lambda \end{aligned}$$

Let's call this grammar G_1. Now a grammar for K_2 is a simple modification of the above. Let's call that G_2

Finally we get our answer.

$$\begin{aligned} S \;\to\; & S_1 \;\mid\; S_2 \\ & + \text{all the rules of } G_1 \\ & + \text{all the rules of } G_2 \end{aligned}$$

Please contemplate the fact that this just what you do when you write programs: break a problem down into smaller pieces until get to a small-ish problem you have to solve by cleverness, then combine those pieces in a systematic, well-understood way.

In the problems you will apply these ideas to building context-free grammars corresponding to regular expressions.

What About Intersection and Complement?

The set of regular languages are not only closed under union, concatenation, and Kleene star, but under complement and intersection as well. We've seen how convenient it is that we can build grammars tracking union, concatenation, and Kleene star for languages, so it comes as a disappointment to learn that, in general, context-free languages do not behave so well with respect to complement and intersection.

18.4 Theorem. *The class of context-free languages are* not *closed under intersection.*

Proof. We have to use the fact that the following language is *not* context-free: $\{a^i b^i c^i \mid i \geq 0\}$. (We mentioned this language in Example 17.22; a proof that it is context-free is in Section 26.1.)

But we can exhibit two context-free languages A_1 and A_2 such that $A_1 \cap A_2$ is $\{a^i b^i c^i \mid i \geq 0\}$. Take

$$A_1 = \{a^n b^n c^m \mid n, m \geq 0\} \qquad \text{and} \qquad A_2 = \{a^m b^n c^n \mid n, m \geq 0\}$$

It very easy to write grammars for A_1 and for A_2. But clearly $A_1 \cap A_2$ is $\{a^n b^n c^n \mid n \geq 0\}$. ///

18.5 Theorem. *The class of context-free languages is* not *closed under complement.*

Proof. For any two sets X and Y,

$$X \cap Y = \overline{(\overline{X} \cup \overline{Y})}$$

So *if* the class of context-free languages were closed under complementation, *then* since we know they are closed under union, it would follow that the context-free languages were closed under intersection. This would contradict the previous result. ///

A mild version of closure under intersection *does* hold, though.

18.6 Theorem. *If G is a CFG and R is a* regular *grammar then we can construct a CFG G_R such that $L(G_R) = L(G) \cap L(R)$.*

We can't easily prove this yet: typical proofs either use pushdown automata (a machine model for *CFGs* that we will study in Section 24) or the fact that we can convert grammars to a certain nice form call Chomsky Normal Form (which we will define in Section 21. We outline a proof in Problem 216.

18.2 Regular Grammars

In this section we will introduce a certain constrained form of context-free grammar, which will give a convenient new perspective on regular languages.

18.7 Definition. A *regular grammar* is a context free grammar such that all of its productions are of the form

$$X \to aY \quad \text{or} \quad X \to \lambda$$

where $X, Y \in V$ and $a \in \Sigma$.

Of course it is allowed that $X = Y$ in the definition above.

18.8 Examples.

1. The grammar in Example 17.9 is regular.
2. Here is a grammar that generates $\{a^n b^m \mid n, m \geq 0\}$ that is, the language $a^* b^*$

$$\begin{aligned} S &\to aS \mid bB \mid \lambda \\ B &\to bB \mid \lambda \end{aligned}$$

3. Here is a grammar that generates $\{a^n b^n \mid n \geq 0\}$ that is, where the numbers of as and bs must be equal.

$$S \to aSb \mid \lambda$$

This grammar is *not* regular. And in fact we cannot replace this grammar with a regular grammar generating the same language (we will be able to prove this eventually).

4. Caution: just because a grammar is not a regular grammar, that doesn't mean that the language it generates isn't a regular language. The following grammar is not regular

$$S \to aS \mid B$$
$$B \to Bb \mid \lambda$$

but it generates the language namely a^*b^*, which of course is a regular language. (And we saw above how to generate a^*b^* by a regular grammar)

18.9 Remark. Some authors define regular grammars slightly differently, allowing an additional form of rule, namely: $A \to a$. But it is easy to see that any grammar using such rules can be transformed into an equivalent grammar which is regular according to our definition. Just do this: invent a new grammar symbol X and replace any rule of the form $A \to a$ by two rules $A \to aX$ and $X \to \lambda$.

Regular Grammars Generate Precisely the Regular Languages

So now we have yet another use of the adjective "regular:" regular grammars. You won't be surprised if I tell you that the regular grammars generate all, and only, regular languages. The proof is very easy, in fact: it turns out that regular grammars can be viewed as really nothing more than a notational variant of *NFAs*.

Building an Automaton from a Regular Grammar If we start with a regular grammar $G = (\Sigma, V, S, P)$ then we can build an automaton M in an obvious way. The states q_i of M are in one-to-one correspondence with the variables Q_i of G; the single start state s corresponds to be the start symbol S of G; and for each production $Q_i \to aQ_j$ in G we have an automaton transition $q_i \xrightarrow{a} q_j$ in M. Finally, whenever $Q \to \lambda$ is a production of G we declare the corresponding state q of M to be accepting.

18.10 Example. Let G be the following regular grammar (whose start symbol is Q_0).

$$Q_0 \to aQ_0 \mid bQ_0 \mid bQ_1$$
$$Q_1 \to aQ_2 \mid bQ_2$$
$$Q_2 \to \lambda$$

When we do the construction outlined above we get the automaton in Example 9.5.

Once we prove that the automaton thus constructed recognizes the same language that the grammar generates, we have the following.

18.11 Lemma. *For any regular grammar G there exists an NFA M such that $L(M) = L(G)$.*

Proof. Easy, the derivations in G are in obvious correspondence with the runs of M. ///

Building a Regular Grammar from an Automaton Each *NFA* yields a regular grammar in a very natural way.

We give a complete proof that the construction works, not because anything is tricky, but just for practice.

18.12 Lemma. *For any NFA M, there exists a regular grammar G such that $L(G) = L(M)$.*

Proof. Suppose $M = (\Sigma, Q, \delta, s, F)$. For each state q_i of M let us introduce a grammar-variable symbol Q_i, and let V_Q be the set of such symbols. The start symbol S of the grammar is V_s the variable corresponding to the start state of the automaton. The productions P of G are given as follows.

- whenever $q_i \xrightarrow{a} q_j$ is a transition in M, P has a production
$$Q_i \rightarrow aQ_j$$
- for each accepting state $q_i \in F$ of M, P has a production
$$Q_i \rightarrow \lambda$$

Let us prove that $L(G) = L(M)$. This is one of those cases where the trick is to prove something stronger than what is asked for.

> ***Claim.*** *For all states p and r of M and every string $x \in \Sigma^*$, $p \xrightarrow{x}\!\!\!\!\!\rightarrow\ r$ if and only if we have a derivation $P \Rightarrow^* xR$ in G.*

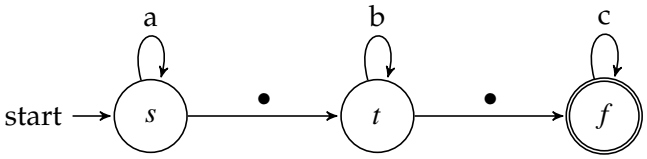
This is an easy induction over the length of x, which we won't give here.

But, once we have that we get the final result we want, by reasoning as follows.

- Suppose $x \in L(M)$. Then $s \xrightarrow{x}\!\!\!\!\rightarrow r$ where s is the start state r is some accepting state. By the claim, in G we have $S \Rightarrow^* xR$. But since r is accepting in M, we also have the production $R \to \lambda$ in G. Putting these together we have $S \Rightarrow S_i \Rightarrow^* x$, which means $x \in L(G)$.
- Conversely, if $x \in L(G)$ then $S \Rightarrow^* x$. By the form of the productions in G this can only happen if the derivation looks like $S \Rightarrow^* xR \Rightarrow x$, for some some accepting state r from M. By the claim, $s \xrightarrow{x}\!\!\!\!\rightarrow r$ in M. But this says that $x \in L(M)$.

///

What About NFA_λ? If we start with an NFA_λ, that is, an NFA with λ-transitions, then when we do the obvious thing to translate into grammar rules, we don't get a regular grammar. For example consider the NFA_λ from Example 9.18.



When we write the rules corresponding to this we naturally get to

$$\begin{aligned} S &\to aS \mid T \\ T &\to bT \mid F \\ F &\to cF \mid \lambda \end{aligned}$$

Those rules such as $S \to T$ don't fit our definition of regular rules (they are called "chain rules," by the way). As we know we can always pass to an ordinary NFA if we like. But that process is interesting from a grammar perspective, as Problem 185 explores later.

18.13 Remark. Our technique (from Section 11) for constructing regular expressions from automata started by writing down, for a given NFA or DFA, a set of equations. Notice that those equations are essentially the same thing as the CFG rules for the regular grammar we have defined in this section! The notation is slightly different (" "=" vs " $\to$ " and "+" vs "|") but the basic idea is exactly the same: capturing automata in an "algebraic" way.

See Remark 21.11 for another example of a single idea that pops up in more than one place, in mild disguise.

Regular Languages are Context-Free

The takeaway here is that regular grammars and automata are really just different notations for the same thing. We state this as a corollary to Lemmas 18.12 and 18.11.

18.14 Corollary. *Regular grammars generate precisely the regular languages.*

This also tells us that the idea of "context free language" extends the idea of "regular language."

18.15 Corollary. *Every regular language is context-free.*

We know that we don't have the converse of this claim: consider the language $\{a^n b^n \mid n \geq 0\}$, which we know to be non-regular and have shown to be context-free.

18.3 There Are Countably Many *CFLs*

How many context-free languages are there? Obviously there are infinitely many: if nothing else, any regular language is context-free. But is the number of CFLs countable or uncountable? Remember that the set of all languages over an alphabet is uncountable.

We are going to show that there are only countably many context-free languages. The strategy is pretty much *exactly* the same as the strategy we used to show that there are only countably many regular languages. Namely we will observe that

1. every context-free language can be associated with at least one grammar
2. one can encode grammars as finite strings
3. there are only countably many finite strings

The only difference between this development and the one for regular languages is that we encoded *NFAs* then, and we encode grammars now.

Since things are so similar we just give the outline here.

Step 1: Encoding grammars

Just as for *NFAs* we first agree to normalize our grammars, that is, use a standard set of symbols for the terminal and variable alphabets.

Then we just define a convention for "serializing" grammars, encoding them as strings.

This is all totally routine, and **it doesn't matter at all exactly how you do it.** All that matters is that one can translate back and forth between grammar and their string-encodings.

18.16 Check Your Reading. *Make up your own personal method for encoding grammars as strings. Verify that no two grammars are encoded as the same string; that's all that matters!*

Step 2: Concluding countability

Armed with your encoding method, we can prove the following.

18.17 Theorem. *There are only countably many context-free languages.*

Proof. For each context-free L, let G_L be the context-free grammar that generates L and whose encoding is lexicographically least among all the $CFGs$ generating L.

This defines an injective function from context-free languages to the set of finite strings over the encoding alphabet. Since the set of finite strings is countable we can conclude that the set of context-free languages is countable. ///

18.18 Corollary. *For any finite alphabet Σ, there exist languages over Σ that are not context-free.*

Proof. We proved earlier that there were uncountably many languages over Σ, so Theorem 18.17 tells us that they cannot all be context-free. ///

As with the regular languages, this is not perfectly satisfying since it doesn't give us any interesting concrete examples of non-context-free languages. We have other techniques for exhibiting specific examples of non-context-freeness. See Section 26

18.4 Problems

182. FAToCFG

Choose 3 examples of *DFAs* or *NFAs* from previous sections, and construct the corresponding regular grammars.

183. LinearGrammar

Regular grammars as we have defined them are sometimes called "strongly right-linear grammars." A "strongly left-linear grammar" is one for which all of its productions are of the form

$$X \to Ya \quad \text{or} \quad X \to \lambda$$

Show that strongly left-linear grammars and strongly right-linear grammars generate the same class of languages.

184. LRLinear

Inspired by Problem 183, let's define a new kind of grammar, called a "left-right-linear" grammar, in which every production is one of the forms

$$X \to aY \quad \text{or} \quad X \to Ya \quad \text{or} \quad X \to \lambda$$

We might then conjecture that "left-right-linear" grammar generate precisely the regular languages.

But no: this is false. Prove it.

Hint. Remember that the following language A is not regular: $A = \{a^n b^n \mid n \geq 0\}$. If you show that this A can be generated by a left-right linear grammar, you will have proven the conjecture false.

185. ElimChain

Recall the discussion about NFA_λ in Section 18.2: we noticed that NFA_λ lead to grammars that are not regular when we naively translate them: they have those "chain rules" that look like $X \to Y$.

Let's explore that here. For a concrete example, look at the NFA_λ M from Example 9.18. A grammar corresponding to this machine was given in Section 18.2.

a) Using Lemma 21.7, make an equivalent grammar that doesn't have the $S \to T$ rule. (Your new grammar will have more rules; it will even have a new chain rule.)

b) Iterate this process to get an equivalent grammar without any chain rules. (Just grind it out, eventually the chain rules will disappear)

c) Make the *NFA* that corresponds to your final grammar. Compare it to the *NFA* built in Example 9.18. What do you notice?

d) Go through this same process for the NFA_λ in Example 9.17.

Articulate a general conclusion.

186. MakeMoreCFGs

For each of the following languages K, construct a context-free grammar G such that $L(G) = K$. Take the lessons of Section 18.1 to heart.

Explain your construction! As an example, we've done the first one for you.

a) $a^n b^m c^n$ with n, m any natural numbers

Solution:

$$\begin{aligned} S &\rightarrow aXc \mid \lambda \\ X &\rightarrow bX \mid \lambda \end{aligned}$$

Explanation:

The variable S clearly generates any string that looks like $a^n X c^n$.

The variable X clearly generates any string that looks like b^m

Putting these together gives us what we want.

b) $a^n b^m c^m d^{2n}$ with n, m any natural numbers

c) $a^n b^m$ with $0 \leq n \leq m \leq 2n$

d) $a^m b^n c^k$ where (m=n) or (m=k)

e) $\{a^i b^i c^k d^k \mid i,k \geq 0\}$

f) $\{a^i b^j c^k d^m \mid i,j,k,m \geq 0, and (i = j \text{ or } k = m)\}$

187. CFGReverse

Give a construction that demonstrates that if A is context-free then the set A^R of reversals of strings in A is context-free.

Your proof should start like this:

Let $G = (\Sigma, V, S, P)$ be a CFG such that $L(G) = A$. We construct a CFG G' such that $L(G') = A^R$.

188. NotDouble

Consider the language L over $\Sigma = \{a,b\}$: $L = \{x \mid x \text{ is } not \text{ of the form } ww\}$. Show that L is context-free.

Hint. This is not easy. The following exercises gradually build up to a solution of that problem.

1. First note that a string is in $\{x \mid x \text{ is not of the form } ww\}$ precisely if
   - $|x|$ is odd, OR
   - x looks like
     $$y_1 \cdots y_{i-1} a y_{i+1} \cdots y_m z_1 \cdots z_{i-1} b z_{i+1} \cdots z_m$$
     OR
   - x looks like
     $$y_1 \cdots y_{i-1} b y_{i+1} \cdots y_m z_1 \cdots z_{i-1} a z_{i+1} \cdots z_m$$
2. Write a CFG for $X = \{uav \mid u,v \in \{a,b\}^*, |u| = |v|\}$.
3. Observe that a CFG for $Y = \{ubv \mid u,v \in \{a,b\}^*, |u| = |v|\}$ is a trivial variation on the one for X.
4. Describe the languages XY and for YX. Observe that it is now easy to write a CFG for the concatenations XY and for YX.
5. Put the pieces together.

189. CFLRegClosure

Suppose L_1 and L_2 are context-free languages and suppose that R is a regular language. For each of the following languages, say whether it is guaranteed to be context-free. If your answer is "yes" prove it; If your answer is "no" give specific languages which comprise a counterexample. (Recall that $A - B$ abbreviates $A \cap \overline{B}$.)

1. $L_1 - R$
2. $R - L_1$
3. $L_1 - L_2$

For this problem you may make use of the result in Problem 243.

Hint. Remember the intersection of a regular language and a context-free language is guaranteed to be context-free.

190. UnionNonCFG

Let R be a regular language and let N be a language which is *not* context-free. Let $X = R \cup N$.

1. Show by means of examples that we cannot conclude whether X is context-free or not.
2. Prove that if we have the further condition that $R \cap N = \emptyset$ then X is guaranteed to be not context-free.

Chapter 19

Proving correctness of grammars

Here we explore how one proves that a certain CFG generates a certain language.

19.1 General Strategy

In order to prove that CFG G generates language L, you must do two things:

1. Prove that $L(G) \subseteq L$.

   To do this you typically use induction on the length of a derivation in G.

2. Prove that $L \subseteq L(G)$.

   To do this you typically use induction on the length of a string in L.

Often the proof of (2) is significantly harder than the proof of (1). Sometimes what you have to do is find some way to characterize the strings in L that helps you put the kind of "structure" on them that the grammar induces.

19.2 Examples

19.1 Example. Let E be the set of strings over $\{a, b\}$ with an equal number of as and bs.

Let G be the following CFG:

$$S \to \lambda \mid SS \mid aSb \mid bSa$$

We wish to prove that $L(G)$ is E.

A solution

We will:

- first show $L(G)$ is a subset of E,
- then show E is a subset of $L(G)$.

Proof that $L(G)$ is a subset of E: We claim that for all w in $L(G)$, w is in E.

Since w is in $L(G)$, there is a derivation of w.

We proceed by induction on the length of this derivation. So note that the induction hypothesis is that: *for any string u in $L(G)$ which has a shorter derivation, u is in E*

If the length of the derivation of w is 1, then w must be the null string λ, which is obviously in E.

Otherwise the derivation has one of 3 possible forms:

1. $S \Rightarrow aSb \Rightarrow^* w$ or
2. $S \Rightarrow bSa \Rightarrow^* w$ or
3. $S \Rightarrow SS \Rightarrow^* w$

In the first case w looks like $aw'b$ and there is a derivation of w' from S. This derivation is shorter than the total derivation of w, so it satisfies the induction hypothesis, so w' is in E. Clearly, then, w itself is in E.

The second case is similar.

In the last case w looks like $w'w''$ and there is a derivation of w' from (the first) S and a derivation of w'' from (the second) S. Each of these derivations is shorter than the total derivation of w, so each satisfies the induction hypothesis, so both w' and w'' are in E. Clearly, then, w itself is in E.

Since we have addressed all cases, this completes the proof of (1), that $L(G)$ is a subset of E.

Proof that E is a subset of $L(G)$: We show that for every string w in E, w is derivable in the grammar. We proceed by induction on the length of w.

There are three possibilities for the form of w:

1. w is the null string λ, or
2. w is of the form au, or
3. w is of the form bu.

(1) When w is the null string λ (which is indeed in E), obviously w is in $L(G)$ by the simple derivation $S \Rightarrow e$.

(2) When w is au:

Define the following function on initial substrings x of w [an initial substring of w is simply "the first k symbols of w" for some k]:

$$d(x) = (\#(a) \text{ in } x) - (\#(b) \text{ in } x)$$

Note that $d(w) = 0$, but for other x, $d(x)$ may fluctuate positive and negative. Now let x be the first non-null initial substring of w where $d(x) = 0$. It may be that (i) x is w itself, or it may be that (ii) the first such x occurs before the end of w. We examine these two cases.

(i) If x is w, this means that w must look like $aw'b$. Furthermore the string w' has $\#(a) = \#(b)$, and so is in E. Since w' is shorter than w, the induction hypothesis applies so w' is in $L(G)$. This gives us a derivation of w itself, as follows:

$$S \Rightarrow aSb \Rightarrow^* aw'b$$

(The $aSb \Rightarrow^* aw'b$ part above is justified by the fact that w' has a derivation from S)

(ii) On the other hand, if x is not all of w, we may let y be the part of w following x, and write w as xy. Note that neither x nor y is null, and so therefore each of x and y is shorter than w. Furthermore, since $d(x) = 0$, we must have x in E. And since w itself is in E, we can further conclude that y is in E. Now the induction hypothesis applies to both x and y, so the are each derivable. This means that there is a derivation of w itself, namely:

$$S \Rightarrow SS \Rightarrow^* xS \Rightarrow^* xy$$

as desired.

(3) The final thing to consider is when w starts with a b, which is to say that w is bu. But this is completely symmetric with the case when w starts with an a, so we needn't repeat the details. Note that the function $d(x)$ now starts out with a negative value, just because the initial substring is "b". But we would still look at the first x where $d(x) = 0$ and proceed accordingly.

This completes the proof of (2), that E is a subset of $L(G)$.

19.2 Example. The grammar in Example 19.1 was simple in the following respect: there was only the one variable S.

When—as is typical—a grammar has more than one variable, reasoning about it involves reasoning not only about the strings that S generates but also about the

strings that other variables generate, since these interact with each other. For instance consider this G:

$$\begin{aligned} S &\to cSc \mid A \\ A &\to aAb \mid \lambda \end{aligned}$$

Suppose we want to prove that $L(G)$ is $\{c^k a^n b^n c^k \mid k \geq 1, n \geq 0\}$?

To do so we need to prove *both*

1. $S \Rightarrow^* w$ if and only if $w \in \{c^k a^n b^n c^k \mid k \geq 1, n \geq 0\}$
2. $A \Rightarrow^* w$ if and only if $w \in \{a^n b^n \mid n \geq 0\}$.

Proving the second of these is straightforward (it uses the techniques the previous example, but it is easier!) so let's assume that has been done. Once we know that about the strings derivable from A we can complete the proof.

Let K denote $\{c^k a^n b^n c^k \mid k \geq 1, n \geq 0\}$. As before, we first show that $L(G) \subseteq K$. Let $w \in L(G)$. We show that $w \in K$ by induction on the length n of the shortest derivation witnessing $w \in L(G)$. If the first step of the derivation is $S \Rightarrow cSc$ then we know that w is of the form $cw'c$ where $S \Rightarrow w'$ by a derivation of length $(n-1)$. By induction, then, x' is in K. So $w = cw'c$ is clearly in K as well.

Conversely suppose $w \in K$; we wish to show $w \in L(G)$. We do this by induction on the length of w. There is a little bit of cleverness involved: we do not decompose w by naively peeling off its first element. Rather we (intuitively) peel off the first and last elements occurrences of c in w, as follows.

Case 1. If w is of the form $a^n b^n$, that is, if there are no c, then since we proved that A generates all such w, we have the following derivation of w in G: $S \Rightarrow A \Rightarrow^* w$.

Case 2. Here w is of the form $c^k a^n b^n c^k$ with $k \neq 0$. Thus w can be written as $cw'c$ where w' is in K. Since w' is shorter that w we know there is a derivation $S \Rightarrow^* w'$. Thus we can derive w, by $S \Rightarrow cSc \Rightarrow^* cw'c = w$.

19.3 Example. Even though the grammar in Example 19.2 had more than one variable, it was still a simple case in the following sense: the two variables didn't *interact.* That is, we were able to reason about variable A on its own, then import the result into reasoning about S. This worked because derivations starting with A never involve any other variable.

Contrast that situation with this one.

$$\begin{aligned} S &\to bS \mid aA \mid \lambda \\ A &\to aS \mid bA \end{aligned}$$

This grammar generates the set Ev_b of strings with an even number of bs. But we cannot prove this by reasoning independently about the terminal strings derivable from S and A independently.

Instead we use the techniques of *simultaneous induction.* We prove that

for all n,

1. if $S \Rightarrow^* w$ in n steps or fewer, then w has an even number of as, and
2. if $A \Rightarrow^* w$ in n steps or fewer, then w has an odd number of as

simultaneously by induction on n. When we have done this we know that $L(G) \subseteq Ev_b$ by invoking the first assertion.

We then prove that

for all words w,

1. if w has an even number of as then $S \Rightarrow^* w$, and
2. if w has an odd number of as then $A \Rightarrow^* w$

simultaneously by induction on the length of w. When we have done this we know that $Ev_b \subseteq L(G)$ by invoking the first assertion.

[Details of this argument are omitted for now . . .]

19.3 Problems

191. PrvAll

Prove that the following grammar generates all strings over $\{a, b\}$.

$$S \rightarrow aS \mid Sb \mid bSa \mid \lambda$$

192. PrvPrefix

Let G be the following grammar.

$$S \rightarrow aS \mid aSbS \mid \lambda$$

Prove that $L(G)$ is the set of strings w over $\{a, b\}$ such that every prefix of w has at least as many as as bs.

193. PrvEvenPal

For this problem: let G be the grammar

$$S \rightarrow aSa \mid bSb \mid \lambda$$

Prove that $L(G)$ is the set of even-length palindromes over $\{a, b\}$. (A palindrome is a string which is equal to its own reversal.)

Hint: Let E denote the set of even-length palindromes over $\{a, b\}$. As suggested above, you need to:

1. Prove that $L(G) \subseteq E$, that is, every string generated by the grammar is a palindrome. (Induct on the length of the derivation.)
2. Prove that $E \subseteq L(G)$, that is, every palindrome is generated by the grammar. (Induct on the length of the palindrome.)

194. PrvMystery

Consider the following grammar G.

$$\begin{aligned} S &\rightarrow aB \mid bA \\ A &\rightarrow a \mid aS \mid bAA \\ B &\rightarrow b \mid bS \mid aBB \end{aligned}$$

1. Spend 5 minutes trying to decide what language G generates (try not to read the next part of this question!)
2. Now suppose you are *told* that $A \Rightarrow^* w$ if and only if w has exactly one more a than it has bs, and that $B \Rightarrow^* w$ if and only if w has exactly one more b than it has as. Can you see what language G generates now? (All you need to do is look at the two S-rules!)
3. Prove that $L(G)$ is what you decided. *Hint.* the thing to do is prove, *simultaneously* that
   (a) $A \Rightarrow^* w$ if and only if w has exactly one more a than it has bs,
   (b) $B \Rightarrow^* w$ if and only if w has exactly one more b than it has as, and
   (c) $S \Rightarrow^* w$ if and only if ...

195. PrvDiff

Compare these two grammars:

G_1 :

$$\begin{aligned} S &\to ASB \\ A &\to a \\ B &\to bb \end{aligned}$$

G_2 :

$$\begin{aligned} S &\to AB \\ A &\to aA \mid a \\ B &\to bbB \mid bb \end{aligned}$$

It is instructive to see how G_1 and G_2 differ ...once you understand the difference, define what $L(G_1)$ and $L(G_2)$ are, respectively, and prove your answers.

Hint. These are again situations where you will want to strengthen your induction hypotheses to include assertions about $A \Rightarrow^* \ldots$ and $B \Rightarrow^* \ldots$ as well as $S \Rightarrow^* \ldots$

196. PrvParens

Let Σ be the alphabet $\{a, b\}$. But think of these as standing for the left and right parentheses symbols, respectively (it's kind of confusing to try to read long strings of parentheses). If u is a string, let us use the notation $\#a(u)$ to stand for the number of occurrences of the symbol a in u, and of course $\#b(u)$ is the number of occurrences of b in u.

Consider the language *Bal* over Σ defined as follows: a string w is in *Bal* if

- $\#a(w) = \#b(w)$, and
- for every initial substring u of w, $\#a(u) \geq \#b(u)$.

Another way to express the conditions above is to define, for any string u, the quantity $\#a(u) - \#b(u)$, and say that as u ranges over larger and larger initial substrings of w, this quantity never goes below 0 and must equal 0 when u reaches w itself.

Convince yourself that *Bal* is the language which we intuitively describe as strings of "balanced" parentheses.

Now let G be the following grammar.

$$S \to SS \mid aSb \mid \lambda$$

Prove that G generates the set *Bal* of "balanced parentheses".

197. PrvEqual

Let E be the set of strings over $\{a, b\}$ with an equal number of as and bs.

Let G be the following grammar.

$$S \to aSbS \mid bSaS \mid \lambda$$

Prove that $L(G)$ is E.

Yes, this is the same language as in the opening example. Very different grammars can define the same language...

Hint: As in problem 196 consider the quantity $\#a(u) - \#b(u)$ as u ranges over prefixes of a string w. Symbolize the strings in the language of equal numbers of as and bs in terms of this function. Then you can proceed as in problem 196.

Chapter 20

Ambiguity

20.1 Defining Ambiguity

It can sometimes happen that a string w can be generated by a grammar G with two different parse trees.

20.1 Example. Let G be the grammar

$$S \to aS \mid Sa \mid \lambda$$

This grammar generates all strings a^n, for $n \geq 0$. But parse trees are not unique. Indeed, there are two parse trees for deriving the single string a!

Here is a more interesting—in a precise sense that we will define soon—example.

20.2 Example.

$$\begin{aligned} S &\to S_1 \mid S_2 \\ S_1 &\to AB \\ A &\to aAb \mid \lambda \\ B &\to cBd \mid \lambda \\ S_2 &\to aS_2d \mid D \\ D &\to bDc \mid \lambda \end{aligned}$$

The grammar generates

$$K = \{a^n b^n c^m d^m \mid m,n \geq 1\} \cup \{a^n b^m c^m d^n \mid m,n \geq 1\}$$

20.3 Check Your Reading. *Build two derivation trees for the string aabbccdd.*

These examples motivate the following definition.

20.4 Definition. A context-free grammar G is *unambiguous* if for every $w \in L(G)$ there is exactly one parse tree for w.

So, a context-free grammar is *ambiguous* if there is at least one string w such that there is more than one parse tree yielding w in G.

In light of our earlier observation that parse trees can be unique associated with leftmost derivations, one can describe ambiguity in terms of derivations, if one insists. Namely, a grammar is *unambiguous* if every derivable string w has exactly one leftmost derivation; so G is *ambiguous* if there is at least one string w such that there is more than leftmost derivation of w in G.

So why do we care of a grammar is ambiguous or not? The answer is that parse trees give structure to derivable strings, and in an applied setting this structure is typically part of giving semantic *meaning* to strings.

20.5 Example. The following grammar generates arithmetic expressions over a terminal alphabet consisting of $+$, $*$, and numerals. To avoid distractions we will just use single-digit numerals as basic expressions. It is traditional to use E as the start symbol for such expression-grammars.

$$E \to E+E \mid E*E \mid 0 \mid 1 \mid 2 \mid \ldots \mid 9$$

Consider these three derivations

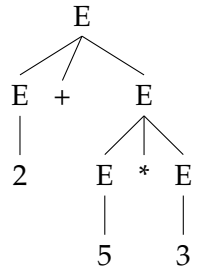
$$E \Longrightarrow E+E \Longrightarrow E+E*E \Longrightarrow 2+E*E \to 2+5*E \to 2+5*3$$

$$E \Longrightarrow E+E \Longrightarrow E+E*E \Longrightarrow E+5*E \to E+5*3 \to 2+5*3$$

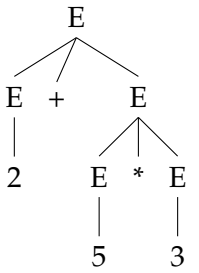
$$E \Longrightarrow E*E \Longrightarrow E+E*E \Longrightarrow 2+E*E \to 2+5*E \to 2+5*3$$

The three derivations yield the same terminal string. But one of these three is different from the other two in an important way. The easiest way to see the difference is to look at the respective parse trees.

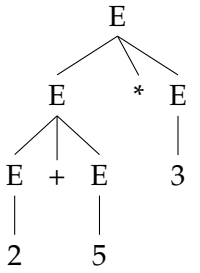
First derivation:



Second derivation:



Third derivation:



The first second and derivations have *the same parse tree,* and the third one is different.

Recalling our earlier discussion about parse trees being the carriers of meaning for expressions, we see that the difference between the first two derivations is a superficial one, the difference between the first and third (and between the second and third) is a significant difference.

The difference between the parse trees would be manifest in the difference in *evaluating* these two trees: clearly for the tree with + near the top of the tree we would get the answer 2 + (5*3) = 17, while with * near the top of the tree we would get the answer (2 + 5) * 3 = 21.

20.2 Removing Ambiguity From Grammars

Ambiguous grammars cause problems in applications (such as compiling programming languages). In this section we see a few tricks for eliminating ambiguity. But note two things:

- It is not always possible to eliminate ambiguity from a grammar (see the next section)
- Simply finding an unambiguous grammar is not usually good enough: usually one wants to find an unambiguous grammar that generates the "right" parse trees based on the intended semantics of the language. This will be clearer when we see examples.

Ad-hoc Techniques

If you are given an ambiguous grammar G and are asked to find an unambiguous generating the same language, in general there are no magic techniques to use.

Recall Example 20.1. An unambiguous grammar deriving the same strings as the grammar G there is

$$S \to aS \mid \lambda$$

Assigning problems to remove ambiguity from a language is a favorite pastime of foundations instructors the world over. But in general these problems just test your cleverness as opposed to imparting any insight or useful general principles. So we will try not to overdo this ....

On the other hand, there are two typical and important kinds of ambiguity that arise in practice: *precedence* ambiguity and *grouping* ambiguity. Understanding how to address these kinds of ambiguity will deepen your understanding of how grammars work and can guide you as you design grammars yourself. So let's explore these.

Precedence Ambiguity

The ambiguity that we saw at the beginning of the section had to do with operator precedence. In our 2+5*3 example, there was a choice as to whether the + or the * was higher up in the parse tree. Once we decide which operate we want to have higher precedence, we can then engineer the grammar to enforce that. With the familiar + and * it is convention for * to have higher precedence, so let's do that.

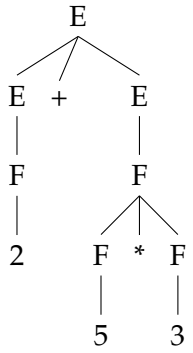
By the way, note that when people speak of "higher precedence" what they are saying is "lower in the parse tree." Don't get confused by that. (Lower in the parse tree means "happens earlier in the evaluation process", hence "precedes", hence "precedence.")

The trick to enforcing precedence is to enforce "layers" in the grammar, which typically involves introducing new variables.

Let's work on the original expression grammar, but we introduce a variable F (for "factor").

$$E \rightarrow E + E \mid F$$
$$F \rightarrow F * F \mid 0 \mid 1 \mid 2 \mid \ldots \mid 9$$

Now we can still derive 2+5 * 3, but in only one way:



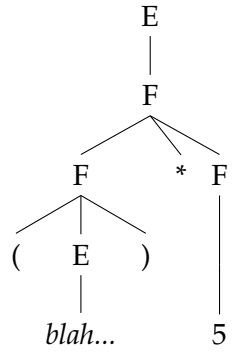
And the natural way to evaluate this will do the multiplication first, and get the answer 17.

Overriding Precedence

Of course we want to provide our users with the possibility of overriding the precedence our grammar has built in. For example the user might really want to write an expression with the meaning "2 plus 5, then multiply by 3". The only way to provide that possibility is to allow parentheses in the language, and write the grammar so that a parenthesized expression is parsed appropriately. Luckily that isn't hard. Here is another enhancement of our expresion grammar, in which **we have added left and right parentheses to the terminal alphabet Σ.**

$$E \to E+E \mid F$$
$$F \to F*F \mid (E) \mid 0 \mid 1 \mid 2 \mid \ldots \mid 9$$

20.6 Example. Now the string (2 + 3) * 5 is in our language, and it is parsed as we would expect:



where *blah...* is a parse tree for 2+3, not shown, so as not to clutter the picture.

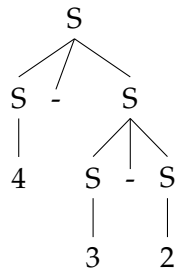
Grouping Ambiguity

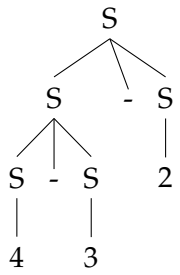
A different kind of ambiguity can arise with a single operator.

20.7 Example. Consider this little expression grammar, with just the single subtraction operator:

$$S \to S-S \mid 0 \mid \ldots \mid 9$$

This is ambiguous, since 4 - 3 - 2 can be derived in two ways

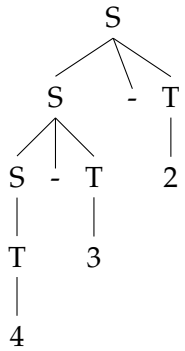




The natural way to evaluate these would yield 3 and -1 respectively. Assume we would like to make a grammar that enforced *left* associativity for -, that is, preferring the answer -1. How do we do it? The trick is to make sure that the symbol S can only be recursive "on the left." To do that we introduce another variable, T, to help out.

$$S \to S - T \mid T$$
$$T \to 0 \mid \ldots \mid 9$$

Now we can still derive 4-3-2, but in only one way:



20.8 Check Your Reading. *Make a grammar for exponentiation (use any symbol you like , maybe an up-arrow ↑) but enforce* right *associativity.*

20.9 Example. Let's put the two ideas together, to get an expression grammar that eliminates precedence ambiguity and grouping ambiguity. Let's agree that multiplication has higher precedence than addition, and the addition should associate to the left, while multiplication should associate to the right. (There no reason for left vs right choice except for the sake of making a better example)

$$E \to E + T \mid T$$
$$T \to I * T \mid I$$
$$I \to 0 \mid 1 \mid 2 \mid \ldots \mid 9$$

20.3 Inherent Ambiguity

Suppose we are given a grammar G that we know to be ambiguous. Can we always construct an unambiguous equivalent G'?

The answer is no. There exist grammars G that are ambiguous but such that there are no unambiguous grammars generating the same language.

One instance: the language

$$K = \{a^n b^n c^m d^m \mid m,n \geq 1\} \cup \{a^n b^m c^m d^n \mid m,n \geq 1\}$$

defined by the grammar in Example 20.2.

It is complicated to prove that the language K can have no unambiguous grammar generating it, so we won't prove it here.

But you should be able to make yourself believe it, if you think about it for a little while...

20.10 Definition. A context-free language K is *inherently ambiguous* if every context-free grammar generating K is an ambiguous grammar.

Please note the distinction between Definition 20.10 and Definition 20.4. One gives a property of *grammars*, the other gives a property of *languages.*

20.4 A Decision Problem: Deciding Ambiguity

Suppose we are given a grammar G. Can we decide algorithmically whether or not G is ambiguous?

The answer is no. Certainly if we find two parse trees that yield the same string we can say that G is ambiguous, but if we are searching for such a pair of trees, that search is—at least naively—an infinite search. Is there a stopping condition that can tell us when we have searched enough? The result is that, no, no such finite search method can exist. This is pretty hard to prove, and we postpone it until Chapter 39.

20.5 Problems

198. BasicAmb

For each grammar, decide whether it is ambiguous or not. If it is, prove it by exhibiting a string with two different parse trees, or two different leftmost derivations. Then find an unambiguous grammar generating the same set of strings.

Hint. Don't feel constrained to apply our systematic methods for addressing precedence and grouping ambiguity. These problems are little self-contained puzzles that don't necessarily fit those patterns.

Remember that it is not true that **every** ambiguous grammar can be replaced by an unambiguous grammar generating the same strings. But in this problem you may be confident that the ambiguous grammars below do have unambiguous counterparts.

(Most of these grammars are from [Sud97])

a)

$$S \to aS \mid Sb \mid ab$$

b)

$$\begin{aligned} S &\to aA \mid \lambda \\ A &\to aA \mid bB \\ B &\to bB \mid \lambda \end{aligned}$$

c)

$$\begin{aligned} S &\to AaSbB \mid \lambda \\ A &\to aA \mid a \\ B &\to bB \mid \lambda \end{aligned}$$

d)

$$S \to aaS \mid aaaS \mid \lambda$$

e)

$$S \to aSb \mid aSbb \mid b$$

f)

$$\begin{aligned} S &\rightarrow A \mid B \\ A &\rightarrow abA \mid \lambda \\ B &\rightarrow aBb \mid \lambda \end{aligned}$$

199. ArithAmb1

We looked at expression-grammars in a fragmentary way in the chapter; let's do an end-to-end example.

Let G be the following grammar.

$$\begin{aligned} E &\rightarrow E+E \mid E*E \mid I \\ I &\rightarrow a \mid b \mid c \end{aligned}$$

Let G^* be the following grammar.

$$\begin{aligned} E &\rightarrow T+E \mid T \\ T &\rightarrow I*T \mid I \\ I &\rightarrow a \mid b \mid c \end{aligned}$$

a) The grammar G is ambiguous not only because the precedence of + and * is not determined, but also because the associativity of the individual operators + and * is not determined. The grammar G^* is unambiguous, and ensures that + and * are parsed as right-associative operators.

Change the grammar of Figure G^* so that it is still unambiguous and generates the same strings with the same precedence, but with + and * parsed as left-associative.

b) Suppose we add a new terminal "$-$" to the language, intended to denote subtraction. Suppose that we enrich the grammar G^* to include subtraction as a binary operator, so that if E_1 and E_2 are expressions, then so is $E_1 - E_2$.

Extend the grammar of the previous part to get an unambiguous grammar generating arithmetic expressions including subtraction. Your grammar should make subtraction left-associative and at the same level of precedence as +.

Note that the arithmetic operation of subtraction is not associative. Repeated subtraction conventionally groups to the left. For example, the conventional value of $5-3-1$ is 1 (not 3). Be sure your grammar induces the correct grouping!

c) Suppose we add a new terminal $\uparrow$ to the language, intended to denote exponentiation. That is, if U and W are expressions, then $U \uparrow W$ is to be an expression, intuitively denoting "U raised to the power W".

Extend the grammar of the previous part to get an unambiguous grammar generating arithmetic expressions including exponentiation. Your grammar should induce the usual rules of precedence involving exponentiation, that is, that $\uparrow$ binds more tightly than any other operator.

Note that the arithmetic operation of exponentiation is not associative. Repeated exponentiation conventionally groups to the right. For example, the conventional value of $2 \uparrow 3 \uparrow 2$ is 2^9. Be sure your grammar induces the correct grouping!

d) Extend the grammar of the previous part to allow the comparison operators = and $<$. They should all be left-associative and at the same level of precedence, below that of +. (That is, an expression like $a+b=c$ should have a parse tree with the = appearing higher than the $+$.)

200. ArithAmb2

a) Consider this grammar, G_1.

$$E \rightarrow I+E \mid I-E \mid I$$
$$I \rightarrow a \mid b \mid c$$

We have designed this so that $+$ and $-$ are at the same level of precedence and are free individually from grouping ambiguity by grouping to the right.

Is G_1 ambiguous? If so, prove it (by building two parse trees yielding the same string). If no, explain why strings have unique parse trees.

b) Consider this grammar, G_2.

$$E \rightarrow I+E \mid E-I \mid I$$
$$I \rightarrow a \mid b \mid c$$

We have designed this so that $+$ and $-$ are at the same level of precedence and are free individually from grouping ambiguity (though they group opposite ways).

Is G_2 ambiguous? If so, prove it (by building two parse trees yielding the same string). If no, explain why strings have unique parse trees.

201. PrefixAmb

The following grammar generates *prefix* arithmetic expressions over $+$, $*$, and $-$.

$$E \rightarrow +EE \mid *EE \mid -EE \mid I$$
$$I \rightarrow a \mid b \mid c$$

Build a parse tree for the term $+*-abab$.

Is this grammar ambiguous? Explain why or why not.

202. BoolAmb

Here is a grammar G generating formulas of propositional logic over the alphabet $\Sigma = \{\wedge, \vee, \equiv, \neg, (,), p, q, r, s, \ldots\}$

$$B \to B \wedge B \mid B \vee B \mid B \equiv B \mid \neg B \mid (B) \mid I$$
$$I \to p \mid q \mid r \mid s \mid \ldots$$

This grammar is ambiguous.

Write an unambiguous grammar G' generating the same strings, enforcing the rules that

- the binary operators are left-associative,
- $\neg$ has highest precedence, followed by $\wedge$, then $\vee$, then $\equiv$ (but parentheses can override these precedences).

For example, the parse tree for $p \equiv \neg q \wedge q \vee r$ should have $\equiv$ as the operator at the top of the tree, with $\vee$ at the next level, $\wedge$ at the level below that, and $\neg$ applied to the symbol q alone.

203. PLAmb

Consider the following grammar over the alphabet

$$\Sigma = \{\texttt{if}, \texttt{then}, \texttt{else}, \texttt{statement}, \texttt{condition}\}$$

$$S \to \texttt{if}\ C\ \texttt{then}\ S$$
$$S \to \texttt{if}\ C\ \texttt{then}\ S\ \texttt{else}\ S$$
$$S \to \texttt{statement}$$
$$C \to \texttt{condition}$$

If we interpret S as a variable standing for statements in a programming language and we interpret C as a variable standing for boolean conditions, then the above grammar generates "if-then" and "if-then-else" statements. (Of course in a grammar specifying more details we would expand `statement` and `condition`...)

1. Show that this grammar is ambiguous. This requires exhibiting two parse trees!
2. This ambiguity is known as the "dangling else" problem; explain what this ambiguity means in terms in the meaning of real programming language statements.

   The way to do this is to show a single (simple!) program statement, such that if it is parsed one way it does one thing and if it is parsed the other way it does a different thing.

3. The convention in most programming languages is that a statement which is ambiguous in the sense that you discovered above should be interpreted using the following rule: *an* `else` *is always paired with the closest preceding* `if` *if that doesn't already have an* `else` *paired with it.* Give an unambiguous grammar which enforces this rule.

204. UnAmb

We noted that the problem of testing a grammar for ambiguity is undecidable. It is useful in applications to find *sufficient* conditions for a grammar to be unambiguous.

a) Suppose G is a regular grammar derived from a *DFA* (see Lemma 18.12). Explain carefully why G is unambiguous.

Given an example to show that this is not true for *NFAs*.

b) If G is a grammar and A is a variable of G, let us call any rule

$$A \to \alpha$$

an "A-rule".

If α is not λ then the first symbol of α is either in Σ or in V.

What part a) really showed is that if G is regular and no two A-rules start with the same Σ-symbol, then G is unambiguous.

Can we generalize this? Suppose $G = (\Sigma, V, S, P)$ is a grammar with the following property:

> for every variable A, no two A-rules start with the same (Σ or V) symbol.

Explain carefully why such a G is unambiguous, or give a counterexample.

205. InherentAmb

a) Be able to explain the difference between *G is an ambiguous context-free grammar* and *L is an inherently ambiguous context-free language*.

b) Does it make sense to talk about a language being ambiguous?

c) Does it make sense to talk about a grammar being inherently ambiguous?

d) Does it make sense to talk about a parse tree being ambiguous?

206. RegAmb

Prove that no regular language is inherently ambiguous.

207. NLAmb

(A very optional problem) Lots of natural language sentences are ambiguous. Take your favorite ambiguous sentence or phrase in your favorite natural language, and show how the ambiguity manifests itself in terms of parse trees.

For inspiration about modeling natural language utterances using context-free grammars, recall Example 17.27.

Chapter 21

Refactoring Context-Free Grammars

Sometimes we would like to change a grammar into another grammar that generates the same language but has nicer properties. Think of this as "refactoring" grammars.

21.1 Overview

This chapter is a collection of small results, but there are a couple of recurring ideas/techniques which are robust and important. The specific class you are taking at this moment might omit some of the material, but presumably will cover enough for you to absorb the lessons of those core ideas.

To make it easier to proceed selectively we collect all the basic definitions here at the beginning.

21.1.1 A Zoo of Definitions

Reachable Variables

Only those grammar variables that can be reached from the start symbol can be used in deriving strings.

21.1 Definition. Let $G = \langle \Sigma, V, P, S \rangle$ be a *CFG*. A variable A of G is *reachable* if there is some G-derivation $S \Rightarrow^* \alpha A \beta$. Here α and β are in $(\Sigma \cup V)^*$.

Generating Variables

Only those grammar variables that can themselves lead to terminal strings can play a role in derivations of terminal strings.

21.2 Definition. Let $G = \langle \Sigma, V, P, S \rangle$ be a CFG. A variable A of G is *generating* if there is some G-derivation $A \Rightarrow^* w$ with $w \in \Sigma^*$.

Nullable Variables

It turns out to be useful to know which variables can (possibly) be erased in a derivation.

21.3 Definition. Let $G = \langle \Sigma, V, P, S \rangle$ be a CFG. A variable A of G *nullable* if there is a derivation $A \Longrightarrow^* \lambda$ in G.

Note that the notion of nullable variable captures a more general concept than rules of the form $A \to \lambda$. For example, if our grammar had rules $A \to BC$, $B \to \lambda$ and $C \to \lambda$ then A would be nullable even though it is not the direct subject of any erasing rules.

Chain Rules

21.4 Definition (Chain Rule)**.** A *chain rule* in a grammar is one of the form $A \to B$, where B is a variable.

Erasing Rules

21.5 Definition (Erasing Rule)**.** An *erasing rule* in a grammar is one of the form $A \to \lambda$.

21.1.2 Menu

In this chapter we will discuss the following refactorings:

1. Given a grammar G, construct a grammar G' generating the same language, such that every variable of G' is both reachable and generating.

2. Given a grammar G, construct a grammar G' generating the same language, such that G' has no chain rules and (essentially) no erasing rules.

   That weasely parenthetical "essentially" will be explained later.

3. Given a grammar G, construct a grammar G' with $L(G') = L(G)$ such that G' is in a certain nice form known as Chomsky Normal Form.

4. Given a grammar G, construct a grammar G' with $L(G') = L(G)$ such that G' is in a certain nice form known as Greibach Normal Form.

Your class might not cover this entire chapter, so it is divided into several small modules, that can be used *a la carte*.

21.2 Adding and Removing Rules

Here are two little things that get used in a couple of places later.

The first observation is that if a variable A can derive something in several steps then there is no harm in just capturing that as a rule.

21.6 Lemma (Adding Rules)**.** *Suppose $G = (\Sigma, V, S, P)$ is a CFG and $A \stackrel{*}{\Longrightarrow} \alpha$ is a derivation in G. If we build G' from G by adding $A \to \alpha$ as a rule:*

$$G' = (\Sigma, V, S, P \cup \{A \to \alpha\})$$

then $L(G') = L(G)$.

Proof. The fact that $L(G) \subseteq L(G')$ follows from the fact that any rule of G is still a rule of G'. The fact that $L(G') \subseteq L(G)$ follows from the fact that any use of $A \to \alpha$ in a G' derivation can be desugared by G steps deriving α from A, so we an turn any G' derivation into a G derivation of the same terminal string. ///

The next observation is a little trickier: it allows us to *eliminate* a rule, if we pay for it by adding certain others.

21.7 Lemma (Replacing Rules)**.** *Suppose $G = (\Sigma, V, S, P)$ is a CFG and*

- $A \to \alpha B \gamma$ *is a rule in G, and*
- $B \to \beta_1 \mid \ldots \mid \beta_k$ *are **all** the B-rules of G.*

If we build G' from G by

- *removing the rule $A \to \alpha B \gamma$ from G, and*
- *adding each of the rules $A \to \alpha\beta_1\gamma \mid \ldots \mid \alpha\beta_k\gamma$*

then $L(G') = L(G)$.

Proof. If we temporarily let G'' be the result of adding the $A \to \alpha\beta_i\gamma$ rules, we have that $L(G'') = L(G)$ by the previous Adding Rules lemma 21.6.

So the lemma will follow if can show that $L(G'') = L(G)$, and for that it suffices to show that any derivation of a terminal string in G'' need not use the rule $A \to \alpha B \gamma$ after all.

So consider a parse tree in G'' that somewhere used the rule $A \to \alpha B \gamma$. This means that in the parse tree we have an occurrence of A whose children are the symbols in $\alpha B \gamma$. (Draw a picture!) Now consider the children of B in this tree: they will be the symbols of one of the β_i. Now consider the tree in which we replace B under A with the symbols of β_i. This represents using the rule $A \to \alpha\beta_i\gamma$ instead of $A \to \alpha B \gamma$. By successively doing this construction we can eliminate all uses of $A \to \alpha B \gamma$, and so have a parse tree which is really parse tree for the grammar G'

///

21.3 Reachable Variables

21.3.1 Computing Reachable Variables

Computing the reachable variables is very easy: we just start with S and "walk forward", remembering which variables we can get to.

Algorithm 28: Compute Reachable Variables

Input: a CFG $G = (\Sigma, V, S, P)$
Output: the set of reachable variables of G
initialize: $V' = \{S\}$;
repeat
  if *there is a rule* $A \to \alpha$ *with* $A \in V'$ **then**
    add each variable of α to V'
until *no change in* V';
return V'

Proof of Correctness. To argue the correctness of this algorithm we need to

- argue that it terminates on all inputs, and
- argue that when it terminates then V' is indeed the set of reachable variables.

The former claim, termination, follows from the observation that the number of steps of the repeat loop is bounded by the number of variables in G. For the latter claim we need only check that (i) every variable added to V' is clearly reachable, by an easy induction over the number of iterations of the repeat loop, and (ii) if a variable A *is* reachable, it will be added to V'; a formal proof would be by induction over the number of steps in a derivation resulting in a word containing A. ///

21.3.2 Eliminating Unreachable Variables

Algorithm 29: Eliminate Unreachable

Input: a CFG $G = (\Sigma, V, S, P)$
Output: CFG G' with $L(G') = L(G)$ and no unreachable variables in G'
let V' to be the result of *ComputeReachable* on G;
set P' to be those rules of P involving only symbols from $V' \cup \Sigma$;
return $G' = (V', \Sigma, S, P')$

Proof of Correctness. To argue the correctness of this algorithm we need to

- argue that it terminates on all inputs, and

- argue that when it terminates with the output grammar G' we have $L(G') = L(G)$.

The former claim, termination, is immediate from the fact that *ComputeReachable* halts on all inputs. For the latter argument, we want to show that $L(G') = L(G)$. The fact that $L(G') \subseteq L(G)$ follows immediately from the fact that the rules of G' re a subset of the rules of G. To show that $L(G) \subseteq L(G')$ it suffices to show that no derivation in G will ever use a rule that we excluded from G'. This is to say that no derivation in G ever uses a rule involving an unreachable symbol. But this is clear. ///

21.4 Generating Variables

21.4.1 Computing Generating Variables

Computing the generating variables is as easy as computing the reachable ones; we just have to reason backwards.

Algorithm 30: Compute Generating

Input: a CFG $G = (\Sigma, V, S, P)$
Output: the set of generating variables of G
initialize: $V' = \emptyset$;
repeat
  if *there is a rule* $A \to \alpha$ *with each symbol in* α *in* $\Sigma \cup V'$ **then**
    add A to V'
until *no change in* V';
return V'

Proof of Correctness. The algorithm terminates on all inputs because the number of steps of the repeat loop is bounded by the number of variables in G. The argument that V' meets its specification is left to you. ///

21.4.2 Eliminating Non-Generating Variables

We can eliminate rules involving non-generating variables without changing the language.

Algorithm 31: Eliminate Non-Generating

Input: a CFG $G = (\Sigma, V, S, P)$
Output: CFG G' with $L(G') = L(G)$ and no non-generating variables in G'
let V' to be the result of *ComputeGenerating* on G;
set P' to be those rules of P involving only symbols from $V' \cup \Sigma$;
return $G' = (\Sigma, V', S, P')$

Proof of Correcteness. similar to the argument for *Eliminate Unreachable* ///

21.4.3 Nullable Variables

The technique for identifying nullable variables is similar to the technique for identifying generating variables, except that we work backwards.

Algorithm 32: Compute Nullable Variables

Input: a CFG G
Output: the set of nullable variables of G

initialize *Nullable* to be $\{A \mid A \to \lambda \text{ is a rule } \}$;
repeat
  if *there is a rule* $A \to \alpha$ *such that each element of* α *is in Nullable* **then**
    add A to *Nullable*
until *no change in Nullable*;
return *Nullable*

Proof of correctness. Similar to the proofs of correctness of Algorithm 24 and 26. Left as an exercise. ///

21.5 Useless Rules

A rule is *useless* if it can never be used in generating a terminal string. This will happen if the rule involves a non-reachable variable or if it involves a non-generating variable. So we can just use the previous two algorithms to eliminate such rules ...provided we are a little bit careful: see Remark 21.9.

Algorithm 33: Eliminate Useless Rules

Input: a CFG $G = (\Sigma, V'S, P)$ with $L(G) \neq \emptyset$
Output: a grammar G'' equivalent to G with no useless variables

let G' be the result of *Eliminate Non-Generating* on G;
let G'' be the result of *Eliminate Non-Reachable* on G';
return G''

Proof of Correctness. The algorithm terminates on all inputs because each of *Eliminate Non-Generating* and *EliminateUnreachable* are known to terminate. The fact that the output G'' satisfies $L(G'') = L(G)$ follows from the fact that each of *EliminateNon-Generating* and *Eliminate Unreachable* are known to preserve the languages of their input grammar. To see that G'' has no useless variables we must show that it has no unreachable variables and that it has no non-generating variables. The first fact follows from the fact that *Eliminate Unreachable* is known to return a grammar with no unreachable variables. The second fact follows from the fact that *Eliminate Non-Generating* is known to return a grammar with no non-generating variables **and** the fact that *Eliminate Unreachable* will not *introduce* any non-generating variables (when it is given the grammar *Eliminate Non-Generating(G)*). ///

21.8 Example. Let G be the grammar

$$\begin{aligned} S &\to AB \mid aA \\ A &\to bA \mid c \mid D \\ C &\to cB \mid c \end{aligned}$$

The generating variables are C, A, and S. Eliminating the rules involving the non-generating symbols B and D we get to

$$\begin{aligned} S &\to aA \\ A &\to bA \mid c \\ C &\to c \end{aligned}$$

The reachable variables of this grammar are S and A. Eliminating the rules involving the non-reachable C yields

$$\begin{aligned} S &\to aA \\ A &\to bA \mid c \end{aligned}$$

This grammar generates the same language as the original G, and has no useless rules.

21.9 Remark. Maybe the above proof of correctness seems to be fussier than it needs to be? Consider the following issue. In the algorithm above we *first* eliminate the non-generating variables, and *then* eliminate the non-reachable variables. What if we did things in the other order? Have a look at Problem 210.

21.6 Chain Rules

Our goal is to prove the following theorem.

21.10 Theorem. *For every CFG G there exists a CFG G' such that*

1. $L(G') = L(G)$
2. *G' has no chain rules*

Furthermore, there exists an algorithm to compute G' from G.

Note that we can't just *delete* chain rules of course: we must compensate for removing them. The strategy:

1. Build an auxiliary grammar G^*;
2. Define G' as G^* with all chain rules deleted

The key idea of the algorithm is contained in the following lemma. It expresses the fact that a certain transformation of a grammar leaves the language of the grammar unchanged.

> *Suppose grammar G has the rule $A \to B$ and a rule $B \to \beta$. Then if we build a new grammar by* adding *the rule $A \to \beta$, the resulting grammar generates exactly the same language as did G.*

This is just an example of Lemma 21.6 at work.

That lemma seems to take us in the wrong direction: it *adds* rules rather than *removing* the ones we don't want. But the trick is that after all such rules have been added, we can *then* just delete the bad rules. Here is this idea expressed as an algorithm.

Algorithm 34: Eliminate Chain Rules

Input: a CFG $G = (\Sigma, V, P, S)$
Output: a CFG G' such that $L(G') = L(G)$ and G' has no chain rules
initialize: set P^* to be P
repeat
  if P^* has a chain rule $A \to B$ and a rule $B \to \beta$ then add $A \to \beta$ to P^*
until *no change in $P*$*;
define P' to be P^* with all chain rules removed ;
return $G' = (\Sigma, V, P', S)$;

Proof of correctness of EliminateChain. We need to prove three things: (i) the algorithm always terminates; (ii) the output G' has no chain rules, and (iii) L(G') = L(G).

Termination: notice that if a new rule is added to the grammar, the left-hand side of that rule is a variable in the original grammar, and the right-hand side of that rule must be a rule in the original grammar. There are only finitely many such potential new rules, in fact no more than $|V||P|$ such. So there are only $|V||P|$ rules that it is possible to be added. This establishes an upper bound on the number of times the repeat loop can be executed.

The fact that the output G' has no chain rules is obvious from the algorithm statement.

To prove $L(G') = L(G)$: we first prove $L(G^*) = L(G)$, and then prove $L(G') = L(G^*)$.

To prove $L(G^*) = L(G)$: Since we start with $P^* = P$ and we never delete rules, it is obvious that $L(G) \subseteq L(G^*)$. To prove $L(G^*) \subseteq L(G)$ it suffices to prove that *at each stage* of the construction of the rules P^* the grammar at that stage generates no new Σ strings. But this is immediate from Lemma 21.6.

Now to prove $L(G') = L(G^*)$: Since the rules of G' are a subset of those of G^* it is obvious that $L(G') \subseteq L(G^*)$. To prove $L(G^*) \subseteq L(G')$. It suffices to prove that for any string $w \in L(G^*)$ there is a derivation of w in G^* that does not use chain-rules.

For this it suffices to show the following claim:

> If $S \stackrel{*}{\Longrightarrow} w$ is the shortest leftmost derivation of w in G^* then no chain rule of G^* is used.

Proof of claim: For sake of contradiction suppose that somewhere a chain-rule was used

$$S \stackrel{*}{\Longrightarrow} xB\alpha \Rightarrow xC\alpha \stackrel{*}{\Longrightarrow} w$$

we must have next rewritten C via some rule $C \to \beta$

$$S \stackrel{*}{\Longrightarrow} xB\alpha \Rightarrow xC\alpha \Rightarrow x\beta\alpha \stackrel{*}{\Longrightarrow} w$$

but then if $B \to C$ in P^* and also $C \to \beta \in P^*$ we must have $B \to \beta \in P^*$ [this is just from the way P^* was constructed]. So in fact there is a shorter derivation:

$$S \stackrel{*}{\Longrightarrow} xB\alpha \Rightarrow x\beta\alpha \stackrel{*}{\Longrightarrow} w.$$

This contradicts our assumption that the derivation we started with was shortest. So this completes the proof that grammar G' performs as advertised. ///

21.11 Remark. As a final remark, we connect the notion of chain rules in *CFGs* with λ-transitions in automata. Recall that an NFA_λ is a finite automaton that, in addition to transitioning from one state to another while reading an input symbol, has the capacity to transition from one state to another without consuming any input at all. We denote such transitions, naturally, as $p \to q$, and call them λ-transitions, or sometimes, "silent" transitions.

These machines are convenient sometimes, but we proved earlier that λ-transitions can be eliminated. The point we want to make here is that the technique we used back in Section 9.6 is *just a special case of eliminating chain rules from a grammar.*

Specifically, suppose M is an NFA_λ. Let G_M be the regular CFG you get in the standard way from M, as in Section 18.2. Then eliminating λ-transitions in M and eliminating chain rules from G_M are *exactly the same algorithm* just expressed in different notation. We just observe that whenever we add rules according the algorithm, the new grammar is still a regular grammar (plus some chain rules) when we delete the chain rules at the last step, we have a regular grammar.

The context-free grammar setting is more general, since not every CFG corresponds to an NFA_λ, but the algorithm for removing λ-transitions from an NFA_λ is just the special case of Algorithm 34 when the input grammar is regular.

21.12 Check Your Reading. *Make sure this last remark is clear to you, by doing this once or twice: (i) looking at a previous example of eliminating λ-transitions from some NFA_λs M (resulting in some M') (ii) writing down the regular grammar corresponding to M (iii) eliminate chain rules from the grammar, (iv) and check that you get exactly grammar corresponding to M'*

21.7 Erasing Rules

Erasing rules are, intuitively, of little use, since they don't contribute to the rule of a non-λ output string. And indeed we will in this section show how to eliminate them.

But there is an annoyance we have to confront. If a grammar has no erasing rules then λ can never be derived as an output (this is easy to see). So if we start with a grammar G that does derive λ as an output, we can never replace it with a G' that is both equivalent to G and also has no erasing rules. Something has to give: we either (i) weaken the claim about $L(G) = L(G')$, or (ii) we allow G' to have some minimal amount of erasing.

Some authors choose (ii); we'll choose (i).

So our goal is to prove the following theorem.

21.13 Theorem. *For every CFG G there exists a CFG G' such that*

1. $L(G') = L(G) - \{\lambda\}$, *and*
2. G' *has no erasing rules*

Furthermore, there exists an algorithm to compute G' from G.

Note that we can't just *delete* erasing rules of course: we must compensate for removing them (by now this is a familiar pattern to you, right?).

21.14 Example.

$$S \to AB$$
$$A \to a$$
$$B \to \lambda$$

Here $L(G) = \{a\}$. But if we delete the erasing rule $B \to \lambda$, then $L(G) = \emptyset$.

So we have to be a little more careful. This is where the idea of *nullable* variables come into play.

21.7.1 Eliminating Erasing Rules

The strategy for eliminating erasing rules is similar—at the top level—to the one we use to eliminate chain rules.

1. We build an auxiliary grammar with extra rules designed to compensate for not having erasing rules. This will involve working with nullable variables.
2. *Then* we delete erasing rules from this augmented grammar.

For instance, in the little grammar of Example 21.14, we would first add the new rule $S \to A$, reflecting the fact that B is nullable, and *then* remove the rule $B \to \lambda$. In general grammars things are more complex; let's describe things carefully.

21.15 Remark. Suppose grammar G has a rule $A \to \alpha B \beta$ (with $\alpha, \beta \in (V \cup \Sigma)^*$), and suppose that B is nullable. Then if we build a new grammar by *adding* the rule $A \to \alpha\beta$, the resulting grammar generates exactly the same language as did G.

Why did we call the above a "Remark" and not a "Lemma" or something? Because it is just an instance of Lemma 21.6: to say B is nullable is to say that $B \stackrel{*}{\Longrightarrow} \lambda$ is a derivation in G, and the new rule added in the above remark is the result of replacing B by λ in $A \to \alpha B \beta$.

Now we can isolate the key step in the algorithm as a definition all its own. It consists of just iterating the above processes as much as we can.

21.16 Definition. Let $G = (\Sigma, V, P, S)$. Let P^λ be the smallest set of rules such that

1. every rule of P is in P^λ, and
2. if $A \to \alpha B \beta$ is in P^λ (with $\alpha, \beta \in (V \cup \Sigma)^*$), and B is nullable in G, then $A \to \alpha\beta$ is in P^λ.

Be careful to note that Definition 21.16 can require adding several "variations" for a given rule, as the next example shows.

21.17 Example. Suppose for example that our grammar had B and C be nullable. Suppose G had the rule

$$A \to aBcCB.$$

Then we would end up adding *seven* new rules to P^λ:

$$\begin{array}{ll} A \to acCB & A \to aBc \\ A \to aBcB & A \to acC \\ A \to aBcC & A \to ac \\ A \to acB & \end{array}$$

We add one new rule for each non-empty subset of the occurrences of nullable variables.

We're ready to give the algorithm for eliminating erasing rules.

Algorithm 35: Eliminate Erasing Rules

Input: a CFG $G = (\Sigma, V, P, S)$
Output: a CFG $G' = (\Sigma, V, P_{new}, S)$ such that $L(G') = L(G) - \{\lambda\}$ and G' has no λ-rules

compute the nullable variables of G;
define $G^\lambda = (\Sigma, V, P^\lambda, S)$ where P^λ is defined as in Definition 21.16
return $G' = (\Sigma, V, P^0, S)$ where P^0 is P^λ with all erasing rules removed

To prove that Algorithm 35 is correct we identify two lemmas. The first says that at the place in the algorithm where we have expanded G to G^λ, we have not changed the language generated.

21.18 Lemma. *Let $G = (\Sigma, V, P, S)$. Let P^λ be defined as in Definition 21.16. Then if we set $G^\lambda = (\Sigma, V, P^\lambda, S)$, we have $L(G^\lambda) = L(G)$.*

Proof. This is an easy consequence of Lemma 21.15. ///

The second lemma says that in G^λ , to derive a non-λ string, we never need to use any erasing rules.

21.19 Lemma. *Let $G = (V, \Sigma, P, S)$. Let P^λ be defined as in Definition 21.16, and set $G^\lambda = (\Sigma, V, P^\lambda, S)$.*

Then for $w \neq \lambda$, if $S \stackrel{*}{\Longrightarrow} w$ then there is a derivation in which no λ-rule of G^λ is used.

Proof. It will be convenient to actually prove a more general statement. Namely:

> *(**) For any $x \in \Sigma^*$ and $\delta \in (V \cup \Sigma)^*$, if $S \stackrel{*}{\Longrightarrow} x\delta$ is the shortest leftmost derivation of $x\delta$ in G^λ, no λ-rule of G^* is used.*

Note that our lemma follows from this statement by taking $x = w$ and δ to be λ.

(i) for sake of contradiction suppose that somewhere a λ-rule was used

$$S \stackrel{*}{\Longrightarrow} xB\delta \Rightarrow x\delta$$

This occurrence of B originated by a derivation step using a rule $A \to \alpha B \beta$

$$S \stackrel{*}{\Longrightarrow} x'A\theta \Rightarrow x'\alpha B\beta\theta \equiv x'\alpha B\delta \stackrel{*}{\Longrightarrow} xB\delta \Rightarrow x\delta$$

But then $A \to \alpha\beta$ is in P^* by the way we built P^*. So here is a shorter derivation of $x\delta$:

$$S \stackrel{*}{\Longrightarrow} x'A\theta \Rightarrow x'\alpha\beta\theta \equiv x'\alpha\delta \stackrel{*}{\Longrightarrow} x\delta$$

This contradicts our assumption that the derivation we started with was shortest. ///

Putting those two lemmas together enables us to prove that Algorithm 35 does the right thing, if we don't worry about λ.

21.20 Theorem (Correctness of Eliminate Erasing)**.** *Algorithm 35 computes, given an arbitrary CFG G, a grammar G' such that G' has no λ-rules, and $L(G') = L(G) - \{\lambda\}$.*

Proof. We need to prove three things: (i) the algorithm always terminates; (ii) the output G' has no λ-rules, and (iii) $L(G') = L(G) - \{\lambda\}$.

Termination: notice that if a new rule is added to the grammar, the left-hand side of that rule is a variable in the original grammar, and the right-hand side of that rule must be a substring of the right-hand side of some rule in the original grammar. There are only finitely many such substrings, let us say there are k of them. So there are only finitely many rules that could possibly be added, no more than $k|V|$. This establishes an upper bound on the number of times the repeat loop can be executed.

The fact that the output G' has no λ-rules is obvious from the algorithm statement.

To prove $L(G') = L(G)$: we first prove $L(G^*) = L(G)$, and then prove $L(G') = L(G^*) - \{\lambda\}$.

The fact that $L(G^*) = L(G)$ is Lemma 21.18.

Now to prove $L(G') = L(G^*) - \{\lambda\}$: Since the rules of G' are a subset of those of G^*, and G' does not generate λ, it is obvious that $L(G') \subseteq L(G^*) - \{\lambda\}$. So it remains to prove that $L(G^*) - \{\lambda\} \subseteq L(G')$. Lemma 21.19 showed that any non-λ string $w \in L(G^*)$ has a derivation of w in G^* that does not use λ-rules. But such a derivation is in fact a derivation in G'.

This completes the proof that grammar G' performs as advertised. ///

21.8 Eliminating Both Erasing and Chain Rules

If we want to eliminate all erasing and chain rules from a grammar, *first* eliminate the erasing rules, and *then* eliminate the chain rules.

Algorithm 36: Eliminate Chain And Erasing Rules

Input: a CFG $G = (\Sigma, V, P, S)$
Output: a CFG G'' such that $L(G') = L(G)$ and G'' is non-erasing with no chain rules

let G' be the result of *Eliminate Erasing* on G;
let G'' be the result of *Eliminate Chain* on G';
return G''

21.21 Theorem (Correctness of Eliminate Chain And Erasing Rules)**.** *Algorithm 36 is correct.*

Once again, we have to be careful what order to do things in. See Problem 213.

Summarizing, we have proved the following.

21.22 Theorem. *Let G be a context-free grammar. There is a context-free grammar G' such that*

- *$L(G') = L(G)$, and*
- *G' is non-erasing and has no chain rules.*

Furthermore, there is an algorithm which given G will return the corresponding G'.

21.23 Example. An exercise from [Sud97]. Let G be the following grammar.

$$\begin{aligned} S &\to A \mid B \mid C \\ A &\to aa \mid B \mid \\ B &\to bb \mid C \\ C &\to cc \mid A \end{aligned}$$

There are no erasing rules in G; if we eliminate chain rules we obtain

$$\begin{aligned} S &\to aa \mid bb \mid cc \\ A &\to aa \mid bb \mid cc \\ B &\to aa \mid bb \mid cc \\ C &\to aa \mid bb \mid cc \end{aligned}$$

Obviously A, B, and C are not reachable, so a grammar equivalent to G is given by just the S-rules.

21.24 Example. Let G be the following grammar.

$$\begin{aligned} S &\to aBb \mid aES \\ B &\to PE \mid aBb \mid BB \\ P &\to c \mid E \\ E &\to \lambda \mid BE \mid e \end{aligned}$$

The nullable variables are E, P, and B. If we eliminate erasing rules from G we get

$$\begin{aligned} S &\to aBb \mid ab \mid aES \mid aS \\ B &\to PE \mid P \mid E \mid aBb \mid BB \mid B \mid ab \\ P &\to c \mid E \\ E &\to BE \mid B \mid e \mid E \end{aligned}$$

Note that our algorithm can add chain rules, as it did above (even dumb rules like $B \to B$). That's OK, it's not the current algorithm's job to worry about those.

Now we want eliminate chain rules. When add all the rules first required by our Algorithm, we get to:

$$\begin{aligned} S &\to aBb \mid ab \mid aES \mid aS \\ B &\to BE \mid PE \mid P \mid E \mid aBb \mid BB \mid B \mid ab \mid c \mid E \\ P &\to BE \mid c \mid e \mid B \mid E \mid PE \mid P \mid aBb \mid BB \mid ab \mid c \\ E &\to BE \mid PE \mid P \mid B \mid e \mid P \mid E \mid aBb \mid BB \mid ab \mid c \end{aligned}$$

If we then remove chain rules we arrive at

$$
\begin{aligned}
S &\to aBb \mid ab \mid aES \mid aS \\
B &\to BE \mid PE \mid aBb \mid BB \mid ab \mid c \mid e \\
P &\to BE \mid c \mid e \mid PE \mid aBb \mid BB \mid ab \mid c \\
E &\to BE \mid PE \mid e \mid aBb \mid BB \mid ab \mid c
\end{aligned}
$$

Note that, amazingly, the variables B, P, and E all have the same right-hand sides! So we could eliminate two of them if we felt like it....

21.8.1 Application: *CFG* Membership

An important payoff from the fact that we can eliminate chain and erasing rules from grammars is an algorithm for the problem of deciding whether a given string is derivable in a given grammar: see Chapter 22. It's a naive algorithm, to be sure, but as noted in Chapter 22 it is not obvious at first glance how to come up with *any* algorithm for *CFG* Membership.

21.9 Chomsky Normal Form

Chomsky Normal Form is useful theoretically, and as a prelude to certain other algorithmic constructions on grammars.

21.25 Definition (Chomsky Normal Form)**.** A grammar G is in *Chomsky Normal Form* if

- it is non-erasing ,
- every rule other than $S \to \lambda$ is of one of the forms $A \to A_1A_2$ or $A \to a$. (We allow A_1 and A_2 to be the same.)

21.26 Theorem (Chomsky Normal Form Theorem)**.** *For every CFG G there exists a CFG G_C such that*

1. $L(G_C) = L(G)$
2. *G_C is in Chomsky Normal form*

Furthermore, there exists an algorithm to compute G' from G.

Proof. What are the obstacles to a grammar being in Chomsky Normal Form?

1. There could be non-start rules whose right-hand side has length 0.
2. There could be rules whose right-hand side has length 1, and that right-hand side is not a terminal.
3. There could be rules whose right-hand side has length 2 but isn't a pair of variables.
4. There could be rules whose right-hand side has length greater than 2.

By Theorem 21.13 we may build an non-erasing G' with no chain rules such that $L(G') = L(G)$. This takes care of the first two problems.

To take care of the third problem: for each terminal symbol a,

- create a *new* variable X_a;
- everywhere a occurs in a right-hand of length greater than 1, replace it by X_a;
- add the rule $X_a \to a$

This clearly doesn't change the language generated. And the result is a grammar that will fail to be in Chomsky Normal Form only to the extent that it has rules that look like

$$A \to B_1 B_2 \dots B_n$$

for $n > 2$. To take care of the right-hand sides of length greater than two, proceed as suggested by the following example. if there is a rule

$$A \to BCDE$$

replace this by the rules

$$\begin{aligned} A &\to BB_1 \\ B_1 &\to CC_1 \\ C_1 &\to DE \end{aligned}$$

It should be clear that this results in an equivalent Chomsky Normal Form grammar. ///

21.27 Example. This little grammar obviously generates $\{a^i b^i \mid i \geq 1\}$ Note that by starting i at 1 we've left out λ.

$$S \to ASb \mid ab$$

An equivalent Chomsky Normal Form grammar is

$$\begin{aligned} S &\to AT|AB \\ T &\to XB \\ X &\to AT \mid AB \\ A &\to a \\ B &\to b \end{aligned}$$

21.10 Greibach Normal Form

This section can be read without having read the prior material in this chapter, other than the definition of non-erasing at the beginning.

21.28 Definition (Greibach Normal Form)**.** A grammar is in *Greibach Normal Form* if

- it is non-erasing ,
- every rule other than $S \to \lambda$ has the form

  $$A \to a\beta$$

  where a is a terminal and β is a string (maybe empty) of terminals and variables.

Greibach normal form is a particularly tidy form for grammars, due to the fact that every rule contributes a terminal symbol to the output.

The surprising thing is any grammar can be transformed into an equivalent Greibach Normal Form grammar.

By the way, often Greibach Normal Form is defined with the stronger requirement that the string β consist entirely of variables. This is just fussiness really. We could always satisfy that requirement if someone insisted, by (i) adding a new variable C for each terminal c (ii) adding new rules $C \to c$, and (iii) using C instead of c if need be in the various β. That stronger requirement doesn't make the things we do in this chapter any easier. So we won't bother to use that more restrictive version of the definition.

21.29 Examples.

- Grammars for arithmetic expressions in prefix form, such as this one (where we have simplified to just a a, b, and c as identifiers)

$$E \to +EE \mid *EE \mid -EE \mid a \mid b \mid c$$

  are in Greibach Normal Form.

- The natural grammar for non-empty palindromes

$$S \to aSa \mid bSb \mid \ldots \mid zSz$$

  is in Greibach Normal Form.

Regular grammars are (almost) in Greibach Normal Form, as we show next.

21.30 Example. The grammars we use to capture *DFAs* and *NFAs* can easily be transformed into Greibach Normal Form.

For example the following regular grammar is derived in the usual way from a *DFA* recognizing the words over $\{a,b\}$ of odd length:

$$\begin{aligned} S &\to aT \mid bT \\ T &\to aS \mid bS \mid \lambda \end{aligned}$$

It is not quite in Greibach normal form, because of the rule $T \to \lambda$. But it is easy enough to eliminate the λ-rules to tweak G to get the following equivalent G' be in Greibach Normal Form:

$$\begin{aligned} S &\to aT \mid bT \mid a \mid b \\ T &\to aS \mid bS \end{aligned}$$

If you read Section 21.7.1 you know the trick. If not, here's what we do. Suppose X is a variable with a rule $X \to \lambda$. For every variable Y with rules like $Y \to aX$, add the new rule $Y \to a$. Do this even when X and Y are the same. After adding all these new rules $Y \to a$ you can delete the $X \to \lambda$. What you have left is a Greibach normal form grammar.

For non-regular grammars things aren't this easy. Nevertheless there is an algorithmic method for converting to Greibach Normal Form no matter what *CFG* we start with. We won't give the details here, but you can find it explained elsewhere. It is worth doing some examples by hand, to gain intuition. The technique in Lemma 21.7 can be a very effective tool.

21.31 Example. Start with

$$\begin{aligned} E &\to I+E \mid I-E \mid I \\ I &\to 0 \mid 1 \end{aligned}$$

Here the terminals are $\{0,1,+,-\}$ The obstacles to Greibach Normal Form are the E-rules, specifically the fact that they start with the variable I. But the I-rules are fine. And the Replacing Rules trick in Lemma 21.7 tells us what we can do: replace bad E by some good versions, "substituting in" for I. We get to

$$\begin{aligned} E &\to 0+E \mid 1+E \mid 0-E \mid 1-E \mid 0 \mid 1 \\ I &\to 0 \mid 1 \end{aligned}$$

which is in Greibach Normal Form.

That example was simple because only E-rules violated the Greibach condition, needed attention, and there was only variable, I, causing trouble, and the I-rules themselves needed no attention. Sometimes you have to iterate the above trick, see Problem 217.

For some grammars these easy tricks don't work. But if we are willing to work harder, we can always find an equivalent Greibach Normal Form. We won't prove it, but here is the statement:

21.32 Theorem (Greibach Normal Form Theorem)**.** *Let G be a context-free grammar. There is a context-free grammar G' in Greibach Normal Form such that $L(G') = L(G)$. Furthermore, there is an algorithm which given G will return the corresponding G'.*

Proof. The proof consists of a quite complex construction which we omit here. ///

Greibach Normal Form will be particularly interesting to us when we study *PDAs* and parsing in Chapter 24.

Application: CFG Membership

Greibach Normal Form gives another algorithm for the problem of deciding whether a given string is derivable in a given grammar: see Chapter 22. It's a naive algorithm in the same spirit of the one we get just by eliminating chain and erasing rules, but (i) it is twice as efficient, and (ii) it is a precursor to algorithms used in practice. (see Chapter 25).

21.11 Problems

208. NilMem

a) For the following grammar G, do we have $\lambda \in L(G)$?

$$\begin{aligned} S &\to XZX \\ X &\to aXa \mid Y \mid Z \\ Y &\to bY \mid b \\ Z &\to cZY \mid \lambda \end{aligned}$$

If so, show a derivation; if not, explain your reasoning in a way that generalizes beyond this one example.

b) For the following grammar G, do we have $\lambda \in L(G)$?

$$\begin{aligned} S &\to XZY \\ X &\to aXa \mid Y \mid Z \\ Y &\to bY \mid b \\ Z &\to cZY \mid \lambda \end{aligned}$$

If so, show a derivation; if not, explain your reasoning in a way that generalizes beyond this one example.

209. ElimUseless

Find a grammar generating the same language as the one below, with no useless rules.

$$\begin{aligned} S &\to dS \mid A \mid C \\ A &\to aA \mid a \\ B &\to bB \mid b \\ C &\to cC \end{aligned}$$

210. OrderMatters

Suppose we first eliminated non-reachable variables from a grammar and then eliminated non-generating variables, Show by means of an example that this does not always yield a grammar with no useless rules.

211. ElimPractice

For each grammar

1. Eliminate useless rules, then
2. Give an equivalent grammar with no λ- or chain rules.

a)

$$
\begin{aligned}
S &\to aBbB \\
B &\to P \mid bBb \\
P &\to c \mid E \\
E &\to e \mid \lambda
\end{aligned}
$$

b)

$$
\begin{aligned}
S &\to AB \mid CA \\
A &\to a \\
B &\to BC \mid AB \\
C &\to aB \mid b
\end{aligned}
$$

c)

$$
\begin{aligned}
S &\to aSbb \mid ST \mid c \\
T &\to bTaa \mid S \mid \lambda
\end{aligned}
$$

d)

$$
\begin{aligned}
S &\to aA \mid bB \mid A \\
A &\to aaA \mid B \\
B &\to cc \mid A
\end{aligned}
$$

e)

$$
\begin{aligned}
S &\to ASB \mid \lambda \\
A &\to aAS \mid a \\
B &\to SbS \mid A \mid bb
\end{aligned}
$$

212. ElimChainSize

For each k, define the following grammar. The set of variables is $\{A_1, \ldots, A_k\}$ and the set of terminals is $\{a_1, \ldots, a_k\}$; the start variable is A_1. The rules are:

$$\begin{aligned} &A_i \to A_{i+1} \mid a_i A_{i+1} && \text{for } 1 \leq i < k \\ &A_k \to a_k \end{aligned}$$

What is the size of the grammar you get when you eliminate chain rules? (Count the number of rules; big-Oh notation is fine.)

213. ChainThenElim

Suppose we were to first eliminate the chain rules from a grammar and then eliminate the erasing rules. Show by means of an example that this does not always yield a grammar with no chain or erasing rules.

The reason that things don't go wrong in a similar way when we first eliminate the erasing rules and then eliminate the chain rules is that eliminating chain rules can't introduce any erasing rules if there weren't any to start with. So once we scrub out erasing rules they are gone for good.

214. MakeCNF1

Build Chomsky normal form grammars equivalent to the grammars below.

a)

$$\begin{aligned} S &\to XY \\ X &\to aXbb \mid abb \\ Y &\to cY \mid c \end{aligned}$$

b)

$$\begin{aligned} S &\to aSa \mid aBa \\ B &\to Bb \mid b \end{aligned}$$

c)

$$S \to aSa \mid bSb \mid a \mid b$$

d)

$$\begin{aligned} S &\to ASc \mid AScc \mid ABc \mid ABcc \\ A &\to Aa \mid a \\ B &\to bB \mid b \end{aligned}$$

e)

$$\begin{aligned} S &\to A \mid aAbB \mid ABC \mid a \\ A &\to aA \mid a \\ B &\to bBcC \mid b \\ C &\to abc \end{aligned}$$

215. MakeCNF2

Give grammars in Chomsky Normal Form for each of the following languages.

1. $\{a^n b^{2n} c^k \mid k,n \geq 1\}$
2. $\{a^n b^k a^n \mid k,n \geq 1\}$
3. $\{a^k b^m c^n \mid k,m,n \geq 1, 2k \geq n\}$

216. RegIntersect

Let's prove the result that if L is a CFL and R is a regular language then $L \cap R$ is a CFL. We present an outline; your job is to work through the individual parts.

a) First, the case when $R = \emptyset$ is easy. Why?

Second, there is no harm in assuming that $\lambda \notin L$. Why?

So we will assume below that $L \neq \emptyset$ and $\lambda \notin L$.

b) Since $R \neq \emptyset$, we can write R as a union $R_1 \cup \cdots \cup R_n$ where each R_i is recognized by a DFA with exactly one accepting state. (Justify that.)

c) We have

$$L \cap R = L \cap (R_1 \cup \cdots \cup R_n) = (L \cap R_1) \cup \cdots \cup (L \cap R_n)$$

So it suffices to prove that each of the $L \cap R_i$ is a CFL. (Justify that.)

d) So we have reduced our problem to the simpler problem of showing that the intersection of L with a regular language recognized by a *DFA* with one accept state is a *CFL*.

Here is the construction for that latter result.

We can assume that L is generated by a grammar $G = (\Sigma, V, P, S)$ in Chomsky Normal Form.

Let $M = (\Sigma, Q, \delta, s, \{f\})$ be a *DFA* with one accept state f.

Now we construct the $G = (\Sigma, V', S', P')$, where

- The variables V' are the triples $[p, A, r]$ where p and q are from Q and A is from V.
- S' is $[s, S, f]$
- The rules P' are given by
  1. for each rule $A \to BC$ of P, for each p, q, r of Q, we have this rule in P':
  $$[p, A, r] \to [p, B, q][q, C, r]$$
  2. for each rule $A \to a$ of P, whenever $\delta(p, a) = r$ in M, we have this rule in P':
  $$[p, A, r] \to a$$

This grammar generates $L(G) \cap L(M)$, as desired.

217. GNFPractice

As we mentioned above there is an algorithm for transforming a grammar into Greibach Normal Form, but we didn't present it, in part because it is very tedious to simulate by hand. But to exercise your understanding of the idea you are asked to do an example, just relying on your human cleverness.

Build a Greibach Normal Form grammar equivalent to the following one. (This grammar doesn't derive λ, obviously.)

$$\begin{aligned} E &\to T + E \mid T \\ T &\to I * T \mid I \\ I &\to 0 \mid 1 \end{aligned}$$

Hint. Use Lemma 21.7 repeatedly.

Special note: You do not[1] have to pass to Chomsky Normal Form first. In particular, for this problem it easy enough to build a Greibach Normal Form without changing the set of variables in the grammar.

[1] . . . contrary to what you may gather if you do a web search for computing a Greibach Normal Form.

Chapter 22

The CFG Membership Problem

The problem of *parsing* relative to a CFG G is this: given a string x of terminals, decide whether x can be derived from G; and if so, build a parse tree for x. This is the first step in a compiler, for example. Parsing is an extremely well-developed subject. We don't cover the nitty-gritty details in this course, but we do explore the foundations of parsing in in this chapter and in Chapter 25.

The very first part of parsing, deciding whether a given string can be derived from a given grammar, is already interesting. It is the CFG Membership Problem.

> *CFG Membership*
>
> INPUT: a context-free grammar G and a string x of terminals.
>
> QUESTION: is $x \in L(G)$?

We should say at the outset that there exists better algorithms for CFG Membership than the ones presented here. There are polynomial-time algorithms such as the Cocke-Kasami-Younger algorithm, for arbitrary $CFGs$. Even more important, in practice, is the fact that when a grammar has certain nice properties there are very efficient parsing algorithms. You can find detailed treatments in most compiling texts. Our goal in *this* text is just touch on the foundations and show the connections with our refactoring results.

22.1 Why is Membership Hard?

It's not so obvious how to algorithmically solve the CFG Membership problem.

22.1 Example.

$$S \to aS \mid bA \mid bS$$
$$A \to aB \mid bB$$
$$B \to a \mid b$$

Is *abb* derivable? How about *bba*? How about *bbabbaab*?

22.2 Example. Here the set Σ of terminals is $\{l, r\}$. Think of these as standing for "left" and "right" parenthesis symbols. left and right parenthesis symbols.

$$S \to AB \mid AC \mid SS$$
$$C \to SB$$
$$A \to l$$
$$B \to r$$

Is this string derivable?

$$lrlrllrlrrllrr$$

22.2 A Naive Algorithm

First let's ignore efficiency and try to construct *some* algorithm to do the job. The obvious thing to try is: just start generating derivations in G and wait to see if x is derived.

$$S \Rightarrow \alpha_1 \Rightarrow \cdots \Rightarrow \cdots \Rightarrow x \; ??$$

But this is not yet a correct solution, since we have not given a stopping condition on how long we need look for a derivation.

Is there a natural relation between the length of string and the length of a derivation? Not necessarily:

22.3 Example. Consider:

$$S \to AA$$
$$A \to BB$$
$$B \to b \mid \lambda$$

Then in fact, λ is generated, but takes 7 steps! (Try it)

For an arbitrary CFG there is no nice relationship between the number of steps in a derivation and the length of the string derived. But if a grammar has no chain or erasing rules, we get an easy upper bound on the length of derivations in the grammar.

22.4 Theorem. *Suppose G is an essentially non-erasing grammar with no chain rules. Let x be in $L(G)$, $x \neq \lambda$. Then every G-derivation of x has no more than $2|x|-1$ steps.*

Proof. If $S \Longrightarrow \alpha_1 \Longrightarrow \ldots \Longrightarrow \alpha_k = x$ is a derivation of x then for each step from α_i to α_{i+1} at least one of the following statements is true: (i) the length of α_{i+1} is greater then the length of α_i or (ii) the number of terminals in α_{i+1} is greater then the number of terminals in α_i. There can be at most $(|x|-1)$ steps of the first type, since the length of the string (of terminals and variables) being derived never decreases, and this length is 1 at the start and $|x|$ at the end. There can be at most $|x|$ steps of the second type, since the number of terminals in the string (of terminals and variables) being derived never decreases, and this length is 0 at the start and is $|x|$ at the end. So an upper bound on the total number of steps is $(|x|-1)+|x| = 2|x|-1$. ///

So we can get an algorithm for membership if we (i) play the refactoring game then (ii) use the upper bound we just derived for "nice" grammars. It's not a great algorithm in terms of complexity, but at least it is guaranteed to give an answer.

Algorithm 37: A Naive Algorithm for *CFG* Membership

Input: a CFG $G = (\Sigma, V, P, S)$ and a string $w \in \Sigma^*$
Decides *whether* $w \in L(G)$;
compute grammar G' by Algorithm 36. **if** $w = \lambda$ **then**
  answer yes if there is an erasing rule from the start state of G'
else
  generate all G'-derivations of length no more than $2|w|-1$;
  if *if any of these yield* w **then**
    return yes
  else
    return no

22.5 Theorem. *Algorithm 37 is correct.*

Proof. Given G, let G' be essentially non-erasing and equivalent to G. It suffices to answer the question "$x \in L(G')$?".

If $x = \lambda$, we simply check whether $S \rightarrow \lambda$ is a rule of G'.

When $x \neq \lambda$, Theorem 22.4 gives an upper bound on how long a derivation could be in a grammar with no chain or erasing rules. There are only finitely many derivation sequences of a given length. So we can decide whether $x \in L(G')$ by examining all of the finitely many G'-derivations of length no more than $2|x|-1$. ///

22.6 Corollary. *The membership problem for CFGs is decidable.*

Proof. Algorithm 37 is the decision procedure. ///

22.2.1 Using Greibach Normal Form

If we are working with a grammar in Greibach Normal Form, the previous exhaustive search can be made a little bit more palatable. As we observed in Section 21.10, if a string x is derivable from a Greibach Normal Form grammar, there is a derivation that takes at most $|x|$ step. So we can decide whether $x \in L(G')$ by examining all of the finitely many G'-derivations of length no more than $|x|$.

22.3 Can We Do Better?

It's easy to see that the exhaustive search through the space of derivations in Algorithm 37 has exponential worst-case complexity. So the above is nice as a *decidability* result, but useless in practice. Can we do better? Is there a polynomial-time algorithm for the membership problem?

Here's a two-part answer.

1. Yes, there is a famous algorithm based on dynamic programming called the *Cocke-Kasami-Younger* algorithm (which we don't present in these notes) that solves the membership problem for an arbitrary context-free grammar in time $O(|x|^3)$. This is still not fast enough for most applications; see the next part.
2. Most grammars that arise in practice are well-behaved enough that we can do much better than cubic time.

Fact (2) above is a crucially important fact in the study of compilers. To go further in exploring (2) we would dive in to the fascinating and well-developed study of LL(k) and LR(k) grammars. We don't do that in these notes (see any good book about compilers), but Section 24 of these notes takes the first baby steps in developing real parsing algorithms, starting with a *machine model* for context-free grammars.

22.4 Problems

218. ExhaustingPractice

Look back at Example 22.1. It happens that this grammar has no chain or erasing rules. Use the exhaustive search algorithms (by hand) to answer some membership questions about this grammar.

219. NoEraseCounting

Suppose that G is a CFG with no erasing rules. Prove that if w has a derivation with m steps, then w has a parse tree with $m + |w|$ nodes.

For the inductive step, consider a derivation

$$S \Rightarrow^* \alpha V \gamma \Rightarrow \alpha\beta\gamma$$

of length $m > 0$, apply the induction hypothesis to the derivation of $\alpha V \gamma$ and do some counting.

Explain how the result you just proved shows that if w is in $L(G)$ and has a derivation with m steps, then w has a parse tree with $m + |w|$ nodes.

220. CountCNF

Suppose G is a grammar in Chomsky Normal Form. Let x be in $L(G)$. Prove that every G-derivation of w has exactly $2|x| - 1$ steps. (Compare Theorem 22.4.)

221. CountingCNFSteps

Suppose G is a CFG in Chomsky normal form. Let w be a string of length n which is in $L(G)$. Every derivation of w has the same number of steps. How many? Prove it.

Hint. Use problem 219

222. CountingCNFNodes

Suppose G is a CFG in Chomsky normal form. Let w be a string of length n which is in $L(G)$. Every parse tree for w has the same number of nodes. How many? Prove it.

Hint. Use problem 221

Chapter 23

More CFG Decision Problems

We consider some decision problems concerning context-free grammars. We've already discussed the most important one:

CFG Membership
INPUT: CFG G, string w
QUESTION: $w \in L(G)$?

In this section we will focus on

- the question of whether the language of a given CFG is empty;
- the question of whether the language of a given CFG is infinite;
- questions involving the interaction between $CFGs$ and regular "conditions" on strings.

23.1 Encoding $CFGs$

23.1 Remark. The point made in Remark 13.3.7 is worth re-emphasizing here. The input to the problems below is **not** a *language*. This wouldn't make any sense, since the input to any decision problem must be a finite object, something that can be presented to a computer. So the inputs below are bitstring encodings of context-free *grammars*. Then the question that gets asked is (typically) about the language associated with the CFG.

As usual, we do allow ourselves the convenience of speaking as though the input to a decision problem "is" a *CFG* as opposed to saying more accurately that it is an encoding of a *CFG*. That degree of sloppiness is used by everybody, since the distinction between a grammar and its encoding is not a *deep* distinction, since they are both finite objects. This is in contrast to the distinction between a grammar and its language, which *is* a deep distinction.

We can encode context-free grammars as bitstrings, in exactly the same spirit as we did for automata in Section 13.2. And once again, the *particular* encoding we use will not play a role in our work: all that will matter is that there is a universal recipe to encode all grammars, and that we can easily translate back and forth between grammars and their encodings as strings.

23.1.1 A Particular Encoding

Since it may be reassuring to see a particular encoding at work we'll define one here.

For the first stage of our encoding let us use the alphabet $\Gamma = \{a, V, r, '\}$. Think of a as standing for "alphabet symbol", V as standing for "grammar variable", r as standing for "rule",

We agree to name every alphabet symbol (of any alphabet) as a string consisting of the symbol a followed by a number of tick-marks; and to name every variable (of any grammar) as a string consisting of the symbol V followed by a number of tick-marks.

It should be clear that any grammar can be represented using such names for its terminal alphabet and variables. In the same spirit as our encoding of automata, we might as well insist that every grammar is named in such a way that the start variable is V' (one tick).

Then our encoding is the concatenation of the following pieces:

- For each rule we add a fact to the encoding by writing the strings $r \quad V\cdots \quad \alpha$ where $V\cdots$ names the left-hand side of the rule and α is the string encoding the right-hand side.
- The entire grammar will be encoded by concatenating the codes for the rules in some order.

Note that there are many ways we might use to concatenate our data, but we declare that all of them are legal encodings. (It would not make anything we do later any easier if we tried to insist on unique encodings.)

For example let G be the following *CFG*

$$\begin{aligned} V' &\rightarrow a'V'' \mid \lambda \\ V'' &\rightarrow V'a'' \end{aligned}$$

We have the following encoding in the first stage

$$rV'a'V'' \; rV'rV''V'a''$$

Then we can compile this down to a bitstring encoding by translating each of $\{a, V, r, s, '\}$ to a bitstring and substituting (as we did in Section 13.2). By the way, since our coding alphabet only has 4 symbols, $a, V, r, '$, we can get away with encoding each symbol as a length-2 bitstring (not that it matters...). There's no value in going through that exercise for our example right now.

As usual, the important takeaways here are

1. Every grammar gets at least one encoding;
2. Not every bit string arises as the encoding of an grammar, but that doesn't cause any harm;
3. Encodings are unambiguous in the sense that no bitstring is the encoding of more than grammar.

Encoding Multiple Inputs

We'll use the same conventions as described in Section 13.2.

23.2 *CFG* Emptiness

Remember that when we explored *DFA* decision problems, the Emptiness questions was fundamental.

> *CFG Emptiness*
> INPUT: *CFG* M
> QUESTION: $L(M) = \emptyset$?

This problem is decidable. We did all the work when we showed how to compute the generating variables of a grammar!

We simply observe that to say that the language of a grammar is **not** empty is just to say that the start symbol is generating. So here is our algorithm for testing emptiness:

Algorithm 38: *CFG* Emptiness

if *the start symbol is generating* **then**
  return NO
else
  return YES

23.3 *CFG* Infinite Language

In the next problem we go almost to the other extreme of *CFG* Emptiness. The actual "other extreme" of *CFG* Emptiness would be *CFG* Universality, the question of whether a grammar generates all strings. But amazingly, that is undecidable! We'll show why later.

But if we don't ask for whether a grammar generates *all* strings, but only whether it generates infinitely many, it turns out that we can decide that.

CFG Infinite Language

INPUT: a context-free grammar G

QUESTION: is $L(G)$ infinite?

This problem is decidable. Here is the main idea of the algorithm. First remember that given G we can always construct a grammar G' such that

- G' has no chain or erasing rules.
- Every variable in G' is reachable and generating.
- $L(G') = L(G) - \{nil\}$

If we do that, then certainly $L(G')$ will be infinite if and only if $L(G)$ is infinite. And grammars with the first two properties are easier to work with using the techniques below. So for simplicity below we might as well assume that our input grammar G already has no chain or erasing rules and has every variable reachable and generating.

Here is a little, but crucial, observation. When G has no erasing rules, any (generating) variable X actually generates a terminal string *that is non-*λ. (This is Problem GenerateNonNil below.)

Next we define the following directed graph $\mathcal{D}_G$ based on G

- The set of nodes is the set V of variables of G
- There is an edge $A \rightsquigarrow B$ in the graph precisely if there is some rule in G of the form
$$A \rightarrow \alpha B \beta$$
We use the funny $\rightsquigarrow$ notation for graph edges to avoid confusion with the $\rightarrow$ notation that we use for grammar rules.

Now we check—using, *e.g.*, a standard depth-first search—to see whether $\mathcal{D}_G$ has any cycles.

Claim #1: If there are no such cycles in $\mathcal{D}_G$ then G makes only finitely many parse trees. The way to see that is to realize that if there are no cycles then in every parse tree, a path can have no more than $|V|+1$ nodes. If there are only finitely many parse trees then certainly $L(G) = L(G)$ is finite. (See Problem 224).

Claim #2: Conversely, if $\mathcal{D}_G$ *does* have a cycle, then G will generate infinitely many strings. Here is the proof of that fact.

1. We start with the following general observation: whenever there is a path
$$A \rightsquigarrow \cdots \rightsquigarrow B$$
in the $\mathcal{D}_G$, we have, by the definition of $\rightsquigarrow$, a derivation
$$A \Longrightarrow \ldots \Longrightarrow \alpha B \beta$$
in the grammar.
Since there are no chain rules, at least one of α or β is not λ. And since all variables are generating we can actually say that for some terminal strings z_1 and z_2,
$$A \Longrightarrow \ldots \Longrightarrow \alpha B \beta \Longrightarrow \ldots \Longrightarrow z_1 B z_2$$
and at least one of z_1 or z_2 is not λ.

2. So now suppose there is a cycle in the $\mathcal{D}_G$
$$X \rightsquigarrow \cdots \rightsquigarrow X$$
Using the idea above (with A and B both being X) we conclude that for some terminal strings v and w,
$$X \Longrightarrow \ldots \Longrightarrow vXy$$
and at least one of v or y is non-λ.

3. Now since X is reachable in G, we have, in the $\mathcal{D}_G$,

$$S \rightsquigarrow \cdots \rightsquigarrow X$$

and so in the same way as above we have

$$S \Longrightarrow \ldots \Longrightarrow uXz$$

for some terminal strings v and x. (By the way, we can't say anything abouy u or x being non-λ because for all we know X might be S and the derivation from S to X might have length zero!)

4. Here's where we are so far. We have

$$S \Longrightarrow \ldots \Longrightarrow uXz$$

and

$$X \Longrightarrow \ldots \Longrightarrow vXy$$

and we know that at least one of v or y is not λ.

Gluing the first two derivations together we have

$$S \Longrightarrow \ldots \Longrightarrow uXz \Longrightarrow \ldots \Longrightarrow uvXyz$$

But we can iterate the $X \Longrightarrow \ldots \Longrightarrow vXy$ derivation as many time as we like. In other words, for every n we can derive

$$S \Longrightarrow \ldots \Longrightarrow uXz \Longrightarrow \ldots \Longrightarrow uv^nXy^nz.$$

5. The final observation is that X is generating, so that for some terminal string x we have $X \Longrightarrow \ldots \Longrightarrow x$. This means we can generate the infinitely many strings

$$uv^nxy^nz.$$

Since at least one of v or y is non-λ, we conclude that $L(G)$ is infinite.

The algorithm.

Algorithm 39: *CFG* Infinite Language

Input: *CFG* G
Decides: $L(G)$ *is infinite*

ensure that G has no chain or erasing rules and that every variable of G is reachable and generating ;
construct the directed graph $\mathcal{D}_G$;
do a depth-first search to see if $\mathcal{D}_G$ has any cycles ;
if $\mathcal{D}_G$ *has cycles* **then**
    return YES
else
    return NO

23.4 Regular-restriction *CFG* Decision Problems

Recall that the *CFLs* are not closed under intersection, but that we have the following weaker intersection result (Theorem 18.6):

> If G is a *CFG* and R is a *regular* grammar then we can construct a *CFG* G_R such that $L(G_R) = L(G) \cap L(R)$.

As an application of this result, we can show problems like the following to be decidable.

> *CFG Any Even*
>
> INPUT: *CFG* M
>
> QUESTION: Does $L(M)$ include any even length strings?

Here is a decision procedure.

Algorithm 41: *CFG* Any Even

Input: *CFG* G
Decides: *Does* $L(G)$ *include any even-length strings?*

let R be a regular grammar generating precisely the even-length strings ;
construct a grammar G_R from G and R such that $L(G_R) = L(G) \cap L(R)$. ;
call algorithm *CFGEmptiness* on G_R ;
if *that answer is YES* **then**
  return NO
else
  return YES

Other examples can be handled using the same ideas as in Section 13.

23.5 A Peek Ahead: Two Undecidable Problems

Here is what seems like an easy natural variation on the *CFG* Infinite problem.

23.5.1 *CFG* Universality

> *CFG Universality*
>
> INPUT: *CFG* M
>
> QUESTION: $L(M) = \Sigma^*$?

This problem is *not decidable.* That is, there can be no algorithm to answer, in a finite time, yes or no as to whether an arbitrary *CFG* generates all strings. This is rather amazing in light of the fact that *CFG* Infinite Language is decidable. We discuss this result in Section 39.4.

Recall Chapter 20 where we discussed some ways of removing ambiguity from grammars. How about the decision problem of determining a whether or not a given grammar is ambiguous in the first place?

23.5.2 *CFG* Ambiguity

CFG Ambiguity

INPUT: *CFG G*

QUESTION: is *G* ambiguous?

This problem is *not decidable.* That is, there can be no algorithm to answer, in a finite time, yes or no as to whether an arbitrary *CFG* is ambiguous. We discuss this result in Section 39.3.

23.6 Problems

When a problem asks for a decision procedure, your answer should be precise but non-fussy pseudocode. If you want to use algorithms for any of the following operations just call them, don't derive them.

- *CFG* Membership
- *CFG* Emptiness
- *CFG* Infinite Language
- the construction from Theorem 18.6

223. PathsLeaves

Forget grammars for a minute; focus on ordinary trees. A *branch* in a tree is a sequence $\langle n_1, \ldots, n_p \rangle$ of nodes where n_1 is the root, n_p is a leaf, and each n_{i+1} is a child of n_i. The *length* is this path is the number of nodes, p.

a) A *full binary tree* is a tree T with the property that every node has 0 or 2 children. Finish the following sentence. *If T is a full binary tree and every path has length no greater than p then the number of leaves is no greater than ...*

Prove your answer, by induction on p.

Hint. How to figure out what goes in the ...? The usual advice: just start drawing examples. Keep going until you see the pattern.

What's great about this is that at the point when you see the pattern, and you really believe it, you have also done the work for the proof, really: you understand what happens as you go from some n to the next, so you just have to capture that in your proof.

b) For the general case: fix k. Finish the following sentence. *If T is a tree such that every path has length no greater than p and every node has at most k children, then the number of leaves is no greater than ...*

Prove your answer, by induction on p.

224. FiniteBound

The discussion of Algorithm 35 asserted that if the graph $\mathcal{D}_G$ of a grammar G has no cycles then $L(G)$ is finite. Let's sharpen that claim, by computing a number n such that every string in $L(G)$ has length bounded by n. If we can do that then it is clear that $L(G)$ is finite.

Finish the following sentence. *If $G = (\Sigma, V, S, P)$ is a grammar with no unreachable or non-generating variables such that $\mathcal{D}_G$ has no cycles, then every string in $L(G)$ has length no greater than ...*

Hint. What can you say about the length of the longest path in a parse tree over G? What can you say about the number of children of a node in a parse tree over G? Use Problem 223.

225. GenerateNonNil

Show that when G has no erasing rules, any (generating) variable X actually generates a terminal string *that is non-*λ.

226. UnderstandCGFInfinite

A great way to understand an algorithm or a theorem is to see how each of the assumptions made is really necessary. Let's play this game with Algorithm 35.

a) Show that Algorithm 35 would be incorrect if we had not ensured that G had no erasing rules. Specifically, show a concrete grammar G such that

- G has no chain rules (but erasing rules are allowed)
- every variable of G is generating and reachable
- Algorithm 35 gives the wrong answer as to whether $L(G)$ is infinite.

b) Show that Algorithm 35 would be incorrect if we had not ensured that G had no chain rules. Specifically, show a concrete grammar G such that

- G has no erasing rules (but chain rules are allowed)
- every variable of G is generating and reachable
- Algorithm 35 gives the wrong answer as to whether $L(G)$ is infinite.

c) Show that Algorithm 35 would be incorrect if we had not ensured that every variable of G were generating. Specifically, show a concrete grammar G such that

- G has no chain or erasing rules
- every variable of G is reachable (but non-generating variables are allowed)
- Algorithm 35 gives the wrong answer as to whether $L(G)$ is infinite.

d) Show that Algorithm 35 would be incorrect if we had not ensured that every variable of G were reachable. Specifically, show a concrete grammar G such that

- G has no chain or erasing rules
- every variable of G is generating (but unreachable variables are allowed)
- Algorithm 35 gives the wrong answer as to whether $L(G)$ is infinite.

227. CFGTerminals

Give an algorithm to compute the following function

CFG Terminals
INPUT: A *CFG* $G = (\Sigma, V, P, S,)$
OUTPUT: the set of all $a \in \Sigma$ such that a occurs in some string $x \in L(G)$

Hint. Don't make this too hard!

228. DecAllEven

Give an algorithm for the following decision problem.

CFG All Are Even
INPUT: *CFG* M
QUESTION: Does every string in $L(M)$ have even length?

229. DecInfEven

Give an algorithm for the following decision problem.

CFG Infinite Even
INPUT: *CFG* M
QUESTION: Does $L(M)$ include infinitely many even length strings?

230. DecLength100

Show that the following problem is decidable.

CFG 100
INPUT: a context-free grammar G
QUESTION: does every string in $L(G)$ have length at least 100?

231. CFGRegularDisjoint

Show that the following problem is decidable.

CFG Disjoint From Regular
INPUT: *CFG* G and regular grammar R
QUESTION: $L(R) \cap L(G) = \emptyset$?

232. DecContainsRegular

Show that the following problem is decidable (no matter what R is).

Most Importantly: explain what this problem has to do with (some of) the earlier examples and problems in this section.

CFG ContainsRegular

INPUT: *CFG* G and regular grammar R

QUESTION: $L(G) \subseteq L(R)$?

233. CFGEquality

Show that the following problem is undecidable, *assuming* that the *CFG* Universality problem is undecidable.

CFG Equality

INPUT: *CFGs* G_1, G_2

QUESTION: $L(G_1) = L(G_2)$?

234. CFGSubset

Show that the following problem is undecidable, *assuming* that the *CFG* Universality problem is undecidable.

CFG Subset

INPUT: *CFGs* M_1, M_2

QUESTION: $L(M_1) \subseteq L(M_2)$?

Chapter 24

Pushdown Automata

@@ CLEAN UP: M vs P, use of start state macro

We have a "machine model" for regular languages, namely finite automata. In this chapter we answer the question, "What is the corresponding machine model for membership in context-free languages?"

Specifically, we will address the following decision problem

> *CFG Membership*
>
> INPUT: a context-free grammar G and a string x
>
> QUESTION: is $x \in L(G)$?

This problem is decidable, that is, that there is an algorithm to decide the question for any G and x: see Corollary 22.6.

But this membership problem as stated is not actually the most interesting problem about languages, in practice. The membership problem is a simplified version of the *parsing problem:* given a string x, decide whether $x \in L$ and if so, *compute a parse tree for x.*

But it turns out that it is not difficult at all to tweak what we do here to generate a parse tree. This will be clear once we have done our work. So we will consider what are doing in this chapter to really be parsing.

In fact the parsing problem is in turn a simplified version of what we typically *really* want in practice, namely: given a string x, decide whether $x \in L$ and (i) if $x \in L$, compute a parse tree for x, together with some semantic actions (such as code generation in a compiler), while (ii) if $x \notin L$ return some useful error message to the user about why $x \notin L$.

It is not so straightforward to add semantic actions, or to generate good error messages; that is more advanced work. But what parsing is the essential first step.

24.1 Informal Description

A pushdown automata (*PDA*) is a machine that scans its input from left to right, maintains a notion of current state, and pushes and pops symbols from a stack memory.

In contrast to *DFAs* and *NFAs* we do not define acceptance in terms of "accepting states." It turns out to be more convenient to say that a *PDA* accepts x if there is a run that consumes x and leaves the stack empty. So, in contrast with *DFAs* and *NFAs*, we won't have a notion of accepting state in our *PDAs*.[1]

There are two types of moves possible.

1. Depending on the current state p, the current input symbol a, and the current stack top B, the machine can advance to a next state q, scan past the input symbol, and replace the current stack top by a string β of stack-alphabet symbols.

   This is denoted less verbosely by writing the pair comprising the relevant stuff before the move, namely, (p, a, B) and the relevant stuff after the move, namely, (q, β). That is, the move is captured by the notation

   $$((p, a, B), (q, \beta))$$

2. Depending on the current state p and the current stack top B, the machine can advance to a next state q and replace the current stack top by a string β of stack-alphabet symbols *without* consuming an input symbol.

   We denote such a move by the notation

   $$((p, \bullet, B), (q, \beta))$$

A PDA is, in general, non-deterministic, meaning that more than one move — or indeed no move — might be applicable in a given configuration. So the δ relation on a *PDA* is akin to the δ relation of an *NFA* or an NFA_λ.

Note that we speak of "replacing" the top stack symbol B by a string β. A conventional "pop" means, then, replacing B by the empty string. A conventional "push" mean replacing B by a string β that has B as its first element. And strictly speaking this is really a sequence of pushes: if b has length greater than 2, this is a convenience. Finally note that although we allow the *PDA* to make moves without scanning past input symbols, we have not allowed our *PDA* to move if the stack is empty.

[1] Actually, there is another flavor of *PDA* that can be defined, which does have accepting states. The two kind of *PDAs* are equivalent in the sense that a language is accepted by some *PDA* under final-state acceptance if and only if that language is accepted by some (typically different) *PDA* under empty-stack acceptance. In these notes we will only consider *PDAs* that accept by empty stack.

To complete the definition of a *PDA* we must designate a start state s of the machine. It is also convenient to postulate a special initial stack symbol $\perp$.

24.2 Formal Definition

Making the above discussion precise we have the following

24.1 Definition. A pushdown automaton M is a tuple $(\Sigma, Q, q_s, \Gamma, \perp, \delta)$, where

- Σ is a finite *input alphabet*,
- Q is a finite set of *states*,
- $q_s \in Q$ is the start state
- Γ is a finite *stack alphabet*,
- $\perp \in \Gamma - \Sigma$ is the initial stack symbol.
- δ is a *move relation* a set of tuples of the form

$$\begin{array}{ll} ((p, a, B),\ (q, \beta)) & \text{or} \\ ((p, \bullet, B),\ (q, \beta)) & \end{array}$$

  where $p, q \in Q$, $B \in \Gamma$, $\beta \in \Gamma^*$

Frequently the terminal alphabet Σ will overlap with the stack alphabet Γ, *except* for the constraint that $\perp$ cannot be a terminal symbol (which is why we wrote $\perp \in \Gamma - \Sigma$).

Pictures

Just as with *DFAs* and *NFAs*, it can be helpful to draw pictures of *PDAs*. We just label the arcs with (i) the input symbol being read, if any, and (ii) the action on the stack. See the example, next. We first saw this example in Section 16.2

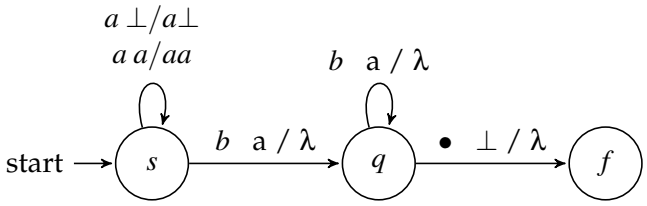
24.2 Example. Here is a *PDA* P designed to accept the language $\{a^n b^n \mid n \geq 1\}$. The intuitive idea is that P reads as, storing them on the stack until it starts to see bs; while it reads bs it pops as off the stack; if the input string is exhausted precisely when the stack has no more as, we accept.

- Σ is $\{a, b\}$
- Q is $\{s, q, f\}$, with s being the start state

- Γ is $\{a, b, \perp\}$, with $\perp$ being the start state
- δ is the following set of pairs

$$((s, a, \perp), (s, a\perp))$$
$$((s, a, a), (s, aa))$$
$$((s, b, a), (q, \lambda))$$
$$((q, b, a), (q, \lambda))$$
$$((q, \bullet, \perp), (f, \lambda))$$

Here is a picture for P. This is nothing more than a way to show the δ relation above in a graphical way.



Here (in English) is how P processes strings. Starting in state s, it reads as and pushes them on the stack. (If the input is empty, P blocks at state s.) Once we see a b in the input, if there is an a on the stack we pop it and move to state q. (If the very first input symbol is a b, we block at state s.)

Once in state q we read bs and pop as from the stack. If there are *more* as on the stack than there are bs to be read, P eventually blocks at state q. If we run out of as on the stack, then, when $\perp$ is at the top of the stack, P can move to state f without reading a symbol. If there are *fewer* as on the stack than there are bs to be read, then we eventually get to the point where $\perp$ is the top of the stack, and so P can take the transition to state f. The stack will be empty, but the input string will not have been read completely, so we do *not* have an accepting run. Finally, if there are *the same number* of as on the stack as there are bs to be read, then we eventually get to the point where $\perp$ is the top of the stack, and the input string has been completely read. P can take the transition to state f, and this will be an accepting run since the input string has been completely read and the stack is empty.

By the way, whatever your feelings about mathematical notation, Greek letters, and such, that last paragraph should be persuasive that English prose is not a very good way to describe computing systems.[2] It's both verbose and potentially ambiguous.

In that spirit let us be rigorous in defining what a computation is and what it means to accept a string. Read on.

[2] and I took a lot of care to make that paragraph as crisp as possible!

24.2.1 Configurations, Transitions, and Computations

Here we define what a *computation* of a *PDA* is. First we say what a *configuration* of a *PDA* is: this is a snapshot of everything relevant about a *PDA* at a given moment in time. These are sometimes called "instantaneous descriptions." Then we define how a *PDA* can make a *transition* from one configuration to another. Finally, a computation is a sequence of transitions, starting with an initial configuration.

Configurations

To describe the configuration of a *PDA* at a particular instant of time we must specify the current state, the portion of the input string remaining to be processed, and the current sequence of symbols comprising the stack. Formally, then:

24.3 Definition (*PDA* Configuration)**.** Let M be a *PDA*. A *configuration* of M is a triple

$$[\, p,\ w,\ \gamma\,] \quad \text{with} \quad p \in Q, \quad w \in \Sigma^*, \quad \gamma \in \Gamma^*.$$

If w is not empty, think of the machine as "looking at" the first letter of w. If γ is not empty, think of the left-most symbol of γ as being the top of the stack.

An *initial* configuration is one of the form $[\, q_s,\ w,\ \perp\,]$ where q_s is the start state (and of course $\perp$ is the initial stack symbol).

Transitions

A transition is a move from one configuration to another.

24.4 Definition (*PDA* Transition)**.** Let $P = (\Sigma, Q, q_s, \Gamma, \perp, \delta)$ be a *PDA*. The one-step transition relation $\Longrightarrow$ is a binary relation on configurations, defined as follows. Let $[\, p,\ ax,\ B\gamma\,]$ be a configuration.

- If $(\,(p,a,B),\ (q,\beta)\,)$ is a move in δ then

$$[\, p,\ ax,\ B\gamma\,] \Longrightarrow [\, q,\ x,\ \beta\gamma\,].$$

- If $(\,(p,\bullet,B),\ (q,\beta)\,)$ is a move in δ then

$$[\, p,\ x,\ B\gamma\,] \Longrightarrow [\, q,\ x,\ \beta\gamma\,].$$

Note that by definition there are no transitions out of configurations that are not of the form $[\, p,\ ax,\ B\gamma\,]$, that is, if the input OR the stack is empty then the *PDA* halts.

We define the relation $\stackrel{*}{\Longrightarrow}$ to be the reflexive transitive closure of $\Longrightarrow$. This means that

$C \stackrel{*}{\Longrightarrow} D$ if D follows from C by a finite number (zero or more) steps of $\Longrightarrow$.

Note that from a given machine configuration C there may be more than one legal move, meaning that there may be more then one configuration C' with $C \Longrightarrow C'$. By the same token there may no transitions available, that is no C' with $C \Longrightarrow C'$. Note in particular that if the stack is empty in a configuration then there are no transitions from C.

Don't confuse the transition relation $\Longrightarrow$ with the move relation δ. It is the transition relation which describes the action of the machine. The move relation is just syntax for defining the moves (which exist just in order for us to define the transition relation). A good metaphor is: δ is the *program* for the machine, while $\stackrel{*}{\Longrightarrow}$ is the set of *computations.*

Computations

We've done all the work for defining what a computation is.

24.5 Definition (*PDA* Computation)**.** Let $P = (\Sigma, Q, q_s, \Gamma, \bot, \delta)$ be a *PDA*.

A *computation* of P on a word $x \in \Sigma^*$ is a sequence

$$[\, q_s,\ x,\ \bot\,] \stackrel{*}{\Longrightarrow}$$

of transitions starting with the initial configuration on x.

24.3 *PDAs* Recognize Languages

We said informally that a string x is accepted by the *PDA* M if there is a computation of M on x which exhausts all of x and terminates in an accepting state. Here is the precise version of this statement using our definitions above.

24.6 Definition (*PDA* Acceptance)**.** Let $P = (\Sigma, Q, q_s, \Gamma, \bot, \delta)$ be a *PDA*. A string x is *accepted* by M if for some state f

$$[\, s,\ x,\ \bot\,] \stackrel{*}{\Longrightarrow} [\, f,\ \lambda,\ \lambda\,].$$

We denote the set of strings accepted by M as $L(M)$.

If we go back to the intuition earlier about *PDAs* as programs, we can think of the input string as generating tasks to be done, and the stack as being a place to keep track of what tasks are waiting to be done. Under this intuition, acceptance by empty stack corresponds to say that we accept if we have read the entire input string and have completed all the tasks on the stack.

24.7 Example (continued)**.** We continue Example 24.2 by showing some computations.

On input *aab*:

$$\begin{aligned}[s,\ aab,\ \bot] &\Longrightarrow [s,\ ab,\ a\bot] \\ &\Longrightarrow [s,\ b,\ aa\bot] \\ &\Longrightarrow [q,\ \lambda,\ a\bot]\end{aligned}$$

and now the machine blocks, since, in state q, if the stack top is a, the only way to make a move is to scan a b...but our input string is exhausted. Note that this is not a run on *aab*, since runs have to consume their entire input. We didn't have any non-deterministic choices in this computation, so it is not hard to see that there can be no accepting run on *aab*. That is, *aab* is not in the language accepting by P.

On input *abb*:

$$\begin{aligned}[s,\ abb,\ \bot] &\Longrightarrow [s,\ bb,\ a\bot] \\ &\Longrightarrow [q,\ b,\ \bot] \\ &\Longrightarrow [f,\ b,\ \lambda]\end{aligned}$$

and now the machine blocks, since there are no δ-moves defined out of f at all. This is also not a run, since runs have to consume their entire input. The fact that the stack is empty at the end of this computation is irrelevant. We didn't have any non-deterministic choices in this computation, so it is not hard to see that there can be no accepting run on *abb*. That is, *abb* is not in the language accepting by P.

On input *aabb*:

$$\begin{aligned}[s,\ aabb,\ \bot] &\Longrightarrow [s,\ abb,\ a\bot] \\ &\Longrightarrow [s,\ bb,\ aa\bot] \\ &\Longrightarrow [q,\ b,\ a\bot] \\ &\Longrightarrow [q,\ \lambda,\ \bot] \\ &\Longrightarrow [f,\ \lambda,\ \lambda]\end{aligned}$$

This is a run. It ends with the stack empty, so it is an accepting run. Thus *aabb* is in the language accepting by P.

24.8 Check Your Reading. *Work out some other computations. Don't read any further until you do this!*

What happens if the empty string λ is the input to PDA? How would you change P to get a PDA that accepts $\{a^n b^n \mid n \geq 0\}$?

24.4 Non-Determinism

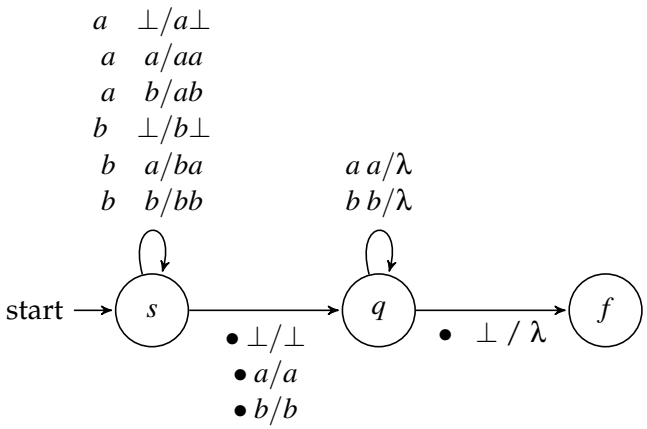
In general, *PDAs* can be non-deterministic: from certain configurations there can be more than one possible transition, or no possible transition. A *PDA* is said to be deterministic if, for every state p, input symbol c and stack symbol X, one of the following holds, but not both:

- there is exactly one (q,β) such that $((p,c,X),(q,\beta)) \in \delta$)
- there is exactly one (q,β) such that $((p,\bullet,X),(q,\beta)) \in \delta$)

In contrast to the situation with *NFAs*, non-determinism in *PDAs* is essential. By this we mean that there are *PDAs* N such that there is no deterministic *PDA* D with $L(D) = L(N)$.

We won't prove that here. But we give an example next: the *PDA* in Example 24.9 below cannot be converted to a deterministic *PDA*. In other words, there is no deterministic *PDA* D such that $L(D) = \{ww^R \mid w \in \{a,b\}^*\}$.

24.9 Example. Here is another example, which shows the power of, and the need for, non-determinism in *PDAs*. The following *PDA* accepts the language $\{ww^R \mid w \in \{a,b\}^*\}$. of even-length palindromes over $\{a,b\}$. (Remember that w^R stands for the reverse of string w.) It is similar to the *PDA* of Example 24.2 in that it pushes symbols onto the stack during the "pushing phase" of the computation and pops them off during the "popping phase", but it is different in two ways. First, it is happy to see either as or bs in the first phase, it simply checks that symbols match when it is time to pop. The second difference is the interesting one: unlike the $a^n b^n$ language there is no explicit way that the input string says, "ok, now it is time for the my second half." So our *PDA* has to guess when to jump from the pushing phase to the popping phase. That is shown in the diagram below by the fact that the transitions from s to q dont scan any input and don't change the stack.



Let's look at some computations.

On input *abba*

$$\begin{aligned}
[\,s,\ abba,\ \bot\,] &\Longrightarrow [\,s,\ bba,\ a\bot\,] \\
&\Longrightarrow [\,s,\ ba,\ ba\bot\,] && \text{the guess happens next} \\
&\Longrightarrow [\,q,\ ba,\ ba\bot\,] \\
&\Longrightarrow [\,q,\ a,\ a\bot\,] \\
&\Longrightarrow [\,q,\ \lambda,\ \bot\,] \\
&\Longrightarrow [\,f,\ \lambda,\ \lambda\,]
\end{aligned}$$

This is a run. It ends with the stack empty, so it is an accepting run. Thus *abba* is in the language accepting by P.

Here's another computation starting with *abba*, in which the machine guesses wrong.

$$\begin{aligned}
[\,s,\ abba,\ \bot\,] &\Longrightarrow [\,s,\ bba,\ a\bot\,] \\
&\Longrightarrow [\,s,\ ba,\ ba\bot\,] \\
&\Longrightarrow [\,s,\ a,\ bba\bot\,] && \text{oops, waited too long to guess} \\
&\Longrightarrow [\,q,\ a,\ bba\bot\,]
\end{aligned}$$

and we are stuck now, since state q won't pop if the current input symbol doesn't match the stack top. But just as with *NFAs*, the fact that there is *some* accepting run on string *abba* is enough to say that the machine accepts *abba*.

On an input that is not an even-length palindrome, there will be no way for the machine to accept, since there will be no way for the machine to guess correctly and have the pushes and pops match up.

24.10 Check Your Reading. *Make some other computations on the machine above, with palindrome inputs and non-palindrome inputs, to be certain you understand how the machine works.*

Problem 235 asks you to modify this *PDA* so that it accepts even-length palindromes *and* odd-length palindromes.

24.5 Problems

235. PDAPal

Let $\Sigma = \{a, b\}$.

Construct a *PDA* M accepting palindromes, *i.e.*, $L(M) = \{x \mid x = x^R\}$.

Don't use the generic construction in Definition 25.2, modify the *PDA* in Example 24.9.

Do a computation on input *aba*; show the corresponding grammar derivation.

Hint. The even-length palindromes were written as $\{ww^R \mid w \in \{a,b\}^*\}$ in Example 24.9. The odd-length palindromes can be written as

$$\{waw^R \mid w \in \{a,b\}^*\} \cup \{wbw^R \mid w \in \{a,b\}^*\}$$

We want to build a *PDA* accepting these strings in addition to those in Example 24.9. Just modify the *PDA* in Example 24.9. You don't even have to add any states.

Chapter 25

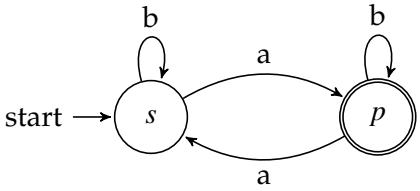
Pushdown Automata and Parsing

25.1 Warm Up: *DFAs* Are Parsers for Regular Grammars

Suppose the *CFG G* we start with is regular (Section 18.2). For instance, take *G* to be

$$S \to aP \mid bS$$
$$P \to aS \mid bP \mid \lambda$$

Let *M* be a *DFA* for $L(G)$. For instance take *M* to be



Then we can think of the states of *M* as being procedures in a standard programming language:

- procedure *S* reads a symbol; if the char is an *a* it calls procedure *P*, else if the symbol is *b* it calls itself; else if the char is end-of-file the program rejects.
- procedure *S* reads a symbol, and if the char is an *a* it calls procedure *S*, else if the symbol is *b* it calls itself; else if the char is end-of-file the program accepts.

It's obvious that this a perfectly general way to think about any *DFA*.

If instead we have an *NFA* for our language, things are slightly more subtle. The non-determinism in *NFAs* mean that to simulate them (naively) by programs we have to use backtracking. This is why it is so nice that *NFAs* can be compiled into *DFAs*.

So. What goes wrong with the story above if the grammar we start with is not regular? The crucial thing about the story above is that each of our procedures, no how complex the *DFA* or *NFA*, reads symbols, then simply branches to another procedure in a *tail recursive* way. What "tail recursive" means is that the procedure call is just a simple goto, and the procedure being called does not have to return anything to the calling procedure.[1]

Even a very simple example shows us what the issue is in the non-regular case. Look at this grammar

$$S \to aAa \mid bAb$$
$$A \to bSb$$

If we think of S as being a procedure in a standard program, it does this (the story for procedure A is similar).

1. procedure S reads a symbol; then makes a call to procedure A
2. *when that call returns*, it reads another symbol and makes sure that it is the same as the char read before the call

It is precisely that business of making a procedure call, expecting that call to return, and doing some subsequent work, that is the essence of the difference between regular and context-free language processing.

In particular, the difference between finite automata, our machine model for regular languages, and pushdown automata, the machine model we develop now, is that pushdown automata have some memory. In fact it suffices to have a *stack* memory. This stack memory is a direct analog of the run-time stack that you know about from studying how programs are executed on a standard computer architecture.

25.2 *PDAs* and *CFGs*

Now we connect *PDAs* and context-free grammars. The examples in the previous section seemed to require some cleverness: the *PDA* we built for (for example) the palindrome language didn't seem to be derived in any systematic way from the grammar we have for this language. So it is somewhat amazing that we will be able to show that for any context-free grammar G at all we can build—systematically—a *PDA* M with $L(M) = L(G)$. That is the content of the first part of the theorem below. It is also true that for every *PDA* there is a corresponding grammar; that is the content of the second part of the theorem.

[1] The word "recursive" in the phrase "tail recursive" is an unfortunate accident of history; people use this expression to refer to any procedure-calling situation, recursive or not, where the last thing that happens is a simple jump...

We won't give a formal proof of the theorem here, but we will give the construction of the *PDA*, for the first part, and do some examples.

25.1 Theorem. *Context-Free grammars and pushdown automata are equivalent for defining lanaguges, in the following sense.*

1. *For every context-free grammar G there is a PDA M such that $L(M) = L(G)$. Furthermore there is an algorithm for computing M from G.*
2. *For every PDA M there is a context-free grammar G such that $L(G) = L(M)$. Furthermore there is an algorithm for computing G from M.*

Proof Idea. The second part of the theorem is very tedious to prove and we will not have reason to *use* the result. So we will skip it.

The first part of the theorem is useful, because it provides a first step in developing real parsing algorithms. So we will explore it a little. Specifically, we will give the construction that starts with any *CFG G* and yields a *PDA M* such that $L(P) = L(M)$. We will not formally prove that the construction works, but the construction is simple enough that an intuitive explanation for why it works will be convincing to you. ///

Here is the construction that is the basis for the first part of the theorem. It is surprising that we can always build the *PDA* we want with only one state.

25.2 Definition (*PDA* from a *CFG*)**.** Let G be (Σ, N, S, P). We construct a *PDA* M from G with the following components.

- one state q, which is of course the start state
- input alphabet Σ
- stack alphabet $N \cup \Sigma$,
- the initial stack symbol is the start symbol S
- the move relation δ is as follows:
  1. for each rule $A \to \alpha$ from P, $((q, \bullet, A), (q, \alpha))$ is a move in δ.
  2. for each symbol $a \in \Sigma$, $((q, a, a), (q, \lambda))$ is a move in δ.

That's the end of the definition; here is the intuition. If string x is presented to M, at any point M will be reading a symbol c of x, and the stack will contain a mix of terminal symbols and variables from G.

While in state q:

- If the top of the stack matches the current input symbol a, that's good, we scan past a in the input and pop a off the stack.
- If the top of the stack is a terminal symbol different from the current input symbol a, that's bad, this attempted run fails (we block).
- If the top of the stack is some variable X, we guess a G-rule of the form $X \to \alpha$, and replace X by α on the stack.

25.3 Theorem. *For any context-free grammar G, the PDA M constructed in Definition 25.2 satisfies $L(M) = L(G)$*

We will not prove the theorem here. But let's see how this works in an example.

25.4 Example. Let G be the following grammar (for the language $\{a^n b^n \mid n \geq 1\}$):

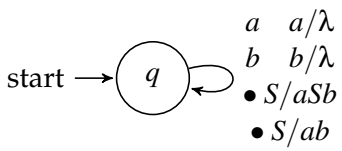
$$S \to aSb \quad | \quad ab$$

The *PDA* M we build has these four δ moves

$$((q,a,a),(q,\lambda)) \qquad ((q,\bullet,S),(q,aSb))$$

$$((q,b,b),(q,\lambda)) \qquad ((q,\bullet,S),(q,ab))$$

A picture:



Here is a run of M on input $aabb$.

$$\begin{aligned}
[\,q,\ aabb,\ S\,] &\Longrightarrow [\,q,\ aabb,\ aSb\,] \\
&\Longrightarrow [\,q,\ abb,\ Sb\,] \\
&\Longrightarrow [\,q,\ abb,\ abb\,] \\
&\Longrightarrow [\,q,\ bb,\ bb\,] \\
&\Longrightarrow [\,q,\ b,\ b\,] \\
&\Longrightarrow [\,q,\ \lambda,\ \lambda\,]
\end{aligned}$$

This is an accepting run, since we have processed the entire input string and the stack is empty. Here is a derivation in the grammar G:

$$S \Longrightarrow aSb \Longrightarrow aabb$$

25.5 Check Your Reading. ***Important!*** *See how the run of M on aabb corresponds in a natural way to the given derivation in G of aabb. Do this by associating each of the derivation steps above with a transition taken by the PDA.*

Perspective

Recall Example 24.2. In that example, we constructed a *PDA* for the same language, $\{a^n b^n \mid n \geq 1\}$, by hand. It looks pretty different from the one we just constructed. Most notably, the one we built by hand did not have to do any guessing, as opposed to the one we built following the general Definition 25.2.

The is an example of a standard phenomenon. When we generate things automatically, using algorithms that have to work for all possible inputs, the results are typically not as nice as when things are written by hand. For example, code generated by a compiler won't be as efficient as hand-crafted assembly language. In our current setting, *PDAs* generated by Definition 25.2 will not take advantage of human insights about specific grammars, which might, for example, eliminate non-determinism.

The key virtue of Definition 25.2 is that it shows how to build a *PDA* for *any CFG*, even if the grammar is too complex for a human to understand it intuitively, with a guarantee that the *PDA* will be correct.

Here are some more examples.

25.6 Example. Suppose we start with the following grammar

$$E \rightarrow E+E \mid E*E \mid E-E \mid 0 \mid 1 \mid 2$$

We get the *PDA* $(\Sigma, Q, q, \Gamma, E, \delta,)$, where δ consists of the following moves. (The numbers are added for future discussion: they aren't part of the *PDA*.)

1. $((q,\bullet,E),(q,E+E))$
2. $((q,\bullet,E),(q,E*E))$
3. $((q,\bullet,E),(q,E-E))$
4. $((q,\bullet,E),(q,0))$
5. $((q,\bullet,E),(q,1))$
6. $((q,\bullet,E),(q,2))$
7. $((q,+,+),(q,\lambda))$
8. $((q,*,*),(q,\lambda))$
9. $((q,-,-),(q,\lambda))$
10. $((q,0,0),(q,\lambda))$
11. $((q,1,1),(q,\lambda))$
12. $((q,2,2),(q,\lambda))$

Here is the trace of a computation accepting the input string $w = 2+0*1$. After each

computation step we have shown the δ rule invoked.

$$\begin{aligned}
[\,q,\ 2+0*1,\ E\,] &\Longrightarrow [\,q,\ 2+0*1,\ E+E\,] && (1)\\
&\Longrightarrow [\,q,\ 2+0*1,\ 2+E\,] && (6)\\
&\Longrightarrow [\,q,\ +0*1,\ +E\,] && (12)\\
&\Longrightarrow [\,q,\ 0*1,\ E\,] && (7)\\
&\Longrightarrow [\,q,\ 0*1,\ E*E\,] && (2)\\
&\Longrightarrow [\,q,\ 0*1,\ 0*E\,] && (4)\\
&\Longrightarrow [\,q,\ *1,\ *E\,] && (10)\\
&\Longrightarrow [\,q,\ 1,\ E\,] && (8)\\
&\Longrightarrow [\,q,\ 1,\ 1\,] && (5)\\
&\Longrightarrow [\,q,\ \lambda,\ \lambda\,] && (11)
\end{aligned}$$

This *PDA* computation corresponds naturally to the following (leftmost) derivation

$$E \Longrightarrow E+E \Longrightarrow 2+E \Longrightarrow 2+E*E \Longrightarrow 2+0*E \Longrightarrow 2+0*1$$

25.7 Check Your Reading. *Associate each of the derivation steps above with a move taken by the PDA.*

25.8 Example. Returning to the grammar of 25.6, here is the trace of another computation on the same input string $w = 2+0*1$, but this time the *PDA* computation corresponds to the following derivation. (This is also a leftmost derivation, but the grammar is ambiguous).

$$_E \Longrightarrow E*E \Longrightarrow E+E*E \Longrightarrow 2+E*E \Longrightarrow 2+0*E \Longrightarrow 2+0*1$$

$$\begin{aligned}
[\,s,\ 2+0*1,\ E\,] &\Longrightarrow [\,q,\ 2+0*1,\ E*E\,] && ()\\
&\Longrightarrow [\,q,\ 2+0*1,\ E+E*E\,] && ()\\
&\Longrightarrow [\,q,\ 2+0*1,\ 2+E*E\,] && ()\\
&\Longrightarrow [\,q,\ +0*1,\ +E*E\,] && ()\\
&\Longrightarrow [\,q,\ 0*1,\ E*E\,] && ()\\
&\Longrightarrow [\,q,\ 0*1,\ 0*E\,] && ()\\
&\Longrightarrow [\,q,\ *1,\ *E\,] && ()\\
&\Longrightarrow [\,q,\ 1,\ E\,] && ()\\
&\Longrightarrow [\,q,\ 1,\ 1\,] && ()\\
&\Longrightarrow [\,q,\ \lambda,\ \lambda\,] && ()
\end{aligned}$$

You are asked in Problem 236 to associate each of the transitions above with the number of a move taken by the *PDA*. See how the sequence of δ-rules used by the *PDA* is different from Example 25.6. This is a reflection of the ambiguity of the grammar.

25.9 Example. Once again with the grammar of 25.6, here is the trace of a FAILED computation on the same input string $w = 2+0*1$.

$$\begin{aligned} [\,s,\ 2+0*1,\ E\,] &\Longrightarrow [\,q,\ 2+0*1,\ E*E\,] && (2)\\ &\Longrightarrow [\,q,\ 2+0*1,\ 2*E\,] && (6)\\ &\Longrightarrow [\,q,\ +0*1,\ *E\,] && (12)\\ &\text{blocked!} \end{aligned}$$

The δ-moves are all legal. It's just that the *PDA* guessed wrong in the second step, replacing the E on the top of the stack by a "2".

25.3 *PDAs* as Parsers?

In the general case a *PDA* for a grammar G can be considered as providing a parsing algorithm only under a pretty liberal understanding of what a parsing algorithm is! As observed earlier (i) *PDAs* can be very non-deterministic, and (ii) there is no *a prioi* upper bound on the number of steps taken by an arbitrary *PDA* on a given input.

In practice what happens is that we do not try to parse arbitrary grammars, but rather focus on grammars that admit *PDAs* that are "nice," *i.e.*, deterministic. Sometimes we take a given grammar G and concentrate on building another, nicer, grammar G' generating the same language. There is a huge amount of research on this topic, so we will only give a few hints here.

Greibach Normal Form grammars If we are lucky enough to be working with a grammar in Greibach Normal Form, things go very smoothly.

Suppose G is a grammar in Greibach Normal Form and let M be a *PDA* constructed from G as in the proof of Theorem 25.1. For simplicity in the following discussion let's assume that G doesn't have $S \to \lambda$ as a rule.

Recall the moves of the *PDA*:

(i) For each rule $A \to \alpha$ from P, $(\,(q,\bullet,A),\ (q,\alpha)\,)$ is a move in δ.

(ii) For each input symbol $a \in \Sigma$, $(\,(q,a,a),\ (q,\lambda)\,)$ is a move in δ.

We can never make two moves in a row of the first type, since after making one such move the top of the stack will be a terminal symbol (precisely since the grammar is in Greibach Normal Form). So after making a move of type (i) we must immediately make a move of type (ii). And if we are not to fail, the symbol on top of the stack *must* be equal to the current input symbol. This means that:

> If the current input symbol is a and the current stack top is the variable A, then we must choose a type-(i) move corresponding to a rule of the form $A \to aB_1 \dots B_k$.

In a certain sense, then, for a *PDA* working with a Greibach Normal Form grammar, there is no real point to having type-(ii) rules at all. Each type-(i) rule pushes a terminal onto the stack (as part of the $aB_1 \dots B_k$ push) and all the type (ii) rules do is immediately pop the terminal off. We won't make any changes to our official definition of how to make a *PDA* from a *CFG*, but it is this kind of insight that one incorporates into actual application code, when one is writing code inspired by *PDAs*.

25.10 Example. Starting with Example 25.6 we started with an ordinary grammar for arithmetic expressions, built a *PDA*, and did some runs.

If one write a grammar for arithmetic expressions in prefix, one has a Greibach Normal Form. Here is a little grammar to generate prefix arithmetic expressions over $+, *, -$, 0 and 1.

$$E \to +EE \mid *EE \mid -EE \mid 0 \mid 1 \mid 2$$

And indeed this grammar is not only in Greibach Normal Form but it is unambiguous. This will mean, essentially, that parsing cannot go wrong: the *PDA* won't make wrong guesses.

Our automatically-generated *PDA* for this grammar has δ-rules as follows. (Again with numbers attached just so we can refer to them ...)

1. $((q,\bullet,E),(q,+EE))$
2. $((q,\bullet,E),(q,*EE))$
3. $((q,\bullet,E),(q,-EE))$
4. $((q,\bullet,E),(q,0))$
5. $((q,\bullet,E),(q,1))$
6. $((q,\bullet,E),(q,2))$
7. $((q,+,+),(q,\lambda))$
8. $((q,*,*),(q,\lambda))$
9. $((q,-,-),(q,\lambda))$
10. $((q,0,0),(q,\lambda))$
11. $((q,1,1),(q,\lambda))$
12. $((q,2,2),(q,\lambda))$

Here is a successful computation on input $+2*01$. We have annotated each step with

a δ-instruction so you can follow along.

$$\begin{aligned}
[\,q,\ +2*01,\ E\,] &\Longrightarrow [\,q,\ +2*01,\ +EE\,] && (1)\\
&\Longrightarrow [\,q,\ 2*01,\ EE\,] && (7)\\
&\Longrightarrow [\,q,\ 2*01,\ 2E\,] && (6)\\
&\Longrightarrow [\,q,\ *01,\ E\,] && (12)\\
&\Longrightarrow [\,q,\ *01,\ *EE\,] && (2)\\
&\Longrightarrow [\,q,\ 01,\ EE\,] && (8)\\
&\Longrightarrow [\,q,\ 01,\ 0E\,] && (4)\\
&\Longrightarrow [\,q,\ 1,\ E\,] && (10)\\
&\Longrightarrow [\,q,\ 1,\ 1\,] && (5)\\
&\Longrightarrow [\,q,\ \lambda,\ \lambda\,] && (11)
\end{aligned}$$

Here is a unsuccessful computation on input $+*10$.

$$\begin{aligned}
[\,q,\ +10*,\ E\,] &\Longrightarrow [\,q,\ +10*,\ +EE\,] && (1)\\
&\Longrightarrow [\,q,\ 10*,\ EE\,] && (7)\\
&\Longrightarrow [\,q,\ 10*,\ 1E\,] && (5)\\
&\Longrightarrow [\,q,\ 0*,\ E\,] && (11)\\
&\Longrightarrow [\,q,\ 0*,\ 0\,] && (4)\\
&\Longrightarrow [\,q,\ *,\ \lambda\,] && (10)
\end{aligned}$$

blocked, no moves apply

That last input, $+*a0$ is not a legal prefix expression, that is, it is not in the language of the grammar.

But, be careful, we cannot conclude that $+*a0$ is not in the language of the grammar just based on the fact that one computation failed to succeed. For a proof, we'd have to argue that *no* computation could succeed.

Deterministic Grammars

The grammar for Example 25.10 was not only in Greibach Normal Form, but no two rules started with the same terminal. This meant that our *PDA* was "deterministic" in a natural sense.

This suggests identifying the the following nice property in general:

> *Property Q:* For each variable A there are no two rules $A \to aB_1 \ldots B_k$ and $A \to aC_1 \ldots C_p$ whose right-hand sides start with the same terminal.

If a grammar in Greibach Normal Form has this property the choice of successful *PDA* move is completely determined. (Think about that.) So the *PDA* never has to make a non-deterministic choice.

Now it is fair to say that we have something that deserves to be called a parsing algorithm.

Unfortunately it is not the case that every grammar can be put into a Greibach Normal Form which also satisfies Property *Q*. But many can. And for many of those that can't there are slightly weaker properties that we can apply, which guarantee reasonable parsing behavior. This is covered in many textbooks, so we don't describe that work here.

Conclusion

This has been a good case-study in the role of theory in the development of an efficient solution to a practical problem. The essential first step was making the correspondence between grammars and machines: a *PDA* is a very abstract representation of a program for answering parsing questions about a grammar. The code one might generate directly from an arbitrary *PDA* is not what we'd like, involving unbounded searching, backtracking through non-determinism, etc. We refined things *by working on the grammar side.* By using the normal form theorems we were able to optimize our grammars for parsing and then use the *PDA* technology to provide a framework for program code. Grammars, as mathematical objects, are more amenable to provably correct transformations than program code. This is a good general lesson.

25.4 Problems

236. AnnotatePDAComp

Refer to Example 25.8.

Annotate the *PDA* computation given there by associating each of the transitions with the number of a δ move. (The moves were numbered in Example 25.6.)

237. CFGToPDA1

Construct a *PDA M* corresponding to the following grammar, using the construction in Definition 25.2.

$$S \to aSb \mid \lambda$$

Do computations on inputs *aabb*, *aab*, and λ; for each accepting *PDA* computation, give a corresponding grammar derivation.

Hint. Just modify the *PDA* in Example 25.4

238. CFGToPDA2

Construct a *PDA M* corresponding to the following grammar, using the construction in Definition 25.2.

$$S \to aS \mid bP$$
$$P \to aP \mid bS \mid a$$

Do a computation on inputs *aba* and *abb*; for each accepting *PDA* computation, give a corresponding grammar derivation

Note. This grammar has the property that the right-hand side of every rule starts with a different terminal. After you build your *PDA* and start to do some computations, note that this fact allows you to be smart about which *PDA* step to do at each moment, *i.e.* you can avoid any bad guesses. The *PDA* is non-deterministic (because that's the way Definition 25.2 builds *PDAs*) but there is a way to "schedule" the *PDA* deterministically, because the grammar is nice. This is the kind of thing we meant when we said that *PDAs* are abstract representations of parsing algorithms.

239. CFGToPDA3

Construct a *PDA M* corresponding to the following grammar, using the construction in Definition 25.2.

$$E \to E+E \mid F$$
$$F \to F*F \mid 0 \mid 1$$

Do a computation on input $1+0*1$, and give a corresponding grammar derivation.

Does your *PDA* have more than one accepting computation on $1+0*1$?

Compare to Example 25.6

240. CFGToPDA4

Construct a *PDA M* corresponding to the following grammar, using the construction in Definition 25.2.

Do computations over $*+101$, $*a+b$, and $+1*01$; for each accepting *PDA* computation, give a corresponding grammar derivation.

$$E \to +EE \mid *EE \mid 0 \mid 1$$

Note. Compare this grammar to the ambiguous one $E \to E+E \mid E*E \mid 0 \mid 1$ As in the previous problem, the fact that in G_4 the right-hand sides of the rules start with different terminals means that each string has at most one successful run; this translates into each string having at most one parse tree. That is, the grammar is unambiguous in addition to being in Greibach Normal Form.

241. CFGToPDA5

Let G be the following grammar.

$$S \to aS \mid bT \mid b$$
$$T \to aS \mid bT \mid b$$

a) What language does G generate?

b) Write down the rules of the *PDA* for G as given by Definition 25.2

c) Show an accepting computation on input *bab*.

242. CFGToPDA6

Let G be the following grammar.

$$\begin{aligned} S &\to aB \mid bA \\ A &\to aS \mid bAA \mid a \\ B &\to bS \mid aBB \mid b \end{aligned}$$

a) What language does G generate?

b) Write down the rules of the *PDA* for G constructed by Definition 25.2

c) Show an accepting computation on input *babbaa*.

d) Show an failing computation on input *babbaa*. That is, show an example where the *PDA* makes a wrong guess, in a situation where it has a choice of move...see what can happen.

243. CFGNotDouble

We have mentioned that the following language

$$D = \{w \in \{a,b\}^* \mid w \text{ can be written as } xx \text{ for some string } x\}$$

is not context-free.

By contrast, $\overline{D}$, the complement of D, is context-free. Prove this.

Hint. The key insight is that $\overline{D}$ can be written as

$$\begin{aligned} &\{w \mid \text{ the length of } w \text{ is odd}\} \\ \cup &\{xax'yby' \mid x,y,x',y' \in \Sigma^*,\ |x| = |x'|,\ |y| = |y'|\} \\ \cup &\{xbx'yay' \mid x,y,x',y' \in \Sigma^*,\ |x| = |x'|,\ |y| = |y'|\} \end{aligned}$$

First, argue that this characterization of $\overline{D}$ is correct!

Second, argue that $\{w \mid \text{ the length of } w \text{ is odd}\}$ is context-free (easy).

Finally, since the *CFLs* are closed under union, it suffices to argue that

$$\begin{aligned} &\{xax'yby' \mid x,y,x',y' \in \Sigma^*,\ |x| = |x'|,\ |y| = |y'|\} \\ \cup &\{xbx'yay' \mid x,y,x',y' \in \Sigma^*,\ |x| = |x'|,\ |y| = |y'|\} \end{aligned}$$

is context-free. Outline a *PDA*. (It would be pretty tedious to give all the details of this *PDA*; give a careful English description of how it would work.)

244. ABCCompl

Let L be

$$\{a^i b^i c^i \mid i \geq 0\}$$

Show that $\overline{L}$ is context-free.

Hint. Use a method similar to Problem 243: present the language as a certain union, and show that the pieces are each context-free.

Chapter 26

Proving Languages Not Context-Free

We observed, by a cardinality argument, that there must be some languages that are not context-free. But that's not very satisfying; it doesn't help us understand whether any *given* language is context-free. In this section we will learn a technique for showing languages to be non-context-free.

Before we reason about grammars at all, here is a purely combinatorial lemma.

26.1 Lemma. *Suppose T is a tree such that every interior node has at most k children. For any number $n \geq 0$: if the number of leaves of T is greater than k^n then there is a path in T with more than n edges.*

Proof. It is more convenient to prove the contrapositive. Namely:

> Suppose T is a tree such that every interior node has at most k children. For any number $n \geq 0$: if every path of T has no more than n edges then the number of leaves of T is no more than than k^n.

We prove this by induction on n. For $n = 0$ the assumption is that every path has no edges; this implies that T is a single node, and so here the number of leaves is 1, which is indeed no more than k^0. When $n > 0$ (and T not the single-node tree) consider the tree T' obtained from T by removing all the leaves of T. So every path of T' has no more than $(n-1)$ edges. By induction hypothesis, then, T' has no more than $k^{(n-1)}$ leaves. But T can be built from T' by adding back the children of the nodes which are leaves of T': since each of these nodes has at most k children the number of leaves of T is no more that $kk^{(n-1)}$, that is, k^n. ///

Applying this result to parse trees we conclude the following.

26.2 Theorem. *Suppose G is a context-free grammar. Suppose that G has n variables and is such that every rule has at most k symbols on its right-hand side. Then for any string $w \in L(G)$ of length greater than k^n, any parse tree for w has a path in T with a repeated variable occurrence.*

Proof. Consider any parse tree T for w. Since T has $|w|$ leaves, Lemma 26.1 tell us that T has a path π with more than n edges. So this π has more than $n+1$ nodes. Since only the leaf of π is a terminal symbol, π has more than n variable nodes. Since the grammar has only n variable symbols, π has a repeated occurrence. ///

26.1 A Language Which is Not Context-Free

First we do a concrete example, then show the general result. The work we do for the particular example is almost all we need to prove the general Pumping Lemma, so once you understand this first section you should be able to proceed to the general result if you care to.

Let's show that the following language

$$L = \{a^n b^n c^n \mid n > 0\}$$

is not context-free.

26.3 Theorem. *The language*

$$L = \{a^n b^n c^n \mid n > 0\}$$

is not a context-free language.

Proof. For the sake of contradiction, suppose G is a CFG supposedly generating L. Without loss of generality we may assume that G has no λ- or chain-rules (since we could eliminate them if need be). Let n be the number of variable symbols in G, let k be the maximum length of any right-hand side of a rule from G, then let p be k^n.

Let z be the string $a^p b^p c^p$. Note that the length of z is greater than k^n, so Theorem 26.2 applies.

Now consider a leftmost derivation of the string z.

Any such derivation has a repeated variable in some path of its parse tree. Focus on the *next-to-last* occurrence of a symbol, call it A, in that path. We have

$$\begin{aligned} S &\stackrel{*}{\Longrightarrow} u\, A\, \sigma \\ &\stackrel{*}{\Longrightarrow} uz_1 \equiv z \end{aligned}$$

Here u is a terminal string and σ is a mix of terminals and variables, and ew have $A\sigma \stackrel{*}{\Longrightarrow} z_1$.

Since that A is repeated on the path we are thinking about, our leftmost derivation must look like

$$\begin{aligned} S &\stackrel{*}{\Longrightarrow} u\, A\, \sigma \\ &\stackrel{*}{\Longrightarrow} u\, v\, A\, \alpha\, \sigma && \text{via } A \stackrel{*}{\Longrightarrow} v\, A\alpha \\ &\stackrel{*}{\Longrightarrow} u\, v\, w\, \alpha\, \sigma && \text{via } A \stackrel{*}{\Longrightarrow} w \\ &\stackrel{*}{\Longrightarrow} u\, v\, w\, x\, \sigma && \text{via } \alpha \stackrel{*}{\Longrightarrow} x \\ &\stackrel{*}{\Longrightarrow} u\, v\, w\, x\, y && \text{via } \sigma \stackrel{*}{\Longrightarrow} y \\ &\equiv z \end{aligned}$$

Now observe that the following, completely different, derivation is also a legal derivation in G

$$\begin{aligned} S &\stackrel{*}{\Longrightarrow} u\, A\, \sigma \\ &\stackrel{*}{\Longrightarrow} u\, v\, A\, \alpha\, \sigma && \text{via } A \stackrel{*}{\Longrightarrow} v\, A\, \alpha \\ &\stackrel{*}{\Longrightarrow} u\, v\, v\, A\, \alpha\, \alpha\, \sigma && \text{via } A \stackrel{*}{\Longrightarrow} v\, A\, \alpha \\ &\stackrel{*}{\Longrightarrow} u\, v\, v\, w\, \alpha\, \alpha\, \sigma && \text{via } A \stackrel{*}{\Longrightarrow} w \\ &\stackrel{*}{\Longrightarrow} u\, v\, v\, w\, x\, x\, \sigma && \text{via } \alpha \stackrel{*}{\Longrightarrow} x \\ &\stackrel{*}{\Longrightarrow} u\, v\, v\, w\, x\, x\, y && \text{via } \sigma \stackrel{*}{\Longrightarrow} y \end{aligned}$$

So the string $uvvwxxy$ is also derivable in our grammar. We will have our desired contradiction if we can show the following

Claim. The string $uvvwxxy$ is not of the form $a^k b^k c^k$ for any k.

Proof of claim. First, an important detail: we claim that at least one of v or x is a non-empty string. To see this, note that since $A \stackrel{*}{\Longrightarrow} vA\alpha$ and $\alpha \stackrel{*}{\Longrightarrow} x$ above, we know that $A \stackrel{*}{\Longrightarrow} vAx$ is a possible derivation in our grammar. Since our grammar has no λ or chain-rules, this implies that at least one of v or x is non-empty.

To complete the argument that $uvvwxxy$ cannot look like $a^k b^k c^k$ we examine cases as to what alphabet symbols appear in v and in x.

If either v or x has more than one kind of letter occurring then $uvvwxxy$ won't even be in the form $a^*b^*c^*$. So we can assume that each of v and x has only one kind of letter.

But now note that in comparing $uvvwxxy$ with the original $uvwxy$ we have increased the number of occurrences of one or two kinds of letters, certainly not all three. So regardless of the ordering of letters $uvvwxxy$ cannot have the same number of each kind of letter.

That finishes the proof of the claim. Since the claim contradicts the fact that G derives only strings in L, this finishes the proof of the Theorem. ///

26.2 A Pumping Lemma for Context-Free Grammars

We can prove a general Pumping Lemma for context-free languages. It uses the same ideas as the special case we just proved, and we have intentionally used the same phrasing as much as possible below, to emphasize that fact.

26.4 Lemma. *Let G be a context-free grammar with no chain or λ rules. Let n be the number of variables of G and let k be the maximum size of a right-hand side of a rule of G. If z is a word of length greater than k^n derivable from G then there is a derivation of z in G of the following form (for some variable A):*

$$\begin{aligned} S &\stackrel{*}{\Longrightarrow} uAy \\ &\stackrel{*}{\Longrightarrow} uvAxy \qquad \text{// via} \quad A \stackrel{*}{\Longrightarrow} vAx \\ &\stackrel{*}{\Longrightarrow} uvwxy \qquad \text{// via} \quad A \stackrel{*}{\Longrightarrow} w \\ &= z \end{aligned}$$

Furthermore

1. *the length of vwx is no more than k^n*
2. *At least one of v or x is not empty*

Proof. Choose some parse tree for z. This tree has a repeated variable in some path. Focus on the *next-to-last* occurrence of a symbol, call it A, in that path. We can arrange the steps in a derivation of z so that

$$\begin{aligned} S &\stackrel{*}{\Longrightarrow} u\,A\,\sigma \\ &\stackrel{*}{\Longrightarrow} uz_1 \equiv z \end{aligned}$$

Here u is a terminal string and σ is a mix of terminals and variables, and ew have $A\sigma \stackrel{*}{\Longrightarrow} z_1$.

Since that A is repeated on the path we are thinking about below the indicated A, our leftmost derivation must, in more detail, look like

$$\begin{aligned}
S &\stackrel{*}{\Longrightarrow} u\,A\,\sigma \\
&\stackrel{*}{\Longrightarrow} u\,v\,A\,\alpha\,\sigma && \text{via } A \stackrel{*}{\Longrightarrow} v\,A\alpha \\
&\stackrel{*}{\Longrightarrow} u\,v\,w\,\alpha\,\sigma && \text{via } A \stackrel{*}{\Longrightarrow} w \\
&\stackrel{*}{\Longrightarrow} u\,v\,w\,x\,\sigma && \text{via } \alpha \stackrel{*}{\Longrightarrow} x \\
&\stackrel{*}{\Longrightarrow} u\,v\,w\,x\,y && \text{via } \sigma \stackrel{*}{\Longrightarrow} y \\
&\equiv z
\end{aligned}$$

Now observe that the following, completely different, derivation is also a legal derivation in G

$$\begin{aligned}
S &\stackrel{*}{\Longrightarrow} u\,A\,\sigma \\
&\stackrel{*}{\Longrightarrow} u\,v\,A\,\alpha\,\sigma && \text{via } A \stackrel{*}{\Longrightarrow} v\,A\,\alpha \\
&\stackrel{*}{\Longrightarrow} u\,v\,v\,A\,\alpha\,\alpha\,\sigma && \text{via } A \stackrel{*}{\Longrightarrow} v\,A\,\alpha \\
&\stackrel{*}{\Longrightarrow} u\,v\,v\,w\,\alpha\,\alpha\,\sigma && \text{via } A \stackrel{*}{\Longrightarrow} w \\
&\stackrel{*}{\Longrightarrow} u\,v\,v\,w\,x\,x\,\sigma && \text{via } \alpha \stackrel{*}{\Longrightarrow} x \\
&\stackrel{*}{\Longrightarrow} u\,v\,v\,w\,x\,x\,y && \text{via } \sigma \stackrel{*}{\Longrightarrow} y
\end{aligned}$$

So the word $uvvwxxy$ is also derivable in our grammar. We will have our desired contradiction if we can show the following

Claim. The word $uvvwxxy$ is not of the form $a^k b^k c^k$ for any k.

Proof of claim. First, an important detail: we claim that at least one of v or x is a non-empty string. To see this, note that since $A \stackrel{*}{\Longrightarrow} vA\alpha$ and $\alpha \stackrel{*}{\Longrightarrow} x$ above, we know that $A \stackrel{*}{\Longrightarrow} vAx$ is a possible derivation in our grammar. Since our grammar has no λ or chain-rules, this implies that at least one of v or x is non-empty.

To complete the argument that $uvvwxxy$ cannot look like $a^k b^k c^k$ we examine cases as to what alphabet symbols appear in v and in x.

If either v or x has more than one kind of letter occurring then $uvvwxxy$ won't even be in the form $a^* b^* c^*$. So we can assume that each of v and x has only one kind of letter.

But now note that in comparing $uvvwxxy$ with the original $uvwxy$ we have increased the number of occurrences of one or two kinds of letters, certainly not all three. So regardless of the ordering of letters $uvvwxxy$ cannot have the same number of each kind of letter.

That finishes the proof of the claim. Since the claim contradicts the fact that G derives only words in L, this finishes the proof of the Theorem. ///

Notes.

1. Of course we could just as well have argued that uv^iwx^iy must be derivable from G, for *each* $i \geq 0$.

2. In the above argument we didn't actually use the fact that we started with the next-to-last repeated symbol in our path, only that there was *some* repetition below.

   Think about the following:

   (a) The fact that there are no other repetitions below our A tells us something about how long the string vwx can be, in light of Lemma 26.1. What is the maximum possible size of vwx?

   (b) Having that bound is sometimes useful in proving other languages to be non-context-free: the ability to restrict the "span" of vwx in the original word is sometimes essential in arguing that some pumped version uv^iwx^iy is not in a language.

26.5 Corollary. *Let G be a context-free grammar with no chain or λ rules. Let n be the number of variables of G and let k be the maximum size of a right-hand side of a rule of G. If z is a word of length greater than k^n derivable from G then z can be written as the concatenation*

$$z = uvwxy$$

such that

1. *the length of vwx is no more than k^n*
2. *At least one of v or x is not empty*
3. *For each $i \geq 0$, the word uv^iwx^iy is derivable from G.*

We can use Corollary 26.5 to show languages not context-free.

26.6 Example. The language $L = \{a^nb^nc^n \mid n \geq 0\}$ is not context-free.

Proof. For sake of contradiction, suppose L were context-free. Let p be the number given in Corollary 26.5. Now let z be the word $a^pb^pc^p$. Consider the words u, v, w, x, y given by Corollary 26.5 , and define z' to be the word uwy. Since z' is uv^0wx^0y, Corollary 26.5 says it must be in L. We will get our contradiction by arguing that it is *not* in fact in L.

First notice that the word vwx cannot contain both as and cs, by virtue of the fact that $|vwx| \leq p$. So vwx consists only of as and bs, or it consists only of bs and cs.

We consider those two cases

- If vwx consists only of as and bs, $z' = uwy$ cannot have p occurrences of both a and b (since we eliminated v and x from z). But z' has p occurrences of c (since v and x don't have any cs). So the number of as, bs, and cs in z' don't match, so z' can't be in L.
- If vwx consists only of bs and cs, the argument is completely symmetric. (It is a good exercise for you to write out the argument).

///

26.3 Problems

245. CFLINtersect

Show that the context-free languages are *not* closed under intersection.

246. CFLComplement

Show that the context-free languages are *not* closed under complement.

247. CFLSubset

1. Prove or disprove: If L is context-free and $K \subseteq L$ then K is context-free.
2. Prove or disprove: If L is context-free and $L \subseteq K$ then K is context-free.

248. CFLPumping

Prove that

$$L = \{a^n b^n c^i \mid i \leq n\}$$

is not context-free.

249. CFLMorePumping

Show that each of the following languages is not context-free.

1. $\{a^n b^m \mid n^2 \geq m\}$
2. $\{a^n b^m \mid n^2 \leq m\}$
3. $\{a^n b^m \mid n \geq m^2\}$
4. $\{a^n b^m \mid n \leq m^2\}$

250. CFLTaxonomy

For each of the following languages, tell whether it is

- regular
- context-free, but not regular
- not context-free

In each case, prove your answer.

What does this mean? To prove a language regular you can exhibit a finite automaton (any flavor) or a regular expression, possibly in conjunction with using some of the known closure properties. To prove a language context-free you can exhibit a PDA or grammar, possibly in conjunction with using some of the known closure properties. To prove a language *not* regular or context-free, you can use a pumping lemma argument.

1. $\{w \in \{a,b,c\}^* \mid w$ has an equal number of as, bs and cs$\}$
2. $\{a^n \mid n$ is a power of 2$\}$
3. $\{w \in \{0,1\}^* \mid w$ represents a power of 2 in binary$\}$
4. $\{a^n b^m \mid n = m\}$
5. $\{a^n b^m \mid n \neq m\}$
6. $\{a^n b^m \mid n \leq m\}$
7. $\{a^n b^m c^k d^l \mid n = m$ or $k = l\}$
8. $\{a^n b^m c^k d^l \mid n = m$ and $k = l\}$
9. $\{a^n b^m c^k d^l \mid n = k$ or $m = l\}$
10. $\{a^n b^m c^k d^l \mid n = k$ and $m = l\}$
11. $\{a^n b^m c^k d^l \mid n > k$ or $m > l\}$
12. $\{a^n b^m c^k d^l \mid n > k$ and $m > l\}$
13. The set of all strings w over $\{a,b\}$ satisfying: w has an equal number of as and bs and each prefix of w has at most 1 more a than b and at most 1 more b than a.

251. ABCLessThan

Prove that

$$L = \{a^i b^j c^k \mid i < j < k\}$$

is not context-free.

252. CFL999

1. Show the following to be a regular language: $\{a^n b^n c^n \mid n \leq 999\}$
2. Show the following to be a non-context-free language: $\{a^n b^n c^n \mid n > 999\}$
   Hint: Use the fact that the context-free languages are closed under union.

253. CFLDouble

Assume the following fact: Over the alphabet $\Sigma = \{a, b\}$, the language $D = \{w \mid w \text{ can be written as } xx \text{ for some string } x\}$ is not context-free.

Prove that the language $C = \{a^n b^m a^n b^m \mid n, m \geq 0\}$ is not context-free.

254. CFLMorphism

1. Let L be a language over an alphabet Σ. Let a and b be elements of Σ and define the function h from Σ^* to Σ^* by: $h(x)$ = the result of replacing all occurrences of a in x by b. Having defined h, let $h(L)$ be the language obtained by applying h to each member of L, that is, $h(L) = \{h(x) \mid x \in L\}$.

   Prove that if L is context-free then $h(L)$ is context-free. (*Hint:* consider a CFG G such that $L(G) = L$; show how to build a CFG G' such that $L(G') = h(L)$.

2. More generally, suppose that h is *any* function mapping elements of Σ to elements of Σ. Extend h to strings: $h : \Sigma^* \to \Sigma^*$ is defined by $h(x)$ = the result of replacing all occurrences of each symbol $c \in \Sigma$ by the symbol $h(c)$. Finally, extend h to languages by: $h(L) = \{h(x) \mid x \in L\}$.

   Prove that if L is context-free then $h(L)$ is context-free. (*Hint:* consider a CFG G such that $L(G) = L$; show how to build a CFG G' such that $L(G') = h(L)$.

   [This phenomenon can be pushed even further, to consider more general mappings from strings into strings, called "homomorphisms," which preserve context-free-ness...]

Part IV

Decidable and Undecidable Languages

Chapter 27

Introduction to Computabilty

Our goal is to study the most fundamental question about computing:

> *what problems can be solved by algorithms?*

Of course ill-defined problems concerning things like politics, the meaning of life, the best flavor of ice cream, *etc.* can't be "solved" by algorithms. Such problems are both ill-defined and don't have objectively-correct answers.

In contrast, an instance of a "problem" in this chapter will be a mathematically precise question with a yes/no answer, such as "is this integer a prime number?" or "does this C program attempt to do division by zero on any inputs?"

The really interesting fact is that lots of mathematically precise yes/no problems cannot be solved by algorithms, and we can give concrete examples of such.

To actually prove such claims we need to decide at the very beginning exactly what we mean by "algorithm." There are lots of definitions that have been offered over the years. The most famous formalism is Turing machines. The most convenient formalism is ordinary programs. As models of what can be computed these are mathematically equivalent. The amazing fact is that every computability formalism that has been proposed over the past century has been mathematically proven to be equivalent to all the others.

In Chapter 31 we introduce Turing machines and explain how they serve as a formalism for defining arbitrary computation. In Chapter 32 we show how to use ordinary programs as an alternative formalism, and explain the sense in which these Turing machines and programs are equivalent. Having done that we will be able to use whatever formalism is most convenient at any given time.

But before we start all that, we begin, in Chapter 28, with some concrete examples designed to sharpen your intuition about why reasoning about programs is hard.

Then in Chapter 29 we prove rigorously that a certain problem involving reasoning about programs is **impossible** to solve algorithmically. Then we will be motivated for all the rest of the technical work about what can and can't be accomplished algorithmically.

Chapter 28

Warm Up: Reasoning About Programs

In this part of the course we are exploring the decidability of problems in a wide range of areas:

- problems about *grammars*,
- problems about *puzzles*,
- problems about *polynomials*,
- problems about *logic*,
- . . . but most significantly if you are a computer scientist: problems about *programs*.

This chapter is an introduction to the last one: reasoning about programs. We won't present any actual results, we just want to get you started thinking.

28.1 Reasoning About Programs Generally

A basic problem in computer science is: here is a program, does it do what I want? A more nuanced way to say that is: here is a program, here is a claim about the program, is that claim true?

Some examples of such claims:

- does it sort integer lists correctly?
- does it correctly decide whether a input number is prime?
- does it ever do a division by 0?
- does it ever go into an infinite loop?
- does it answer database queries correctly?
- is there any dead code?
- does it allow read access to the system password file?
- does it ever go into an infinite loop?

These are all instances of the following pattern:

> *Fix some claim $\mathcal{R}$ about program behavior. Here is a program p; is the claim $\mathcal{R}$ true about p?*

You do this all the time, every time you write some code p and then try to understand it.

What if we are more ambitious? Namely, for a given claim $\mathcal{R}$, what if we try to write a systematic check that will evaluate the claim *for any* program?

For example we might ask

- can we write an automatic procedure that will decide—given any program p as input—whether p sorts integer lists correctly?
- can we write an automatic procedure that will decide—given any program p as input—whether p correctly tests for primality?
- can we write an automatic procedure that will decide—given any program p as input—whether p ever does a division by 0?
- and so on . . .

In fact we will mostly focus on one particular claim, seemingly a bit easier than the ones above: does a given program eventually halt on a given input? This problem may seem artificial. But we will see as we go on that it is at the heart of all reasoning about program. Have faith.

28.2 Halting

In the two examples below we try to answer the question: which inputs will cause the program to eventually halt.

These should be read as C programs, but for legibility below we've taken slight liberties with program syntax. For example instead of `cin >> x` we just write `read x`.

28.1 Example.

```
void main() {
    int x,y;
    read x;
    read y;
    while (x >= 0) {
      if (y > 0) {
         y--;


      }
      else {
        x--;
        read y;
       }}}
```

Answer: This always halts. Starting with initial `(x,y)` values (n,m), we eventually get to (n, 0); then to (n-1, some new m'); etc, so that eventually n gets to be 0.

28.2 Example.

```
void main() {
    int x,y;
    read x;
    read y;
    while (x >= 0) {
      if (y > 0) {
         y++;
      }
      else {
        x--;
        y = y / x ;
       }}}
```

Answer: This only halts when the initial x is non-negative and the initial y is non-positive.

28.2.1 Not Just Halting

Maybe it seems boring to just ask when programs halt. We'll see later that halting really is fundamental. As a hint towards why that might be true, consider the following question about Example 28.2: *does this program ever do a division by zero? If so, under what circumstances?* If you think about this you should see that you have to think about the same kinds of issues as you do in thinking about halting.

28.2.2 You Have to Decide! (Very Important)

This is a good time to bring up a crucial point. Suppose you are given a concrete program p. Now suppose you want to decide whether p wilt halt on input 17. Obviously one thing you can do is just run the program on 17. If it halts, you're done: answer yes. But suppose it runs for a year or so with no answer? Maybe it will halt in the next 5 minutes. Do you wait 100 years? Is there ever a time when you can definitively say: "nope, this program will not halt on 17"?

Now, maybe you can arrive at this answer "no halt" answer by doing some analyzing of the program text. You just did something like that in the two examples above. This chapter is all about doing such program *analysis*, as opposed to testing. But can't ever be sure of a "no halt" answer *just* by running the program.

Be very sure this point is clear to you. It's the #1 source of confusion for students studying this material, trust me!

28.3 More Halting Examples

We collect more examples as puzzles, used as problems for the end of the chapter. For each program, try to answer: which inputs will cause the program to eventually halt?

28.3 Example.

```
void main() {
    int x,y;
    read x;
    read y;
    while (x >= 0) {
      if (y > 0) {
         x--;
      }
      else {
        x--;
```

```
        read y;
       }}}
```

28.4 Example.

```
void main() {
    int x, y;
        read x;
        read y;
        while (x >= 0) {
                x = x + y;
                y = y - 1;
        }}
```

28.5 Example.

```
void main() {
    int x;
    int y;
    read x;
    read y;
    while (x >= 0) {
         x = x + y;
         y = y + 1;
        }}
```

Example due to: Ton Chanh Le (chanhle@comp.nus.edu.sg). Taken from the Termination Problem Database at https://github.com/TermCOMP/TPDB

28.6 Example.

```
void main() {
    int x;
    int y;
    read x;
    read y;
    if ( y <= x) {
       while (x >= 0) {
         x = x - y;
        }}
```

Example due to: Ton Chanh Le (chanhle@comp.nus.edu.sg). Taken from the Termination Problem Database at https://github.com/TermCOMP/TPDB

28.7 Example.

```
void main() {
    int x, y, n;
        read n;
        read x;
        read y;
        while (x >= 0) {
                while (y >= 0) {
                        y = y - 1;
        }
                x = x - 1;
                while (y <= n) {
                        y = y + 1;
        }}}
```

Adapted from an example due to: Caterina Urban. Taken from the Termination Problem Database at https://github.com/TermCOMP/TPDB

28.8 Example.

```
void main() {
    int x;
    read x;
    while (1 < x) {
      if (x is even)
          x = x div 2
      else
          x = 3*x + 1
}}
```

Run this example by hand on a few inputs. Look at what happens!

This is a famous problem, see for example https://en.wikipedia.org/wiki/Collatz_conjecture

We hope to have persuaded you that **reasoning about programs is hard.** In future chapters we will demonstrate that in a certain sense, it is impossible!

28.4 Problems

255. HaltPredicting

a) For which x and y does Example 28.3 halt?

b) For which x and y does Example 28.4 halt?

c) For which x and y does Example 28.5 halt?

d) For which x and y does Example 28.6 halt?

e) For which x, y and n does Example 28.7 halt?

f) For which x does Example 28.8 halt?

Hint. If you solve this you get an A in the class. If you solve it 10 years from now I will go back and change your grade to A, if necessary.

Chapter 29

The Halting Problem

In Chapter 28 you looked at examples of specific programs and inputs and tried to predict halting. Here we raise the stakes, by asking whether there can be a single algorithm that *uniformly* answers the problem of halting, that is to say, whether there can be a single algorithm that can process *any* program p and *any* input x and predicts whether p will halt on x.

For concreteness, when we say "program" in this section we mean "C program".

Remember that a program is just a text file. So it makes perfect sense to run a program on another program. Some examples:

- A compiler p takes programs q as input, to translate q to lower-level code;
- A dead-code analyzer p takes a program q as input and tries to optimize q by identifying functions in q that will never be called;
- At a very basic level, a program p that simply counts the number of symbols in a file can certainly take a program as input: it simply says how long the program q is;
- In fact any program p at all that reads from standard input can be run with a program q as input.

The problem we want to look at now is the problem of predicting what will happen when one runs a program p on an input. In fact we'll focus on a very simple question: will the program eventually halt, or will it go into an infinite loop? [1]

The Halting Problem

[1] If the program p has certain expectations about the format of its input, which may make it "refuse" to accept itself as proper input, then such immediate termination counts as halting.

INPUT: a C program p and a bitstring x

QUESTION: does p halt when given x as input?

Is there an algorithm to solve this problem? We are going to show that there isn't. In fact we are going to show that even if we ask a very specific instance of this problem, there is no algorithm (and so there cannot be an algorithm for the more general problem either):

The Self-Halting Problem

INPUT: a C program p

QUESTION: does p halt when given itself as input?

This is an admittedly artificial, even bizarre, problem to try to solve. We use it because it is the easiest first example to work with; it is a stepping-stone to looking at more realistic problems (soon).

Now, there is an obvious first thought we have about answering the Self-Halting Problem by a program: we can write a program h that (i) reads its input p, (ii) makes a copy of p, (iii) then, just like an operating system does, runs p with input p, (iv) if this simulation halts, answer "yes".

This is a coherent thing to do, but such an h isn't a solution to the decision problem, because it never returns "no". If in fact the original input p is a program that does not halt on itself, then the *simulation* in part (iii) above will run forever, and there is never a time point at which we can observe that the computation is going to run forever.

OK, we have seen this situation before, a decision problem where there is a too-naive attempt at a solution, but with cleverness we can find a real solution. Think about the *DFA* Emptiness problem, or the *CFG* Membership problem: each of these invited a (wrong) naive solution but ultimately submitted to a clever solution that avoided infinite search.

The Self-Halting Problem is different. No matter how clever we are, we can't find an answer.

29.1 Theorem. *There is no C program that solves the Self-Halting Problem.*

Proof. We assume that there is a C program deciding the Self-Halting Problem and obtain a contradiction. So **suppose** that `selfHaltTest` is a program taking one input p with

$$\texttt{selfHaltTest}(p) = \begin{cases} \text{returns } 1 & \text{if } p \text{ halts on } p \\ \text{returns } 0 & \text{if } p \text{ fails to halt on } p \end{cases}$$

We can then write the following code.

```
int main (p) {
       if (selfHaltTest(p))   /* if p halts on p         */
         while (1) do ;       /* then loop forever       */
       else                   /* if p doesn't halt on p  */
         return(1);           /* then return 1           */
        }
```

So what is the behavior of `main` of some input p? Just by looking at the code, we have

$$\texttt{main}(p) = \begin{cases} \text{loops forever} & \text{if } p \text{ halts on } p \\ \text{returns 1} & \text{if } p \text{ fails to halt on } p \end{cases}$$

So consider: what is the result of the execution `main[main]`?

$$\texttt{main (main)} = \begin{cases} \text{loops forever} & \text{if main halts on main} \\ \text{returns 1} & \text{if main fails to halt on main} \end{cases}$$

This is clearly absurd.

We conclude that if `selfHaltTest` does what we claim it does, we always get a contradiction. Thus there can be no procedure `selfHaltTest` that performs as we claimed originally. That's the end of the proof. ///

29.2 Theorem. *There is no C program that solves the Halting Problem.*

Proof. Obviously if we had a C program that solved the full Halting Problem, a trivial modification of it would solve the Self-Halting Problem; an impossibility. ///

29.1 The Halting Problem in Pictures

Here is the same proof, presented visually. This presentation shows a subtle relationship with the diagonalization argument used to prove certain sets uncountable. (Compare the discussion below to Section 2.4.1.) We stress that we are showing **the same proof** as above: the only difference is that we don't give C code here, but we draw pictures.

Since program is just a text file, it is a string. We know how to order the Σ_2 strings, so we can order the strings that are programs, as $p_0, p_1, \ldots$, each string occurring as one of the p_i.

So now imagine a table, extending infinitely down and to the right, where — intuitively — the rows are indexed by the programs and the columns are indexed by the programs as input strings.

	p_0	p_1	p_2	p_3	p_4	$\cdots$	$\cdots$
p_0							
p_1							
p_2							
p_3							
p_4							
$\vdots$							
$\vdots$							

Now for a given row r, that is, the row corresponding to program p_r, we can imagine placing a $\downarrow$- mark in column i precisely when program p_r halts on p_i (as an input string).

	p_0	p_1	p_2	p_3	p_4	$\cdots$		$\cdots$	
p_0	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$		$\downarrow$	
$\vdots$									
p_{2869}	$\downarrow$	$\uparrow$	$\downarrow$	$\uparrow$	$\downarrow$	$\uparrow$	$\downarrow$	$\cdots$	
$\vdots$									
$p_{5000001}$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\cdots$	
$\vdots$									

The picture above suggests the situation where

- program number 0 halts on all inputs;
- program number 2869 halts on precisely the strings with even indices;
- program number 5000001 halts on no inputs; $\ldots$

Then the `main` program above is the program that on any input p, goes down the diagonal to the (p,p) entry, and there is a $\downarrow$ there, goes into an infinite loop, and if there is $\uparrow$ there, returns 1 (and halts). That is, `main` inverts the diagonal. The argument that starts with "So consider: $\ldots$" is precisely the argument that this inverted diagonal cannot correspond to any of the rows. This means that the behavior of `main` cannot be the behavior of any of the p_i programs. But we apparently gave legal code for `main`!

So where is the contradiction? It lies in the fact that the $\downarrow$ and $\uparrow$ symbols on the diagonal cannot be computed by a *program* `selfHaltTest`. Then the code for `main` isn't really code at all, since it is calling for a helper function that knows how to tell it whether there is a $\downarrow$ or a $\uparrow$ in a given place.

So the argument is actually even more subtle than the diagonalization arguments we did earlier, about things not being countable. In those uncountability arguments we started with a listing of things, and constructed a new thing (inverting the diagonal) to show that we didn't start with a complete listing after all. But in *this* argument everything is countable, there's not dispute about that. We can be sure that we have a complete, list of all programs as rows. What we show is that the diagonalization trick *cannot actually be done by a program* since the result would be a program not in the list.

As a final remark let us stress that the table of $\downarrow$ and $\uparrow$ symbols is perfectly sensible mathematically. There is no funny business here: program p_r on input p_i really does either halt or not. What our proof shows is just that **there is no *program* that can fill in that table.** This is what the undecidability of the Halting Problem says, in a picture.

To go further in our study of decidability we will need some careful definitions and preliminary groundwork. We start that business in the next chapter.

29.2 Perspective

It is crucial to understand the consequences of results like these for reasoning about programs. Suppose your job is to write a tool that—perhaps among other things—determines whether a given program will give an answer on a given input.

(If you object that this doesn't sounds like the kind of job that arises in practice much, be patient. We are just getting started. We will tackle many other program-analysis tasks as we go. See Chapter 34!)

The fallout from the undecidability of the Halting Problems is that your tool **must**, for some inputs p and x,

- either say that p halts on x even though it really doesn't,
- or fail to recognize that p halts on x even though it does.

This kind of choice between under-approximation and over-approximation is inherent in the field of program analysis.

The Halting Problem, and the proof that there is no algorithm to solve it, can stand alone independently of the formalisms of this course: strings, languages, formal machine models, etc. Indeed we presented it that way in this chapter to emphasize that the result stands above arcane definitions and notations.

29.3 Problems

256. HaltVariation1

Consider the following proposed variation on the proof of the undecidability of the Halting Problem, in which we avoid the use of an induced loop and simply "swap answers" in the `main` program.

```
int main1 (p)
{
   if (selfHaltTest(p))   /* if p halts on p         */
        return(0);        /* then return 0           */
    else                  /* if p doesn't halt on p  */
        return(1);        /* then return 1           */
}
```

Will this support a proof-by-contradiction as we did earlier? That is, can we get a contradiction by assuming that `selfHaltTest` behaves as claimed, and then reasoning about this `main1` program in exactly the same way as we did for `main`, namely by examining `main1(main1)`?

257. HaltVariation2

Consider the following proposed variation on the proof of the undecidability of the Halting Problem.

```
int main2 (p) {
       if (selfHaltTest(p))   /* if p halts on p         */
         while (1) do ;       /* then loop forever       */
       else                   /* if p doesn't halt on p  */
         return(0);           /* then return 0          */
        }
```

Will this support a proof-by-contradiction as we did earlier? That is, can we get a contradiction by assuming that `selfHaltTest` behaves as claimed, and then reasoning about this `main2` program in exactly the same way as we did for `main`, namely by examining `main2(main2)`?

258. HaltVariation3

Consider the following proposed variation on the proof of the undecidability of the Halting Problem.

```
int main3 (p) {
       if (selfHaltTest(p))   /* if p halts on p         */
         while (1) do ;       /* then loop forever       */
       else                   /* if p doesn't halt on p  */
         while (1) do;        /* then loop forever       */
        }
```

Will this support a proof-by-contradiction as we did earlier? That is, can we get a contradiction by assuming that `selfHaltTest` behaves as claimed, and then reasoning about this `main3` program in exactly the same way as we did for `main`, namely by examining `main3(main3)`?

259. TMDecidability

(from Kozen) Show that the following problems are decidable, by giving pseudocode for a program that solves them. In each case the input is a Turing machine M.

a) Question: Does M take more than 1000 steps on input λ?

b) Question: Does there exist an input w such that M take more than 1000 steps on w?

c) Question: Does M take more than 1000 steps on *all* inputs w?

d) Question: On input λ, does M ever move its head more than 1000 tape cells away from where it started?

Chapter 30

Programs and Computability

This chapter is self-contained but for motivation it will be useful to have read Chapter 29.

For various reasons, Turing machines have emerged as a very convenient formalism for theoretical investigations. So most textbooks and research papers use them as their foundation. But we can present the most important results in the whole field just by using a formalism that is more familiar to you: ordinary programs. In this chapetr we lay the groundwork for doing that.

Restriction to the Binary Alphabet

> For simplicity, from now on, whenever we do not say anything to the contrary, we work over the binary alphabet $\Sigma_2 \stackrel{\text{def}}{=} \{0, 1\}$.

A big part of what we will be doing is relating different (often wildly different) kinds of problem to each other. But any finite alphabet can be easily encoded into Σ_2, so we might as well take advantage of that much uniformity. That is, if we assume that all of our different kinds of problems are presented using Σ_2 we can eliminate an inessential source of distraction.

30.0.1 Overview

This is just a chapter of definitions: we are giving names to some important ideas that connect computer programs, mathematical functions, and languages. If you plow through this chapter quickly you may have trouble seeing the point to all these definitions and what may seem like fussy distinctions. Have faith, and patience. You'll refer back to this chapter a lot!

We are going to define

1. how a *program* defines a mathematical *function*, and hence, what a *computable* function is;
2. the idea of the *language* of a program;
3. what a *decidable* language is;
4. what a *semidecidable* language is.

First, the basis of everything: the connection between programs and functions.

30.1 Programs as a Formal Model of Computing

Recall a few conventions about programs in the C programming environment:

- A text file is conventionally modeled as a sequence of symbols (terminated by an end-of-file marker, which we will ignore here). In standard formal-languages terminology, *a text file is simply a string over the alphabet* Σ_2
- A C program is itself just a text file.
- A C program p can read strings from the standard input file `stdin`, and it can write strings to `stdout`
- It is possible that on some inputs p fails to terminate at all, that is, the computation runs forever. (Remember that in our computational model we are not allowing the operating system to halt our program due to a stack overflow or other resource constraints.)

The takeaway from the above discussion is just this: a program is a device for transforming strings into other strings.

30.1 Definition (Program Computation)**.** Suppose the program p is executed with standard input consisting of the single bitstring x. Then we write

- $p[x]\downarrow y$ if the execution of p halts, and y is the maximal initial sequence of 0s and 1s on standard output. (If there are any non-blank symbols beyond the first blank after y we just ignore them.)
- $p[x]\uparrow$ if the execution of p does not halt.

We call this function the function computed by p and use the notation $\mathsf{pfn}(p)$.

Not every function can be represented by a program. Those that can are given a special name.

30.2 Definition. A partial function f is computable if it can be computed by a program, that is , if there is some program p such that f is $\mathsf{pfn}(p)$.

30.1.1 Discussion

Programs Are Not Functions!

At first it may seem that insisting on the distinction between programs and functions is just quibbling. But think a little more and you'll see that *of course* we want to make this distinction. Obviously we can have two radically different programs that happen to compute the same function. Indeed, if this were not true then it would make no sense to refactor or optimize programs: when you refactor a program p you making it into a different program, certainly, but you presumably want the new program p' to have the same input/output behavior. That is, you will have $p \neq p'$ but $\mathsf{pfn}(p) = \mathsf{pfn}(p')$. This helps to clarify the distinction between programs and functions.

What About Running out of Time or Memory?

We suppose that there are no constraints on the time or space allocated to the process in which our program runs. This is an important point. We are interested in the *pure* behavior of programs and want to abstract away from annoying interventions from the operating system, for example if it doesn't want to give our program as much time or stack space it wants.

When we study the *complexity* of computations, we will certainly want to measure the amount of time and/or the amount of space that a computation takes. But not here.

What About Multiple Inputs?

Often we are interested in functions that, conceptually, take more than one argument. But the above formalism encompasses that too. Mathematically speaking all functions are functions of a single argument (for example when we speak of "functions of two variables" in calculus we really mean a function whose inputs are ordered pairs $p \in \mathbb{R}^2$). If we want to speak of functions that conceptually take several string arguments, we simply encode pairs, or triples, or whatever, as single strings, using techniques we have already discussed.

30.2 Programs and Languages

OK, so we've connected programs with mathematical functions. But most of this course is about languages, sets of strings. Let's connect program to languages. The guiding intuition is simply this: we view programs just like we view finite automata or pushdown automata. They process strings and, by accepting them or not, define languages.

30.2.1 The Language of a Program

For *DFAs*, *PDAs*, etc, we have the notion of "the language $L(M)$ of a machine M". We can define "the language of a program" in the following natural way.

30.3 Definition (Language of a Program)**.** Suppose p is a program. The *language accepted by* p, denoted $L(p)$, is the set of all bitstrings on which p returns 1:

$$L(p) \stackrel{\text{def}}{=} \{x \in \{0,1\}^* \mid \ p[x] \downarrow 1\}$$

Note that a string x *fails* to be in $L(p)$ if

- *either* $p[x]$ returns some value other than 1,
- *or* $p[x] \uparrow$, *i.e.* p fails to halt on x.

30.2.2 Semidecidable Languages

Semidecidable languages are to programs as regular languages are to *DFAs* (or context-free languages are to *PDAs*).

30.4 Definition (Semi-decidable Language)**.** A language $L \subseteq \Sigma_2^*$ is *semi-decidable* if and only if it is $L(p)$ for some program p.

This is the same as saying:

A language $L \subseteq \Sigma_2^*$ is semidecidable if there is a program p such that

- whenever $x \in L$, $\mathsf{pfn}(p)(x) = 1$, and
- whenever $x \notin L$, $\mathsf{pfn}(p)(x)$ is undefined or is a value $\neq 1$

The term *recursively enumerable* is a synonym for "semidecidable."

The phrase "recursively enumerable" may seem a bit strange. It will make more sense if you read Chapter 37 below. A convenient abbreviation for "recursively enumerable" is "RE."[1]

30.2.3 Decidable Languages

Semidecidable languages are slightly strange beasts, as we will see. A strengthening of the idea, decidable languages, turns out to be easier to work with at first.

30.5 Definition. A language $L \subseteq \Sigma_2^*$ is *decidable* if there is a program p with

$$\mathsf{pfn}(p)(x) = \begin{cases} 1 & \text{if } x \in L \\ 0 & \text{if } x \notin L \end{cases}$$

If a language is not decidable we say it is *undecidable*.

30.6 Remark. If you remember the term "characteristic function,", from Section 1.3: to say that a language $L \subseteq \Sigma_2^*$ is decidable is precisely to say that the characteristic function of L is a computable function.

Programs such as the above that always return 0 or 1 are important enough to deserve their own name.

30.7 Definition (Decision Procedures)**.** Let p be a program. Say that p is a *decision procedure* if for every input x, either p returns 1 or p returns 0.

The fundamental thing about decision procedures is not the specifics of returning 0 and 1, that's a detail: any two distinct return values would have the same effect. As will become clearer, the crucial fact is that they never run forever on any input. They are guaranteed to give us an answer, no matter what the input.

30.2.4 Decidable vs Semidecidable

The definitions of "A is decidable" and "A is semidecidable" look very similar. Each of them depends on having a program that accepts precisely the members of A (that is recognizes A). The difference is just this: to show A to be decidable you have to show there is a Turing machine recognizing A that always halts.

[1] Indeed this is such an easy-to-pronounce shorthand that many people prefer to use "RE" to refer to a semidecidable language.

The significance of this will become clearer as we go along.

You might be inclined to think the following. "There's no real difference between A being decidable and being semidecidable, because if you give me some program p recognizing A, I'll just build a new Turing machine p' recognizing A that always halts, by making p' halt and reject on all the inputs where p fails to halt. That way, whenever a language A is semidecidable, it is actually decidable as well."

OK, so, that isn't crazy. It has the same feeling somehow of the trick we played to convert *NFAs* to *DFAs*, showing that what looked like a big difference in machines turns out to be no rel difference.

But **the idea doesn't work!**. There are semidecidable languages that are not decidable. Which means there are programs that cannot be converted into an always-halting programs. Stay tuned.

30.2.5 Decidable Problems/Languages

The vast majority of problems you have ever thought about before are decidable. Asking whether integers are prime, asking whether a graph is connected, asking whether a Java program is syntactically valid, asking whether a one-variable polynomial has integer roots, *etc.*. Since computer science is so much about the study of algorithms for solving problems we, almost by definition, tend to study decidable problems.

By the way, the study of decidability does not concern itself with the *complexity* of algorithms for solving a problem. Once we know an algorithm exists, we declare victory and move on.

Any Regular Language is Decidable

To see this, all we need to do is realize that given any *DFA* M we can write a program p_M that simulates M in the sense that

- if M halts in an accepting state on x, then p_M will halt and return 1 on x;
- if M halts in a non-accepting state on x, then p_M will halt and return 0 on x.

Any Context-Free Language is Decidable

To do this, we *could* proceed as for regular languages and show how to simulate *PDAs* by programs. This is doable, although the fact that *PDAs* can be non-deterministic means we have to make our programs do some kind of backtracking.

But having done the work in Chapter 22 all we have to do is quote the main result there, Corollary 22.6 (which says precisely that that for every *CFG* G there is a program deciding membership in $L(G)$).

Non-Context-Free Decidable Languages

The language

$$D \stackrel{\text{def}}{=} \{ww \mid w \in \{0,1\}^*\}$$

is decidable. A program to decide membership in D can just take its input x, divide it in half, rejecting if the length of x is odd, then check that the two halves are the same. But D is not context-free.

As another example: the language of prime numbers in binary is a decidable language. This just means that there exists a program p that takes strings x as input and (i) halts in an accepting state if x codes a prime number and (ii) halts in a non-accepting state if x codes a non-prime.

An equivalent thing to say is that the following *problem* is decidable.

> *Primality*
>
> INPUT: a binary string x
>
> QUESTION: does x code a prime in binary?

Undecidable Languages

But there are lots on undecidable problems out there. Chapter 29 was all about showing you a concrete example of one.

30.8 Example. The *language* $\mathsf{SelfHalt} \stackrel{\text{def}}{=} \{p \mid \ p \text{ halts on } p\ \}$ is an undecidable language. This is what Then Theorem 29.1 says.

An equivalent thing to say is that the Self-Halting *problem* is undecidable.

Besides the Halting Problem, three interesting examples are: asking whether a Java program is semantically valid, asking whether a context-free grammar is ambiguous, and asking whether a multi-variable polynomial has integer roots. We'll study these in Chapter 34, Chapter 39, and 41, respectively.

30.3 Undecidable Problems: a Cardinality Argument

Chapter 29 gives a concrete example of an undecidable language, SelfHalt. But if we didn't care about concrete examples we can show that there exist undecidable languages, just based on cardinality. There can only be *countably* many decidable languages, because there are only countably many Turing machines. So most languages are undecidable!

30.9 Theorem. *There exist languages that are not decidable.*

Proof. A Turing machine is a finite object built over certain finite alphabets of symbols. Thus the set of all Turing machines is countable. Thus the set of all computable functions is countable. So certainly the set of all computable characteristic functions over any Σ^* is countable. But there are uncountably many characteristic functions over any Σ^*. So there are languages whose characteristic functions are not computable: these are the undecidable languages. ///

30.4 Summary

1. Any program p determines a partial function $\mathsf{pfn}(p)$.
2. A function is *computable* if there is some program p such that the function is $\mathsf{pfn}(p)$
3. The *language* of a program p is
$$L(p) \stackrel{\text{def}}{=} \{x \in \{0,1\}^* \mid \ p[x] \downarrow 1\}$$
4. A language is *semidecidable* language if it is $L(p)$ for some p.
5. A language is *decidable* language if it is $L(p)$ for some p that always halts.
6. One example of an *undecidable* language is SelfHalt.

Caution

Programs are used in two completely different ways when we study computability.

1. they are a way to define what "computability" *means* (as in Chapter 32), and also
2. they are sometimes the things that we ask computability questions *about*.

So when we ask decidability questions about programs we are asking whether there can be a program that can answer yes/no questions other programs. Don't let this throw you! It makes perfect sense; programs process other programs all the time: think about compiling, type-checking, refactoring, etc.

30.5 Problems

260. DescribeFuns

For each part,

1. Describe the partial function f from Σ_2^* to Σ_2^* computed by the programs described by the following pseudocode.
2. Say whether f is a total function

a) a program that

1. scans each symbol in `stdin`;
2. copies that symbol to `stdout`;
3. terminates when end-of-file is reached in `stdin`

b) a program that immediately halts without reading any of `stdin`.

c) a program that immediately goes into an infinite loop without reading any of `stdin` or writing anything to `stdout`.

d) a program that immediately begins writing an infinite sequence of 1s on `stdout` (and never terminates).

e) a program that scans past all the symbols in `stdin` (without writing anything) and then halts.

f) a program that scans past all the symbols in `stdin`, writes "010" on `stdout`, then goes into an infinite loop.

g) a program that scans `stdin`, appending a "1" to the current contents of `stdout` as each symbol is read.

h) a program that maintains a stack memory and behaves like this:

1. it scans past symbols in `stdin` and pushes them onto the stack until end-of-file is read;
2. then it pops each symbol off the stack and writes it to `stdout`

261. Acceptance

Here is a variation on the Halting Problem, inspired by our definition of what it means for a program to accept a string.

> *The Acceptance Problem*
>
> INPUT: a C program p and a bitstring x
>
> QUESTION: does p accept x?

So the answer to the question should be yes if $p[x] \downarrow 1$ and should be no otherwise (if either $p[x] \downarrow$ some value other than 1 OR $p[x] \uparrow$).

Prove that the Acceptance Problem is undecidable.

Hint. Look at the proof that the Halting Problem is undecidable. You don't have to change very much!

Chapter 31

Turing Machines

You are familiar with two models of computation:

- finite automata, which recognize the *regular* sets, and
- pushdown automata, which recognize the *context-free* sets.

Pushdown automata are more powerful than finite automata: they can recognize languages that finite automata cannot, such as $\{a^n b^n \mid n \geq 0\}$. But pushdown automata are not the most powerful computing device we can imagine, since they cannot recognize certain sets that we can imagine recognizing by a machine; an example is $\{a^n b^n c^n \mid n \geq 0\}$.

So we now move on to a model of computation which is more powerful than finite and pushdown automata.

- Turing machines, which (as we will see) recognize the *semidecidable* sets.

Roughly speaking a Turing machine is a finite automaton with an unbounded read/write memory. A Turing machine can be used as a language recognizeor just as finite automata and pushdown automata can, but since they can write as well as read symbols they can also be viewed as computing functions.

This model will be powerful enough to capture everyone's intuitive notion of "computing machine." In fact Turing, in his 1937 paper [Tur37], defined Turing machines (though of course he didn't call them that) as part of a thought experiment about what "computing" is in its essence. It is really interesting to read the beginning of the paper. Remember that he did all of this before computers as we know them had been invented.

We'll follow the same pattern we did in defining pushdown automata. After an informal description, we define

- what a Turing machine is
- what a configuration is
- what a transition is
- what a computation is

31.1 Informal Description

A Turing machine (TM) is a machine that maintains a notion of current state and can read and write a linear memory which at any instant is a sequence of symbols.

The states are elements of a finite set Q; there is a single start state q_s and a set F of accepting states.

The memory is traditionally called a *tape*. We imagine that the tape is (potentially) infinite to the left and the right.[1]

There is an input alphabet Σ as usual. There is a tape alphabet Γ, which includes Σ but will have extra symbols, including at least the special symbol $\square$ not in Σ, which we refer to as "blank."

Depending on the current state p and the current input symbol a, the machine can advance to a next state p', rewrite a to be symbol a', and move either left or right. It is permitted that $p = p'$ and/or $a = a'$.

So the moves can be defined formally by a move function δ, as described in Definition 31.1. Moving left is indicated by "L" in the δ function; moving right is indicated by "R" .

Note that δ is a *partial* function, that is, for some q and a, $\delta(q,a)$ need not be defined. If we ever have a situation where δ does not apply to the current (state, symbol) pair then the machine just halts.

31.2 Formal Definition

[1] Some versions allow the tape be infinite only in one direction. Details like this don't change what can be computed.

31.1 Definition. A Turing machine M is a tuple $(\Sigma, \Gamma, Q, \delta, q_s, F)$, where

- $\Sigma \subseteq \Gamma$ is a finite *input alphabet*, with $\square \notin \Sigma$;
- Γ is a finite *tape alphabet*, containing at least $\square$ and the symbols of Σ;
- Q is a finite set of *states*
- $q_s \in Q$ is the start state;
- $F \subseteq Q$ is the set of "accepting" states;
- δ is a partial function:

$$\delta : (Q \times \Gamma) \rightharpoonup (Q \times \Gamma \times \{L, R\})$$

Note that for convenience we view the symbol $\square$ as being "globally" defined, that is, we don't allow each Turing machine to determine its own blank. This just cuts down on the notational baggage in defining a Turing machine.

Pictures

Just as with finite automata and with *PDAs*, it can be helpful to draw pictures of Turing machines. We use circles for the states, and label the arcs with (i) the symbol being read, (ii) the new symbol, and (iii) the direction, left or right.

Our Turing machine pictures look a bit like our *PDA* pictures earlier, but with a tweak. In a *PDA* picture an arc label

$$a\ b/x$$

means: "when reading input symbol a, if the stack has b on the top, replace that b with string x." In a Turing machine picture an arc label

$$a/a'\ D$$

means: "when reading input symbol a, replace a by a' on the tape, and move in direction D (left or right)."

31.2.1 Turing Machine Examples

Here are a few simple Turing machines and their pictures. At this point we can't really say, if we are intellectually honest, what these machines *do* since we haven't yet

given formal definition of Turing machine *computations*. But we need to see examples to gain intuition! So use these examples to get familiar with the notation, then, after we define Turing machine computations in Section 31.3, you can in principle come back and check our descriptions.

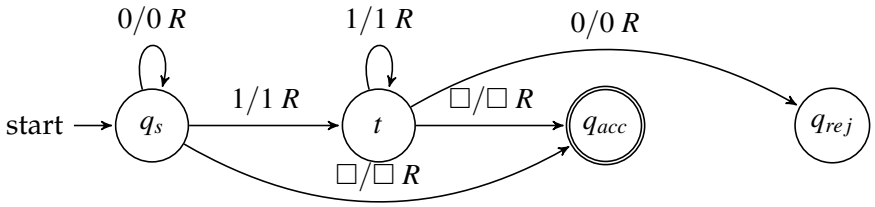
31.2 Example. Let M_1 be defined by

- $Q = \{q_s, t, q_{rej}, q_{acc}\}$ $\quad \Sigma = \{0,1\}$ $\quad \Gamma = \{0,1,\square\}$ $\quad F = \{q_{acc}\}$
- the move function δ is given by
  - $(q_s, 0) \mapsto (q_s, 0, R)$
  - $(q_s, 1) \mapsto (t, 1, R)$
  - $(q_s, \square) \mapsto (q_{acc}, \square, R)$
  - $(t, 0) \mapsto (q_{rej}, 0, R)$
  - $(t, 1) \mapsto (t, 1, R)$
  - $(t, \square) \mapsto (q_{acc}, \square, R)$

Note that this Turing machine never changes the tape symbol and always moves right, and it halts when it reads the first $\square$. So it is essentially a DFA.

If M_1 is started with a string w of the form 0^*1^* then it will scan that string and eventually end up in state q_{acc}. If M_1 is started with a string w that doesn't start in the form 0^*1^* then it will eventually end up in state q_{rej} and halt.

Here is a subtlety: If M_1 is started with a tape consisting of a w of the form 0^*1^* then a blank, then anything else, it will still end up in state q_{acc} and halt without reading that stuff after the first blank. We will have to be careful about this kind of thing when we define carefully what it means to do a computation on a string.



31.3 Example. Let M_2 be defined by

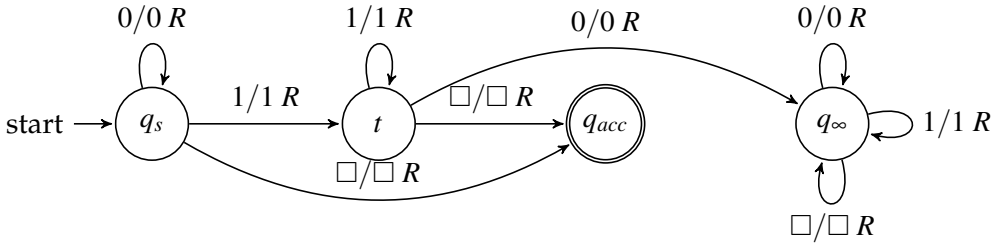
- $Q = \{q_s, t, q_\infty, q_{acc}\}$ $\quad \Sigma = \{0,1\}$ $\quad \Gamma = \{0,1,\square\}$ $\quad F = \{q_{acc}\}$
- the move function δ is given by
  - $(q_s, 0) \mapsto (q_s, 0, R)$
  - $(q_s, 1) \mapsto (t, 1, R)$

- $(q_s, \square) \mapsto (q_{acc}, \square, R)$
- $(t, 0) \mapsto (q_\infty, 0, R)$
- $(t, 1) \mapsto (t, 1, R)$
- $(t, \square) \mapsto (q_{acc}, \square, R)$
- $(q_\infty, 0) \mapsto (q_\infty, 0, R)$
- $(q_\infty, 1) \mapsto (q_\infty, 1, R)$
- $(q_\infty, \square) \mapsto (q_\infty, \square, R)$

This machine is very similar to the previous example. As a check on your understanding of the notation, see if you can say what the difference is in their behavior before reading further.

If M_2 is started with a string w of the form 0^*1^* then it will scan that string and eventually end up in state q_{acc}. If M_2 is started with a string w not of the form 0^*1^* then it will never halt.

So this is the difference between the examples: M_2 differs from M_1 in that if a string is not accepted by M_2 the computation never halts at all (as opposed to M_1, where the computation halts, but in a non-accepting state). But the set of strings taking M_2 to an accepting state is the same as the set of strings taking M_1 to an accepting state.



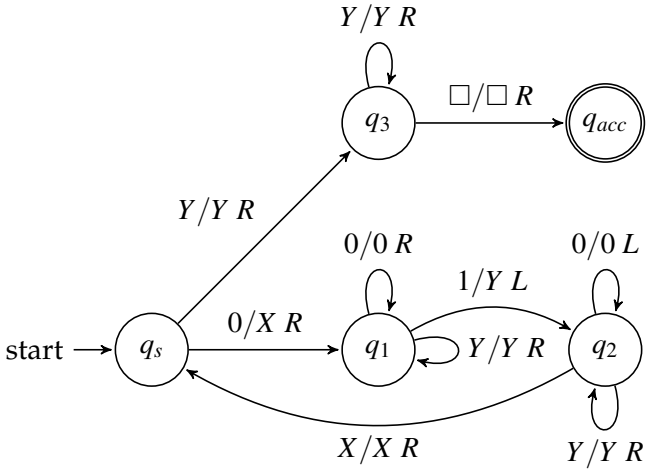
31.4 Example. The previous two machines were designed to accept strings in the regular language 0^*1^*. Here is a Turing machine designed to accept the non-regular language $\{0^n1^n \mid n > 0\}$. This machine is (essentially) taken from the Hopcroft, Motwani, and Ullman text ([HMU06]).

Let M_3 be defined by

- $Q = \{q_s, q_1, q_2, q_3, q_{acc}\}$ $\quad \Sigma = \{0, 1\}$ $\quad \Gamma = \{0, 1, X, Y, \square\}$ $\quad F = \{q_4\}$
- the move function δ is given by
  - $(q_s, 0) \mapsto (q_1, X, R)$
  - $(q_s, Y) \mapsto (q_3, Y, R)$
  - $(q_1, 0) \mapsto (q_1, 0, R)$

- $(q_1, 1) \mapsto (q_2, Y, L)$
- $(q_1, Y) \mapsto (q_1, Y, R)$
- $(q_2, 0) \mapsto (q_2, 0, L)$
- $(q_2, X) \mapsto (q_s, X, R)$
- $(q_2, Y) \mapsto (q_2, Y, L)$
- $(q_3, Y) \mapsto (q_3, Y, R)$
- $(q_3, \square) \mapsto (q_{acc}, \square, R)$

If M_3 is started with a string w of the form 0^n1^n for some n then it will scan that string and eventually end up in state q_{acc}. If M_3 is started with a string w not of the form 0^n1^n then it will eventually halt in a state other than q_{acc}.



31.3 Turing Machine Computations

As with our previous machine models, we formalize a computation as a sequence of configurations of a machine.

31.3.1 Configurations

To describe the configuration of a Turing machine at a particular instant of time we must specify

1. the current state,
2. the contents of the tape, and
3. the position on the tape currently being scanned by the reading head

One way to do this is to write down three things: (i) the contents of the tape to the left of the reading head, (ii) the current state, and (iii) the contents of the tape at and to the right of the reading head.

Immediately we see an awkwardness in writing down what the contents of the tape are, due to the fact that the tape is infinite. We do not want to have to write down an infinite string of tape alphabet symbols! But we are saved by the fact that at any moment in a computation there will be only finitely many non-blank symbols on the tape. Why?

1. We always start computations with only finitely many non-blank symbols on the tape, and
2. we add at most one non-blank symbol at each step.

So after finitely many steps in a computation there can only be finitely many non-blank symbols on the tape (a mixture of Σ symbols and other Γ symbols). So we can communicate the contents of the tape by simply writing down enough Γ symbols to include all the non-blank ones, with the understanding that there are infinitely many blanks to the left and to the right of what we write.

This leads to the official definition.

31.5 Definition (Turing Machine Configuration)**.** Let $M = (\Sigma, \Gamma, Q, \delta, q_s, F)$ be a Turing machine.

- A *configuration* for M is a string of the form

  $$[\,\gamma_1,\, q,\, \gamma_2\,]$$

  where $q \in Q$ and $\gamma_1, \gamma_2 \in \Gamma^*$.
- An *initial configuration* is one of the form

  $$[\,\lambda,\, q_s,\, x\,]$$

  where $x \in \Sigma^*$
- A *halting configuration* for M is one where no transition applies.

A few notes:

- An initial configuration is not a different kind of thing from an ordinary configuration: the empty string λ and a string $x \in \Sigma^*$ certainly count as elements of Γ^*.

- Once a computation gets going there may be blanks sprinkled in among non-blanks.
- We don't worry about writing down "extra" blanks. We permit ourselves to have configurations with tape contents having some blanks at the beginning and/or at the end. Honest, life is easier if we allow this.

  This convention does have the consequence that there is not a unique way to write a configuration to capture the Turing machine at a given instant. But this causes no harm.

Note that being a halting configuration depends on the transitions defined for M; some $[\,\gamma_1,\, q,\, \gamma_2\,]$ might be a halting configuration for some M but not be a halting configuration for some other M.

31.3.2 Transitions

A transition is a move from one configuration to another.

31.6 Definition (Turing Machine Transitions)**.** Let $M = (\Sigma, \Gamma, Q, \delta, q_s, F)$ be a Turing machine.

The one-step transition relation $\Longrightarrow$ is a binary relation on configurations, defined as follows.

Let

$$[\,X_1 \ldots X_{i-1},\, q,\, X_i \ldots X_n\,]$$

be a configuration. The next configuration is determined by $\delta(q, X_i)$; there are two cases depending on whether the reading head moves left or right.

- *Case 1:* $\delta(q, X_i) = (p, Y, L)$. Then

$$[\,X_1 \ldots X_{i-1},\, q,\, X_i \ldots X_n\,] \Longrightarrow [\,X_1 \ldots X_{i-2},\, p,\, X_{i-1} Y X_{i+1} \ldots X_n\,]$$

- *Case 2:* $\delta(q, X_i) = (p, Y, R)$. Then

$$[\,X_1 \ldots X_{i-1},\, q,\, X_i \ldots X_n\,] \Longrightarrow [\,X_1 \ldots X_{i-1} Y,\, q,\, X_{i+1} \ldots X_n\,]$$

There are various corner cases to be fussed over, for example when $X_1 \ldots X_{i-1}$ is the empty string or $X_i \ldots X_n$ is the empty string (meaning that there are all-blanks to the left or to the right of the head. The notation above correctly captures these situations using our convention that we can always "add blanks" to either end of a configuration without changing its meaning.

Halting If $\delta(p, X_1)$ is not defined then there is no transition out of the configuration $X_1 \ldots X_{i-1} q X_1 \ldots X_n$. We say that the machine *halts.*

31.3.3 Computations

Now we can formally define what a Turing machine computation is: it is a sequence of transitions starting with an initial configuration.

As usual we define the relation $\overset{*}{\Longrightarrow}$ to be the reflexive transitive closure of $\Longrightarrow$. This means that $C \overset{*}{\Longrightarrow} D$ if D follows from C by a finite number (zero or more) steps of $\Longrightarrow$.

31.7 Definition (Turing Machine Computation)**.** Let $M = (\Sigma, \Gamma, Q, \delta, q_s, F)$ be a Turing machine.

A *computation* of M on word $x \in \Sigma^*$ is a sequence

$$[\,\lambda,\, q_s,\, x\,] \overset{*}{\Longrightarrow}$$

of transitions starting with an initial configuration on x.

Don't confuse the transition relation $\Longrightarrow$ with the move relation δ. The relation $\Longrightarrow$ acts on configurations and so describes the action of the machine. The move relation δ just describes "locally" what can happen. In a sense δ exists only in order for us to define $\Longrightarrow$. A good metaphor is: δ is the *program* for the machine, while $\overset{*}{\Longrightarrow}$ describes the *processes* that can happen.

From a given machine configuration C there is at most one move which can be taken meaning that there is at most one configuration C' with $C \Longrightarrow C'$. It follows that computation is deterministic, meaning that for any initial C_{initial} there is at most one halting configuration C_{final} with $C_{\text{initial}} \overset{*}{\Longrightarrow} C_{\text{final}}$.

There may be no halting configuration reachable from C_{initial}, though: the computation might never halt.

Computation Examples

Here we revisit the machines from Example 31.2, Example 31.3, and Example 31.4.

1. Here is a computation of machine M_1 in Example 31.2 on input 001

$$\begin{aligned}
[\,\lambda,\, q_s,\, 001\,] &\Longrightarrow [\,0,\, q_s,\, 01\,] \\
&\Longrightarrow [\,00,\, q_s,\, 1\,] \\
&\Longrightarrow [\,001,\, t,\, \square\,] \\
&\Longrightarrow [\,001\square,\, q_{acc},\, \square\,]
\end{aligned}$$

Since no δ moves apply in this last configuration, M_1 halts on 001.

Note that we have taken advantage of the flexibility we have granted ourselves in showing or not showing trailing or leading blanks. For example the configuration $[\,001,\ t,\ \Box\,]$ in the third line could also have been written $[\,001,\ t,\ \lambda\,]$. Writing it the way we did made it easier to see that the transition out of state t reading a blank was the one that applied.

2. Here is a computation of machine M_1 in Example 31.2 on input 0100

$$\begin{aligned}[\,\lambda,\ q_s,\ 0100\,] &\Longrightarrow [\,0,\ q_s,\ 100\,] \\ &\Longrightarrow [\,01,\ t,\ 00\,] \\ &\Longrightarrow [\,010,\ q_{rej},\ 0\,]\end{aligned}$$

Since no δ moves apply in this last configuration, M_1 halts on 0100. Note that it did not read the entire input.

3. Here is a computation of machine M_2 in Example 31.3 on input 0100

$$\begin{aligned}[\,\lambda,\ q_s,\ 0100\,] &\Longrightarrow [\,0,\ q_s,\ 100\,] \\ &\Longrightarrow [\,01,\ t,\ 00\,] \\ &\Longrightarrow [\,010,\ q_\infty,\ 0\,] \\ &\Longrightarrow [\,0100,\ q_{infty},\ \Box\,] \\ &\Longrightarrow [\,0100\Box,\ q_\infty,\ \Box\,] \\ &\Longrightarrow [\,0100\Box\Box,\ q_\infty,\ \Box\,] \\ &\ldots\end{aligned}$$

This is a non-terminating computation.

These examples illustrate a tricky, but essential, point. There are two different ways that a Turing machine M can fail to accept a string x: either (i) M halts on x in a non-accepting state or (ii) M fails to halt on x.

Here is a computation of machine M_3 in Example 31.4 on input 01

$$\begin{aligned}[\,\lambda,\ q_s,\ 01\,] &\Longrightarrow [\,X,\ q_1,\ 1\,] \\ &\Longrightarrow [\,\lambda,\ q_2,\ XY\,] \\ &\Longrightarrow [\,X,\ q_s,\ Y\,] \\ &\Longrightarrow [\,XY,\ q_3,\ \lambda\,] \\ &\Longrightarrow [\,XY,\ q_{acc},\ \lambda\,]\end{aligned}$$

There are no more transitions defined, and we are in an accepting state, so this is an accepting computation.

31.4 Turing Machines Recognize Languages

We can view a Turing machine M as accepting strings $x \in \Sigma^*$. Namely: run M on x and see if M halts in an accepting state. In this way Turing machines recognize languages.

31.8 Definition. Let $M = (\Sigma, \Gamma, Q, \delta, q_s, F)$ be a Turing machine. The *language* $L(M)$ of M is the set of those $x \in \Sigma^*$ such that

$$[\,\lambda,\, q_s,\, x\,] \stackrel{*}{\Longrightarrow} [\,\gamma_1,\, f,\, \gamma_2\,]$$

where $[\,\gamma_1,\, f,\, \gamma_2\,]$ is a halting configuration and $f \in F$.

Remember from Definition 31.5 that a *halting configuration* for M is one where no transition applies.

31.9 Examples. Returning again to the examples from Section 31.2.1.

1. Machine M_1 on input 001:

   Since M_1 halts in the accepting state q_{acc} on 001, M_1 accepts the string 001.

2. Machine M_1 on input 0100:

   Since M_1 halts in the non-accepting state q_{rej} on 0100, M_1 does not accept the string 0010.

3. Machine M_2 on input 0100 :

   Since M_2 fails to halts on 0100, M_1 does not accept the string 0010.

31.10 Definition (Turing-Semidecidable Language)**.** A language $L \subseteq \Sigma^*$ is *Turing-semidecidable* if there exists a Turing machine M with $L = L(M)$.

31.5 Some Turing Machines Decide Languages

For a *DFA*, a computation on a string x always halts, in an accepting state or a non-accepting state. So it is natural to speak of a *DFA* "rejecting" a string.

For Turing machines things are trickier. For a given input x, the Turing machine can fail to accept x either by halting in some non-accepting state, or by looping forever. These are intuitively very different situations. The first is an "observable outcome" but the infinite-loop situation leaves us hanging: there is never a particular moment when we know that the computation will not accept.

We'll say that a Turing machine decides a language L if it recognizes L and never leaves us hanging. Formally:

31.11 Definition. Let $M = (\Sigma, \Gamma, Q, \delta, q_s, F)$ be a Turing machine. Say that M *decides* language $D \subseteq \Sigma^*$

- M recognizes L, and
- M halts on every input.

Another way to phrase this is to say that M decides D if D is $L(M)$ and M halts on every input.

The crucial requirement of Definition 31.11 is that M must halt on inputs *not* in L as well as those in L.

Important It is a very strong claim to say that a certain Turing machine M decides a language D. **Every** Turing machine *recognizes* some language. But only particularly **nice** Turing machines can *decide* a language.

31.12 Definition (Turing-Decidable Language)**.** A language $L \subseteq \Sigma^*$ is *Turing decidable* if there exists a Turing machine M that decides L

Any Regular Language is Turing Decidable

Let M be any *DFA*. We can view M as a Turing machine which (i) never changes the tape, (ii) always moves to the right, and (iii) halts as soon as it sees a blank.

To do this formally we need to deal with the fact that the δ function of a *DFA* takes state-symbol pairs as input and return states as output, while the δ function of a Turing machine takes state-symbol pairs as input and returns a triple of a state, a tape symbol, and a direction.

So formally, if M is $(\Sigma, Q, q_s, \delta, F)$ as a *DFA* we define the Turing machine M' to be $(\Sigma, Q, q_s, \delta', F)$ where δ' is defined as

$$\delta'(q,a) = ((\delta(q,a), a, R))$$

Any Context-Free Language is Turing Decidable

It is not so simple to view a *PDA* as a Turing machine. First of all, a *PDA* has *two* memories: the input tape and the stack. Second of all, *PDAs* can be non-deterministic. Nevertheless it is true that given any *PDA* we can build a Turing machine that accepts the same language.

We can simulate the *PDA* stack by using the portion of *TM* tape to the right of the input. We can simulate the *PDA* nondeterminism by programming the Turing machine to do backtracking. You don't want to see the details.

Non-Context-Free Turing Decidable Languages

The language

$$D \stackrel{\text{def}}{=} \{ww \mid w \in \{0,1\}^*\}$$

and the language of prime numbers in binary are each Turing decidable.

It would be no fun to design Turing machines to decide these languages. In each case the way we know there *is* a Turing machine doing the job is to use the very general fact, discussed in Chapter 32, that whenever there is a *program* to do a certain job, there will be a Turing machine to do that job.

Non-Turing-Decidable Languages

But there are lots of non-Turing-decidable languages. The easiest way to see *that* is to rely on the other direction of the result that Turing machines and programs are equally powerful: if there *cannot* be a program to do a certain job, there cannot be a Turing machine to do that job. So for example the program SelfHalting problem is one that cannot be decided by a Turing machine.

31.6 Turing Machines Determine Functions

So far we have considered Turing machines simply as as string-acceptors, that is, as defining languages. This is to stress the relationship with finite automata and pushdown automata. But since Turing machines can write to their tape, it is natural to view them as computing functions from Σ^* to Σ^* , not just accepting strings or rejecting strings.

If an input string $x \in \Sigma^*$ is given to a Turing machine M, and M happens to halt with string $y \in \Sigma^*$ on its tape, then it is reasonable to say that M has produced put y on input x.

We needn't care what state M halted in. We do need to be a little fussy about blanks and other non-Σ symbols on the tape at the end.

31.13 Notation. Let $M = (\Sigma, \Gamma, Q, \delta, q_s, F)$ be a Turing machine.

Suppose M is executed with input $x \in \Sigma^*$ Then we write

- $M[x] \downarrow y$ if M halts on input x and in the halting configuration, $y \in \Sigma^*$ is the maximal initial sequence of Σ-symbols on the tape, starting from the left. If there are any non-blank symbols beyond y we just ignore them.
- $M[x] \uparrow$ if the execution of M does not halt on x.

It's a bit arbitrary to accept an output y as liberally as the definition does. But our motivation to be able to say that any time a Turing machine halts on an input, a function out is determined. In this way the only time a Turing machine fails to return an answer for a given input is when the computation fails to halt. Things are technically simpler this way.

The definition of "partial function" is given in Section 1.2.1.

31.14 Definition (Partial Function Computed by a Turing Machine)**.** Let $M = (\Sigma, \Gamma, Q, \delta, q_s, F)$ be a Turing machine.

The *partial function computed by* M, denoted $\mathsf{pfn}(M)$, is defined on strings $x \in \Sigma^*$ as follows

$$\mathsf{pfn}(M)(x) = \begin{cases} y & \text{if } M[x] \downarrow y \\ \text{undefined} & \text{if } M[x] \uparrow \end{cases}$$

31.15 Definition (Computable Function)**.** A partial function f is *computable* if it can be computed by a Turing machine, that is , if there is some Turing machine M such that f is $\mathsf{pfn}(M)$.

What About Running out of Time or Memory? We suppose that there are no constraints on the time or space allocated to the process in which our machine runs. This is an important point. We are interested in the *pure* behavior of machines and want to abstract away from annoying interventions from the outside world; for example if it doesn't want to give our Turing machine as much time or tape space it wants.

When we study the *complexity* of computations, we will certainly want to measure the amount of time and/or the amount of space that a computation takes. But not here.

What About Multiple Inputs? Often we are interested in functions that, conceptually, take more than one argument. But the above formalism encompasses that too. Mathematically speaking all functions are functions of a single argument (for example when we speak of "functions of two variables" in calculus we really mean a function whose inputs are ordered pairs $p \in \mathbb{R}^2$). If we want to speak of functions that conceptually take several string arguments, we simply encode pairs, or triples, or whatever, as single strings, using techniques we have already discussed.

31.7 Turing Machine Variations

There is a tremendous variety of syntactic sugar that can be added to Turning machines. In other words, there are many variations on Turing machines which ostensibly give more programming power but in fact are equivalent to the standard model. Examples include

- Turing machines with multiple tapes.
- Turing machines with 2-way infinite tapes.
- Turing machines with 2-dimensional tapes.
- Turing machines with multiple tracks. This is a single tape Turing machine whose tape is divided into parallel tracks. In each move, the Turing machine can read all the tracks at once and move and change the contents of the tracks.
- two-way Turing machines
- nondeterministic Turing machines

But in every case we can compile one of these richer machines into a standard machine. You can find many examples with an internet search.

31.8 Summary

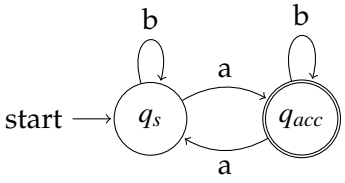
1. Every Turing machine **recognizes** a language. Such languages are called Turing-semidecidable.
2. Some Turing machines—the ones that halt on all inputs—**decide** a language. Such languages are called Turing-decidable.
3. Every Turing Machine determines a **partial function.**
4. Some Turing machines—the ones that halt on all inputs—determine a **total function.**
5. There is a subtle difference between the way we use the terms *decidable* and *computable*. It is *functions* that are either computable or not; it is *languages* that are either decidable or not.

Caution: it is possible that a language A is recognized by a certain Turing machine M but is not decided by M, while there could be another, smarter, Turing machine M' that decides A. In this case A is indeed a Turing-decidable language (even though the dumber Turing machine M doesn't let us see that).

31.9 Problems

262. DFA-TM

Consider the obvious DFA over $\Sigma = \{a, b\}$ which has two states and which accepts precisely the strings of with an odd number of as.



a) Informally, this DFA "is" a Turing machine which simply never changes the input tape and always move to the right. Formalize this intuition by defining formally the Turing machine corresponding the DFA. That is, write down a Turing Machine

$$(Q', \Sigma, \Gamma, \delta', q_s, F')$$

where δ' is now a TM-appropriate partial function

$$\delta' : (Q' \times \Gamma) \rightharpoonup (Q' \times \Gamma \times \{L, R\})$$

Say explicitly what each of Σ, Q', Γ, q_s, and F' is; you can convey the δ' partial function as a picture if you like.

b) Write down explicitly the computation (sequence of configurations) that results when this machine is started on input *abbab*.

263. RevisitEgs1

For the machines in Example 31.2 and Example 31.3, write down the computation arising from input 0010. In each case your answer will be a sequence of configurations.

Of course it is possible that a computation might be infinite. If so, generate enough of it so that the pattern is clear, and explain.[2]

264. RevisitEgs2

For the machine in Example 31.3 write down the computation arising from input 00. Your answer will be a sequence of configurations.

[2] By the way, we will soon see that if a Turing machine does have an infinite computation on an input, there may not be any easily-describable "pattern" to the sequence of configurations. But that doesn't happen in this problem.

265. LameTMs

Fix the alphabet $\Sigma = \{a, b\}$.

Make a Turing machine that, on any input $w \in \Sigma^*$ halts and accepts immediately.

Make a Turing machine that, on any input $w \in \Sigma^*$ halts immediately without accepting.

Make a Turing machine that, on any input $w \in \Sigma^*$ goes into an infinite computation (that is, never halts, and so never accepts).

Each of these should be very easy. See how small you can make your machines.

266. DecCompl

a) Suppose L is a decidable language. Explain why $\overline{L}$ is decidable.

b) Suppose L is an undecidable language. What can you say about $\overline{L}$?

267. SemiDecCompl

If you did problem DecComp:

Suppose L is a semidecidable language that is not decidable. Do you think that $\overline{L}$ is necessarily semidecidable?

You are not in a position yet to prove anything here. The purpose of this problem is to invite you to think about whether the argument you gave for problem DecComp can be adapted to work here. It is a very illuminating exercise, trust me!

268. TMSwap

1. Give a formal definition of a Turing machine which behaves as follows.

   - The input alphabet is $\{a, b\}$
   - The tape alphabet is $\{a, b, \Box\}$
   - The machine scans the tape once, replacing as by bs and vice-versa, and goes into a halting-accepting state when it first sees a blank.

   Your main job is to decide what states are needed, and write the above description as a set of quintuples comprising the δ move relation.

2. Write down explicitly the sequence of configurations that results when this machine is started on input *abbab*.

269. TMSwap2

Repeat the previous problem for the Turing machine similar to the one there but which replaces each a by a b and replaces each b by an aa.

270. TMNo00

Give a formal definition of a Turing machine M over the input alphabet $\Sigma = \{0,1\}$ such that

- $L(M) = \{1^n \mid n > 0\}$;
- M fails to halt on input string w whenever there are two consecutive 0s in w
- M halts normally in state q_{rej} for all other strings

271. TMOneAcc

Show that any Turing machine can be imulated by a Turing machine with a single accepting state. That is, show that given a Turing machine M there exists a Turing machine M' with one accepting state such that $L(M') = L(M)$.

272. TMStayPut

A Turing machine must move the tape head either left or right at each step. Sometimes this restriction is annoying. Consider the following variation on Turing machines: at each step the tape head can move left, or move right, or remain in the same place. The notions of computation and language acceptance for these new machines are the obvious trivial generalizations of the definitions for ordinary TMs.

Show that this variation adds no computing power, in the sense that given any machine M of the new type there exists an ordinary Turing machine M' such that$L(M') = L(M)$. The obvious idea for the construction works; the real content of this problem is to have you write down the construction rigorously.

273. PDA2

A pushdown automaton is, as you recall, an NFA equipped with a stack memory. It is easy to see that PDAs are not as powerful as Turing machines, since there exists a Turing machine to accept the set $\{a^n b^n c^n \mid n \geq 0\}$ (we know this since it is easy to see that there is a C program to accept this language).

Let's define a "PDA2" to be just like a PDA except that it has two stacks.

1. Make a formal definition of PDA2. Take as your model the definition of standard PDA which follows.

   Definition. A pushdown automaton M is a tuple $(Q, \Sigma, \Gamma, \delta, s, \perp)$, where

   - Q is a finite set of *states*, with $s \in Q$,
   - Σ is a finite *input alphabet*,
   - Γ is a finite *stack alphabet*, with $\perp \in \Gamma$, and
   - δ is a *move relation*: $\delta \subseteq (Q \times (\Sigma \cup \{\lambda\}) \times \Gamma) \times (Q \times \Gamma^*)$

   For simplicity assume that the two stacks have the same stack alphabet, and that at each move both stacks can be altered.

2. Show that PDA2 are as powerful as Turing machines. That is, describe how an arbitrary Turing machine can be simulated by a PDA2. You do not need very gory detail here, but be precise.

Chapter 32

Turing Machines and Programs

In Chapter 30 we discussed programs as a model of computation, and defined "computable function," "decidable language," and "semidecidable language."

In Chapter 31 we defined Turing machines as a model of computation, and defined "Turing computable function," "Turing decidable language," and "Turing-semidecidable language."

This is too many terms to keep track of! But there is one simple message for this chapter:

> Programs and Turing machines can perform exactly the same computations.

And so "computable function," is the same as "Turing computable function," "decidable language," is the same as "Turing decidable language," and "semidecidable language" is the same as "Turing-semidecidable language."

So why do we bother to talk about both things, programs and Turing machines? We have been doing this kind of thing all through the book: when we you have two different definitions that turn out to be equivalent, it's a big plus: you can decide to work with whichever definition makes your life easier for the job at hand.

32.1 Programs Can Simulate Turing Machines

It is easy to see that Turing machines are no more powerful than programs, in the sense that: given any Turing machine there is a program that can simulate it.

32.1 Theorem. *Given a Turing machine M there is a C program p_M simulating M in the sense that, for every string x,*

- *if M halts in an accepting state on x, then p_M will halt and return 1 on x;*
- *if M halts in a non-accepting state on x, then p_M will halt and return 0 on x;*
- *if M fails to halt on x then p_M will fail to halt on x.*

Proof Idea. This is easy to prove; it is a simple programming problem to write a C program to simulate a Turing machine. ///

32.2 Turing Machines Can Simulate Programs

The result in this section is way more interesting!

32.2 Theorem. *Given a C program p there is a Turing machine M_p simulating p in the sense that, for every string x,*

- *if p returns 1 on input x then M_p halts on x in an accepting state;*
- *if p returns something other than 1 on input x then M_p halts on x in an non-accepting state;*
- *if p fails to return on input x then M_p fails to halt on input x.*

Proof Idea. The proof is a matter of doing a very careful (tedious) analysis of how Turing machines can simulate the things that programs can do (for example random access to memory). If you have studied machine architecture and seen how complex systems are built from very primitive components, you will not be surprised by this theorem. We will not distract ourselves here with the development. ///

On a philosophical level: it is remarkable that a single modification of finite automata (adding read/write memory) buys us universal computing power. So that is a fascinating insight into the nature of computation.

Theorem 32.1 and Theorem 32.2 are **extremely** convenient! It is deathly tedious to construct Turing machines even for trivial tasks. Bu by using the results above, if we ever have to convince someone (ourselves, for instance) that a certain job can be done by a Turing machine, all we have to do is argue that it can be done by a C program. In the other direction (and more importantly) these theorems facilitate arguments

about what *cannot* be computed. If we ever have to convince someone (ourselves, for instance) that a certain job cannot be done by any C program, all we have to do is argue that it cannot be done by any Turing machine. Since Turing machines are much simpler than high-level source code, it is easier to reason about the space of "all Turing machines."

32.3 Cashing In

Here we just record for clarity and reassurance that when we want to speak about computable functions or (semi-)decidable languages, we can think in terms of programs or in terms of Turing machines.

32.3 Theorem.

- *A function is computable if and only if it is Turing-computable.*
- *A language is semidecidable if and only if it is Turing-semidecidable.*
- *A language is decidable if and only if it is Turing-decidable.*

Proof. Each one of these follows immediately from Theorem 32.1 and Theorem 32.2. ///

32.3.1 Application: The Halting Problem for Turing Machines

It is sometimes more convenient to work with a version of the Halting Problem that talks about Turing machines rather than programs. There is no extra work to be done once we accept Theorem 32.2 and Theorem 32.1; here we just record the result for emphasis.

The Halting Problem for Turing Machines is the following problem.

> *Turing Machine Halting*
>
> INPUT: a Turing machine M and input string x
>
> QUESTION: does M halt on input x?

Remembering that our official definitions of decidable and undecidable are in terms of Turing machines, we record that following easy consequence of Theorem 32.4.

32.4 Theorem. *The Turing Machine Halting Problem is undecidable.*

Proof. To say the Turing Machine Halting Problem is undecidable is, by definition, to say that there is no Turing machine that decides it. But Theorem 32.1 says that if there were a Turing machine deciding Turing Machine Halting Problem then there would be a C program deciding it. But Theorem 32.4 says there isn't such a program. ///

Actually, the undecidability of the Halting problem is more traditionally stated purely in terms of Turing machines. We prefer the approach we've taken because (i) it's easier to work with programs, and (ii) nobody cares about Turing machines per se anyway.

32.4 Other Languages, Other Formalisms

There is nothing special about the choice of the C language in Theorems 32.1 and 32.2. No matter what standard programming language we might have chosen—Java, Python, Racket, Haskell, Prolog, Perl, *etc.*—the corresponding versions of Theorem 32.1 and Theorem 32.2 would be true.

> **Facts** All standard high-level programming languages, such as C, C++, Java, Python, Haskell, Scala, LISP, Fortran, ..., compute exactly the same partial functions from Σ_2^* to Σ_2^*.
>
> All standard high-level programming languages recognize exactly the same languages over Σ_2^*.
>
> All standard high-level programming languages decide exactly the same languages over Σ_2^*.

The above facts have been rigorously established in all the specific cases in the "such as ..." clause. But we don't state the above as a general theorem, just because we don't want to mathematically define what we mean by "standard high-level programming language."

So indeed from now on we will often be content to just say "program" rather than bother to say "C program."

Furthermore, many other formalisms—such as register machines, unrestricted grammars, the λ-calculus, to name a few—have been proposed to capture the intuitive notion of "computable." In every case they have been mathematically proven to have precisely the same computing power as Turing machines. Many such results were obtained by the pioneering logicians of the 1930's, such as Stephen Kleene, Alonzo Church, Emil Post, and Kurt Gödel.

Note that this observation itself says something interesting about the notion of computation. It says that the notion is *robust* in the sense that it is not sensitive to the various details that distinguish programming languages. The fact that so many ostensibly different definitions end up defining the same notion is strong evidence that that notion is truly fundamental. And it tells us that the notion of Turing machines, although much simpler than a modern programming language, really does capture all the complexity we can imagine in computing.

32.5 Summary

1. For every Turing machine there is an equivalent program.
2. For every program there is an equivalent Turing machine.
3. (a) Whenever we want to prove that a function is computable (or not computable) we may, if convenient, exhibit a program to compute it (or show that no such program can exist).
   (b) Whenever we want to prove that a language is semidecidable (or not semidecidable) we may, if convenient, exhibit a program to recognize it (or show that no such program can exist).
   (c) Whenever we want to prove that a language is decidable (or not decidable) we may, if convenient, exhibit a program to decide it (or show that no such program can exist).

In short, we can totally ignore Turing machines if we really want to!

(They do turn out to be useful sometimes, though . . .)

Chapter 33

Always-Halting Programs

33.1 Who Cares About Halting?

At this point one can imagine someone making the following complaint.

> *All this stuff about programs halting is fine, but in real life we don't really care about programs which don't halt. I want a theory of computability which takes* ***always-halting*** *programs as the fundamental objects of study, and investigates these directly.*

This is a perfectly reasonable point of view. There is one problem, though. *It is impossible to carry out such a research plan.* The reason for that is that it is impossible to recognize these programs! That is, the set of programs which always halt is not itself a well-enough behaved set of strings that we can tell whether a program is always-halting or not.

This is analogous to a lepidopterist deciding to specialize in a certain sub-species of butterfly, but not being able to recognize them even under a microscope.

We are going to show that there no decision procedure to answer whether or not an arbitrary program halts on all inputs. [1] The argument for this is illuminating: it gives a bit more insight into what is going on with the self-referential aspect of the proof that the halting problem is not decidable.

Formalizing the decision problem :

> *Always-Halting*
>
> INPUT: A program p
>
> QUESTION: Does p halt on all inputs?

[1] in fact there cannot even be an algorithmic way of *listing* all of the terminating programs. We explore this strengthening later.

By the way, don't fall into the trap of thinking, "Well, of course. If it is undecidable to test whether a program halts on a single input then of course is undecidable to test whether a program halts on all inputs." That reasoning is bogus. Remember that we are not confined to trying to answer these questions by simulating the program on inputs! It is perfectly plausible to imagine that in order for a program p to have to halt on all inputs the program p has to be simpler, or more uniform, in some sense, than an arbitrary program taken at random. Indeed for certain kinds of properties this does hold true: it is easier to decide a strict property than to decide a more lenient one.

But in this case that phenomenon doesn't happen. Indeed, always-halting is undecidable.

33.1.1 The Undecidability Theorem

33.1 Theorem. *There is no algorithm to solve the Always-Halting problem.*

Proof. Recall that we have a specified, using lexicographic order, an enumeration of all bitstrings as $x_0, x_1, x_2, \ldots$

Let

$$T \stackrel{\text{def}}{=} t_0, t_1, t_2, \ldots$$

be a list of all, and only, the bitstrings that are always-halting programs, written in lexicographic order. So this is a subsequence of the x_is.[2]

Now for sake of contradiction suppose that there were an algorithm to solve the Always-Halting problem. In that case, given any n we could compute exactly what program t_n is. We just run through the $x_0, x_1, x_2, \ldots$ testing each for whether they are always-halting, keeping a count of how many always-halting programs we've seen, and stopping when we get to the nth success.

Now construct the following program t^*.

```
on input x;
let i be the number such that x_i is x;
generate t_i and simulate it on x_i;
let y be the result of this computation;
return y1 (the result of appending 1 to the end)
```

The key point is that the result of t^* on x_i differs from the result of t_i on x_i. Note that this implies that t^* computes a different function from each one of the t_i (for example t^* doesn't compute the same function as t_{173} because they differ at least on input x_{173}).

[2] maybe t_0 is $x_{1,642}$ and t_1 is $x_{1,998}$ or whatever. The numbers don't matter!

But the description above *does* defines a program, because we have assumed that we can generate the ith terminating program for any i, and once it is generated we can proceed to simulate it.

Furthermore, the program t^* we've defined is itself terminating. This is because each t_i is guaranteed to be terminating, so the tests in the code above always return.

But this is a contradiction: since t^* does not compute the same function as any of the t_i, our original listing was not complete after all.

///

Summarizing: there can be no algorithm to test whether a given program halts on all inputs. Echoing the remarks at the end of Chapter 29:

If your job is to write a tool that detects the possibility of infinite loops in programs, you can only do a "best-effort" job. That is, your tool must either

- give some false positives: say that some programs p are always-halting even though they are not,
- give some false negatives: fail to recognize some programs that are in fact always-halting
- sometimes give up: not return a yes-no answer, for some inputs

Chapter 34

Rice's Theorem

Problems like testing whether a program halts on an input seem a bit artificial. In practice we are much more likely to want to do things like

- testing whether a program meets its specification,
- testing whether certain code optimizations preserve the input/output behavior of a program,
- testing whether two given programs compute the same function.

So a reasonable question at this point is whether the undecidability results and techniques we have so far tell us anything about these situations. They do, in fact. The news is bad, though.

34.1 Some Undecidable Questions about Programs

All of the following questions are undecidable.

- given a program, does it sort integers correctly?
- given a program, does correctly compute a Fast Fourier Transform?
- given a program, does it allow world access to the system password file?
- given a program, does it ever do a division by 0?
- given a program, does it halt on *all* of its inputs?

In this section we will prove a $\emptyset$single theorem, Rice's Theorem, that has all of the results above as special cases. This is a monster theorem which, informally, says that

any non-trivial functional property of programs is undecidable.

Our first job is to say carefully what those words mean.

Properties, Mathematically The way we precisely capture the notion of *property* of programs is to consider *sets* of programs.

This is the standard way to capture mathematically the somewhat philosophical notion of "property": the property of being red can be identified mathematically with the set of red things in the world; the property of being round can be identified mathematically with the set of round things in the world; the property of being even can be identified mathematically with the set of even numbers, and so on.

34.2 Functional Sets of Programs

Our goal is to show that any non-trivial functional set of programs is undecidable. As noted earlier, we need to make sure this has a precise meaning.

- *Non-trivial* means that we have to be talking about a property that holds of some programs but not by all programs.
- The subtle aspect of the claim is the fact that we speak of properties of the *functions* that programs compute rather than properties of programs as strings.

To appreciate the last point, note that there is typically no difficulty in deciding properties of programs as text objects (for example, do they have even length? Are they syntactically legal as C code? ...). But these questions are not questions about *the behavior of the process that the code generates when run.* It is this *behavior* that is the subject of Rice's Theorem.

So our first interesting job is to say mathematically what we mean when we say when something is a functional set of programs.

34.1 Definition. Let L be a set of strings, considered as programs. We say that L is a *functional set of programs* if whenever two programs p and q compute the same partial function, then $p \in L$ if and only if $q \in L$.

As a little metaphor, let's say that two programs p and r are *sisters* if they compute the same partial function. Then to say that L is functional is to say that when a program lies in L then all of its sisters must be in L as well.

So a set L will *fail* to be a functional set precisely if there exist programs p and q compute the same partial function yet $p \in L$ and $q \notin L$. That is, there is at least one pair of sister programs with one sister in L and the other sister not in L.

34.2 Examples. Each of the following sets of strings is a functional set of programs.

1. The set of all programs computing the identity function. For example, the program might reverse the input and then reverse it again, then print it out. Or, of course the program might simply echo its input.
2. The set of all programs that compute the empty function, that is, that never return an answer, on any input.
3. The set of programs computing increasing functions (with respect to lexicographic order on strings).
4. The set of programs that correctly perform prime factorization on integer inputs.
5. The set of all programs that halt on all inputs.
6. The set of all programs that halt on input "010100".

Be careful: a function set of programs is not necessarily the set of all programs computing a certain single function. In three of the above examples the functional set L was the set of programs for a non-singleton *set* of functions (which three?).

What do we mean by "non-trivial"?

Here are two rather dumb sets. But they *are* functional sets.

- $L = \emptyset$, and
- $L = {\Sigma_2}^*$.

Let's see that they are functional sets. Consider first $L = \emptyset$. We just have to show that it is impossible to have p and r computing the same function but with $p \in \emptyset$ and $r \notin \emptyset$. But obviously this is impossible since $p \in \emptyset$ never happens for any p at all! In the same way $L = \Sigma_2^*$ is a functional set, simply because there can never be any p which fails to be in L.

34.3 Check Your Reading. *Make sure you see the difference between (i) the set of all programs that compute the empty function and (ii) the empty set of programs.*

Make sure you see the difference between (i) the set of all programs that return an answer on all inputs and (ii) the set of all programs.

34.3 The Theorem

We are now ready to state Rice's Theorem, which addresses the question: *when are functional sets of programs decidable?* There are two easy examples: the sets $\emptyset$ and Σ_2^* are certainly decidable languages. Rice Theorem says, amazingly, that these are the only examples.

34.4 Theorem (Rice's Theorem)**.** *Let L be a functional set of programs. Further suppose that L is not the empty set, nor is L the set of all programs. Then L is not decidable.*

Proof Idea.[1] We are going to build a total computable function f such that

$$\text{for all } w, \quad w \in \mathsf{SelfHalt} \text{ if and only if } f(w) \in L$$

This will be enough to show that P is undecidable. Since: if we had a method for deciding membership in P we could use it to build a method for deciding SelfHalt. Namely, to determine whether a string w is in SelfHalt we could just apply f to w and ask whether the result is in P.

Note that f is a program transformer: given a string w, thought of as a program, it constructs the text $f(w)$ of a new program.

Here is the formal proof:

Proof. First, let p_0 be the following program: ``` ``while true do ;''. ``` Certainly p_0 computes the empty function, that loops forever on every input.

Next, note that since a language is decidable if and only its complement is decidable, it will do no harm to assume that L does not contain p_0. Otherwise we do do the proof below on the language $\overline{L}$ and show *it* to be undecidable.

Now we can let p_1 be some program in L whose function is *not* the empty function.

Note that it is just here that we have used the fact that L is not empty nor is it the set of all programs.

Summarizing the setup:

- p_0 is not in L and computes the empty function,
- p_1 is in L and computes some non-empty function.

Now we define a computable function f. The "p_1" referenced in the code below is the p_1 we referred to above: p_1 is in L and p_1 computes a non-empty function.

[1] the discussion in the "Proof Idea" recapitulates the introductory idea of Section 40. It is included here for readers who have not studied that section

for any string w:
if w is not a legal program, $f(w)$ is p_0.
otherwise $f(w)$ is code for the following program:

```
On input x:
simulate w on w;
simulate p1 on x
```

We see that f is a computable terminating function from strings to strings. What it *returns* is code for a program, $f(w)$; exactly *which* program $f(w)$ is depends on w. What can we say about this created program $f(w)$?

- Suppose w is in SelfHalt. Then on any input x, the program $f(w)$ does the same things as p_1 does on x (after the intial simulation of w on w, which takes time but does not change the fact that the subsequent computation is just that of p_1). So for such a w, the program $f(w)$ (i) halts on the same inputs as p_1 and (ii) gives the same answers as p_1. This means that the program $f(w)$ is in L, since $p_1 \in L$.
- Suppose w is not in SelfHalt. Then on any input x, the program $f(w)$ will loop forever. So for such a w, the program $f(w)$ never halts on any inputs. So the program $f(w)$ (i) halts on the same inputs as p_0 and (ii) gives the same answers as p_0. This means that the program $f(w)$ is not in L, since $p_0 \notin L$

We have shown that

$$w \in \mathsf{SelfHalt} \text{ if and only if } f(w) \in L$$

Since f is total and computable, L is not decidable, otherwise SelfHalt would be. ///

34.5 Examples. Each of the sets of strings from Example 34.2 is an undecidable language. Rephrasing these examples as decision problems, for practice, we now know that

1. The problem of deciding whether a given program computes the identity function is not decidable.
2. The problem of deciding whether a given program computes the empty function is not decidable.
3. The problem of deciding whether a given program computes an increasing function is not decidable.
4. The problem of deciding whether a given program computes prime factorization on integer inputs is not decidable.

5. The problem of deciding whether a given program halts on all inputs is not decidable.

6. The problem of deciding whether a given program halts on input "010100" is not decidable.

Since these all correspond to functional sets of programs, and are not trivial, Rice's Theorem immediately applies to them.

A non-example

It is interesting to note that our fundamental undecidable problem SelfHalt does not correspond to a functional set of programs.

The language $\mathsf{SelfHalt} = \{p \mid p[p]\downarrow\}$ is not a functional set of programs. Intuitively, this because a program q being in SelfHalt or not has to do with whether it halts on its own code as input, and there could easily be some other program that halts on the things as q halts on, but doesn't happen to halt on itself.

For a concrete example, suppose q is a program that halts on all strings of even length and no others. We may suppose that the program q itself has even length (we could always add a blank symbol at the end of the program). Now let q' be a program also that halts on all strings of even length and not others, but take q' to have odd length (again, easy to arrange). Then $q \in \mathsf{SelfHalt}$, but $q' \notin \mathsf{SelfHalt}$, yet q and q' compute the same partial function.

Perspective

Rice's Theorem is an all-purpose tool for showing sets of programs undecidable. In practice, the import of Rice's Theorem is that if you are interested in building a tool that will analyze programs in terms of their behavior, then your tool is destined to be incomplete.

If your job is to write a tool that detects *any* non-trivial behavior of programs, your tool must

- either say that some programs p have this behavior even though they don't ,
- or fail to recognize some programs that in fact have that behavior.

You will always have to rely on heuristics and approximations. By no means does it mean that you shouldn't work on such tools, they are important! It just means that you will never succeed perfectly.

34.4 Problems

274. ApplyRice1

For each language L below:

- say whether or not L is a functional set of programs
- say whether or not L is decidable

Note that "the domain of $\mathsf{pfn}(p)$" is simply the set of those strings x such that $\mathsf{pfn}(p)(x)$ is defined. This happens to be the set of those x such that p halts on x.

a) $\{p \mid 010 \in$ the domain of $\mathsf{pfn}(p)\}$.

b) $\{p \mid$ the domain of $\mathsf{pfn}(p)$ has at most 10 strings$\}$.

c) $\{p \mid$ the domain of $\mathsf{pfn}(p) = \Sigma_2^*\}$.

d) $\{p \mid$ the domain of $\mathsf{pfn}(p)$ is regular$\}$.

e) $\{p \mid p$ halts on $p.\}$

f) $\{p \mid p$ is the shortest program computing $\mathsf{pfn}(p)\}$.

g) $\emptyset$.

h) $\{p \mid p$ halts on all inputs. $\}$

275. ApplyRice2

For each language L below:

- say whether or not L is a functional set of programs
- say whether or not L is decidable

Note that "the domain of $\mathsf{pfn}(p)$" is simply the set of those strings x such that $\mathsf{pfn}(p)(x)$ is defined. This happens to be the set of those x such that p halts on x.

a) $\{p \mid p$ has even length$\}$

b) $\{p \mid$ the domain of $\mathsf{pfn}(p)$ has at least 10 strings$\}$.

c) $\{p \mid$ the domain of $\mathsf{pfn}(p) \neq \emptyset\}$.

d) $\{p \mid \mathsf{pfn}(p)$ is the identity function$\}$.

e) $\{p \mid \mathsf{pfn}(p)$ is a constant function$\}$.

f) $\{p \mid \mathsf{pfn}(p) = \mathsf{pfn}(q)\}$, where q is a fixed program that returns 0 on odd length strings and loops forever on even-length strings

g) $\{p \mid$ the domain of $\mathsf{pfn}(p) = \emptyset\}$

h) $\{p \mid$ the domain of $\mathsf{pfn}(p)$ is finite. $\}$

i) $\{p \mid$ there is some shorter program q computing the same function as p.$\}$

j) $\{p \mid$ the domain of $\mathsf{pfn}(p)$ is decidable. $\}$

276. FunctionalClosure

Here is a series of true/false questions to test your understanding of "functional set of programs". If you answer "false" to a give problem, give a concrete counterexample.

a) If P_1 and P_2 are functional sets of programs then $P_1 \cup P_2$ is a functional set of programs.

b) If P_1 and P_2 are functional sets of programs then $P_1 \cap P_2$ is a functional set of programs.

c) If P is a functional set of programs and $Q \subseteq P$ then Q is a functional set of programs.

d) If P is a functional set of programs and $P \subseteq Q$ then Q is a functional set of programs.

e) If P is a functional set of programs then $\Sigma^* - P$ is a functional set of programs then

f) If P is a functional set of programs, with $P \neq \emptyset$ and $P \neq \Sigma^*$, and p and q are in P, then $\mathsf{pfn}(p) = \mathsf{pfn}(q)$

277. FunctionalFunctionSet

Let $\mathcal{F}$ be some set of partial computable *functions.* Let L be the set of programs that compute some function in $\mathcal{F}$:

$$L = \{p \mid \mathsf{pfn}(p) \in \mathcal{F}\}$$

True or false? L is a functional set of programs.

True or false? Every functional set of programs arises in this way. (In other words: if L is a functional set of programs then there is some set $\mathcal{F}$ of partial computable functions such that

$$L = \{p \mid \mathsf{pfn}(p) \in \mathcal{F}\}$$

278. RiceProof

To exercise your understanding of the proof of Rice's Theorem, see if you can pinpoint exactly which places in the proof we used the fact that L was a functional set of programs (as opposed to some arbitrary set of programs).

To be completely precise: point out one or more sentences in the proof that are false statements if we do not assume L is a functional set of programs.

Chapter 35

Decidable Languages

From now on we are going make heavy use of our results that Turing machines and programs are equivalent: we will speak about programs rather than Turing machines whenever we can.

Remember the definition of decidable language, from Definition 30.5:

Definition. A language $L \subseteq \Sigma_2^*$ is *decidable* if there is a program p with

$$\mathsf{pfn}(p)(x) = \begin{cases} 1 & \text{if } x \in L \\ 0 & \text{if } x \notin L \end{cases}$$

If a language is not decidable we say it is *undecidable*.

Recall also that it is convenient to use the term "decision procedure" for a programs like those, that halt on every input and return either 0 or 1.

35.1 Properties of the Decidable Languages.

Let's collect some easy abstract properties of decidable languages.

You have already proved the following result if you did Problem DecCompl in Chapter 31. It is easy to prove using Turing machines, and also easy to prove using programs.

35.1 Theorem. *If A is decidable then $\overline{A}$ is decidable.*

Proof. Let p be a decision procedure accepting A. Build program p' as follows: p' simulates p exactly, except that whenever p is ready to return 1, p' returns 0, and vice versa. We claim that p' decides $\overline{A}$. This is because for any string w, w is accepted by p if and only if w is rejected by p' (note that the fact that p halts on all inputs is used here!). Therefore p' is a decision procedure, and therefore witnesses the decidability of $\overline{A}$. ///

Note carefully that the program p must halt on all inputs in order for the construction above to have the effect of complementing the original language. If p didn't halt on all inputs, the construction above wouldn't work. This should remind you of complementing *DFAs* (and how the simple technique there does not work for *NFAs*).

These next results are sort-of-easy to prove using Turing machines, but super-easy to prove using programs.

35.2 Theorem. *The decidable languages are closed under the operations of*

1. *intersection,*
2. *union,*
3. *concatenation, and*
4. *Kleene star*

Proof.

1. Suppose A and B are decidable languages; we wish to show that $A \cap B$ is decidable. Let p_A be a decision procedure deciding A, and let p_B be a decision procedure deciding B. Our goal is to show that there exists a decision procedure r deciding $A \cap B$. We exhibit pseudocode for such a program as follows:

   on input w;
   run p_A on w;
   run p_B on w;
   if each of these runs returns 1, then return (1) else return (0)

   To defend the claim that this program suffices we must argue that r is a decision procedure and that it decides $A \cap B$. The fact that r is a decision procedure follows from the facts that both p_A and p_B are decision procedures, and so when they are run on w they are guaranteed to return. The fact that r decides $A \cap B$ is immediate.

2. Suppose A and B are decidable languages; we wish to show that $A \cup B$ is decidable. Let p_A be a decision procedure deciding A, and let p_B be a decision procedure deciding B. Our goal is to show that there exists a decision procedure r deciding $A \cup B$. We exhibit pseudocode for such a program as follows:

   on input w;
   run p_A on w;
   run p_B on w;
   if either of these runs returns 1, then return (1) else return (0)

   To defend the claim that this program suffices we must argue that r is a decision procedure and that r decides $A \cup B$. The fact that r is a decision procedure follows from the facts that both p_A and p_B are decision procedures, and so when they are run on w they are guaranteed to return. The fact that r decides $A \cup B$ is immediate.

3. Next is concatenation; this is a little more interesting. Suppose A and B are decidable languages; we wish to show that AB is decidable. Let p_A be a decision procedure deciding A, and let p_B be a decision procedure deciding B. Our goal is to show that there exists a decision procedure r deciding AB. We exhibit pseudocode for such a program as follows:

   on input w;
   consider in turn each pair of strings w_1, w_2 such that $w_1 w_2 = w$:
      run p_A on w_1;
      run p_B on w_2;
      if each of these runs returns 1, then return (1)
   // If we get here we have failed...
   return (0).

   To defend the claim that this program suffices we must argue that r is a decision procedure and that r decides AB. The fact that r is a decision procedure follows from the facts that both p_A and p_B are decision procedures, and that given w there are only finitely many pairs of strings w_1, w_2 such that $w_1 w_2 = w$. (How many are there precisely, in terms of $|w|$?). That is, the for-loop is guaranteed to have only finitely many iterations. The fact that r decides AB is straight from the definition of concatenation: a string w is in AB if and only if there are strings $w_1 \in A$ and $w_2 \in B$ such that $w = w_1 w_2$.

4. Finally, for Kleene star: Suppose A is a decidable language; we wish to show that A^* is decidable. Let p_A be a decision procedure deciding A. Our goal is to show that there exists a a decision procedure program r for A^*. We exhibit pseudocode for such a program as follows:

on input w;
consider in turn each sequence of strings $w_1, w_2, \ldots w_n$ such that $w_1 w_2 \ldots w_n = w$:

run p_A on w_1;
run p_A on w_2;
...
run p_A on w_n;
if each of these runs returns 1, then return (1)

// If we get here we have failed...
return (0).

To defend the claim that this program suffices we must argue that r is a decision procedure and that r decides A^*. The fact that r is a decision procedure follows from the fact that p_A is a decision procedure, and that given w there are only finitely many sequences of strings $w_1, w_2, \ldots, w_n$ such that $w_1 w_2 \ldots, w_n = w$. That is, the for-loop is guaranteed to have only finitely many iterations, and each for-loop body has only finitely many statements. The fact that r decides A^* is straight from the definition of asterate: a string w is in A^* if and only if there are strings $w_1, w_2, \ldots w_n$ such that each w_i is in A and $w_1 w_2 \ldots w_n = w$.

///

35.2 Extensionality

When we say that a language is, or is not, decidable, we are making a *mathematical* statement about the language, as opposed to a *psychological* statement about how we human being understand the language. This is a very important point, but one that is easy to get confused about.

35.3 Example. A very famous unsolved mathematical problem is Goldbach's Conjecture, which states: *every even number greater than 2 can be written as the sum of two prime numbers.* For example 4 = 2+2; 6 = 3+3; 8 = 5+3; *etc.*.

No one has found a counterexample to this, yet no one has been able to prove it to be universally true.

Now consider the following function

$$f(x) = \begin{cases} 0 & \text{if Goldbach's conjecture is true} \\ 1 & \text{if Goldbach's conjecture is false} \end{cases}$$

The f is a computable function. Why? Because the output doesn't depend on the input x, so it is a constant function, and so there certainly is a computer program that computes it.

You may object, "but we don't know whether it is the constant function 0 or the constant function 1!" But that doesn't matter. The point is that regardless of our human state of knowledge at the moment[1], we have shown that **there exists** an algorithm for f. The fact that we don't happen to know what algorithm is the correct one is not relevant. This is what we meant above when we said that decidability is a mathematical notion, not a psychological one.

35.4 Example. For another example, consider the following two problems, taken from [Rog67].

> *Exactly n 5s*
>
> INPUT: The binary encoding of a natural number n
>
> QUESTION: Does there exist a run of *exactly* n consecutive 5's in the decimal expansion of π?

At present, it is not known whether this problem is decidable. But constrast this with the following problem.

> *At Least n 5s*
>
> INPUT: The binary encoding of a natural number n
>
> QUESTION: Does there exist a run of *at least* n consecutive 5's in the decimal expansion of π?

This problem is decidable. You don't need to know anything about π to know this.

Why? Observe that there are two possibilities: either (i) there are arbitrarily long runs of consecutive 5's in the decimal expansion of π, or (ii) there is a longest such run. This much is clear.

Suppose (i) is the case. Then the following is an algorithm for the *At Least n 5s* problem:

```
on input n, return 1.
```

Suppose (ii) is the case. Let k be the length of the longest run of consecutive 5's in the decimal expansion of π. Then the following is an algorithm for the *At Least n 5s* problem:

```
on input n, if n ≤ k, return 1, else return 0.
```

[1] 2020 Mar 17 12:59:04 as I write this

You may object, "but we don't know whether (i) or (ii) is true, and even if (ii) is true we don't know what k is!" But as in the previous example, that doesn't matter. **There exists** an algorithm for the *At Least n 5s* problem, and the fact that we don't happen to know what algorithm is the correct one is not relevant.

What you should take away from these examples is a deeper understanding of what undecidability means. It is not about how hard certain problems are *for us to understand,* it is about the *inherent complexity* of certain languages as mathematical objects.

If the above discussion is troubling, answer the following question

35.5 Check Your Reading. *True or false: the number*

$$2^{100} - 17$$

has a prime factorization.

I hope you answered True. And I hope you didn't feel like you needed to *find* that factorization before being confident in your answer. We know that the assertion that that number has a prime factorization is true even without being able to say what it is. In the same way, we can sometimes say that certain decision problem has an algorithm, without being able to say what it is.

35.3 Problems

279. DecSubset

1. Prove or disprove: If L is decidable and $K \subseteq L$ then K is decidable.
2. Prove or disprove: If L is decidable and $L \subseteq K$ then K is decidable.

280. DecClosure

We only have a couple of examples of undecidable languages so far. We can exploit the closure properties of decidable languages to conclude that languages are *undecidable* ... sometimes.

Let D be a decidable language and let N be a language which is not decidable.

1. Suppose X is a language such that $X = D \cap N$. Does it follow that X is necessarily decidable? If so, say why. Does it follow that X is necessarily non-decidable? If so, say why. If your answers to the two previous questions was no, name a decidable D and non-decidable N satisfying $X = D \cap N$ with X non-decidable, and name a decidable D and non-decidable N satisfying $X = D \cap N$ with X decidable.
2. Suppose X is a language such that $N = D \cap X$. Does it follow that X is necessarily decidable? If so, say why. Does it follow that X is necessarily non-decidable? If so, say why. If your answers to the two previous questions was no, name a decidable D and non-decidable N satisfying $N = D \cap X$ with X non-decidable, and name a decidable D and non-decidable N satisfying $N = D \cap X$ with X decidable.
3. Suppose X is a language such that $D = N \cap X$. Does it follow that X is necessarily decidable? If so, say why. Does it follow that X is necessarily non-decidable? If so, say why. If your answers to the two previous questions was no, name a decidable D and non-decidable N satisfying $D = N \cap X$ with X non-decidable, and name a decidable D and non-decidable N satisfying $D = N \cap X$ with X decidable.
4. Suppose X is a language such that $D = \overline{X}$. Does it follow that X is necessarily decidable? If so, say why. Does it follow that X is necessarily non-decidable? If so, say why. If your answers to the two previous questions was no, name a decidable D satisfying $D = \overline{X}$ with X non-decidable, and name a decidable D satisfying $D = \overline{X}$ with X decidable.

5. Suppose X is a language such that $N = \overline{X}$. Does it follow that X is necessarily decidable? If so, say why. Does it follow that X is necessarily non-decidable? If so, say why. If your answers to the two previous questions was no, name a non-decidable N satisfying $N = \overline{X}$ with X non-decidable, and name a non-decidable N satisfying $N = \overline{X}$ with X decidable.

281. Dec1x

Prove that if A is decidable then the language

$$A_1 \stackrel{\text{def}}{=} \{1x \mid x \in A\}$$

is decidable.

282. Undec1x

Prove or disprove: if A is undecidable then the language

$$A_1 \stackrel{\text{def}}{=} \{1x \mid x \in A\}$$

is undecidable.

283. 2ADec

Prove that if A is decidable then the language

$$2A \stackrel{\text{def}}{=} \{xx \mid x \in A\}$$

is decidable.

Note that $2A$ is not the same language (usually) as AA.

284. RevDec

Prove that if A is decidable then the language

$$A^R \stackrel{\text{def}}{=} \{x^R \mid x \in A\}$$

of reverses of strings in A, is decidable.

Chapter 36

Semi-Decidable Languages

Remember the definition of semidecidable language, from Definition 30.4:

Definition (Semidecidable Language)**.** A language $L \subseteq \Sigma_2^*$ is *semi-decidable* if and only if it is $L(p)$ for some program p.

This is the same as saying:

A language $L \subseteq \Sigma_2^*$ is semidecidable if there is a program p such that

- whenever $x \in L$, $\mathsf{pfn}(p)(x) = 1$, and
- whenever $x \notin L$, $\mathsf{pfn}(p)(x)$ is undefined or is a value $\neq 1$

Representative Examples

Here are two examples of semi-decidable languages.

36.1 Example.

> *CFG Ambiguity*
> INPUT: a *CFG G*
> QUESTION: is *G* ambiguous

To show that *CFG* Ambiguity is semi-decidable we show that there is a program p such that when G is ambiguous $p[G]$ halts with 1 and when G is not ambiguous $p[G]$ fails to halt. Here is pseudocode for such a p.

```
// Checks a CFG for ambiguity
read (encoding of) grammar G from stdin;
while true :
```

```
systematically generate all parse trees G can make;
whenever a tree T is built, with frontier w, see if w is generated by any
previous different tree T';
if so return 1;
```

So p is a pretty dumb program, but you should be able to see that if an ambiguous grammar is presented as input to p then p is guaranteed to eventually realize that p is ambiguous and halt with return 1; while if p is given a non-ambiguous grammar it will never halt. Thus $L(p)$ is the set of ambiguous grammars.

Note that p is not a decision procedure! Indeed, p never returns 0 on any grammars! If G is a CFG which is not ambiguous and which generates infinitely many strings, p will fail to halt on that grammar.

36.2 Example. The second example is our old friend the Halting Problem, which you might feel is not as natural but turns out to be the the most fundamental semi-decidable problem (in a sense that can be made precise in terms of reducibility).

The example is important enough so that we record it as a theorem.

36.3 Theorem. *The Program Halting Problem is semi-decidable.*

Proof. Consider the following pseudocode.

```
On input p and x:
Simulate p on input x;
If and when this simulation halts, return 1
```

Let h be a program that implements this pseudocode.

Then h semidecides the Halting Problem, since, if p halts on x, $h[x] \downarrow 1$, while if p doesn't halt on x, $h[x]$ does not return 1 (it happens that in fact $h[x]$ loops forever but that doesn't matter here). ///

36.1 Decidable versus Semi-Decidable

You might be puzzled by the terminology *semi*-decidable, since it suggests that a semi-decidable is "halfway-decidable" somehow. Let's explore that.

Decidable Implies Semi-Decidable

For starters we observe that claiming a language to be decidable is at least a strong as claiming it to be semidecidable.

36.4 Lemma. *If A is decidable then A is semi-decidable.*

Proof. This is immediate from the definitions. Let p be a decision procedure for A, that is

$$\mathsf{pfn}(p)(x) = \begin{cases} 1 & \text{if } x \in A \\ 0 & \text{if } x \notin A \end{cases}$$

This implies immediately that A is $L(p)$.

///

Important! If we ever say that a language A is semi-decidable , this should not be taken to imply that A is not decidable. If all one says is "A is semi-decidable " then this means "A is semi-decidable; it may or may not also be decidable."

Semi-Decidable Does Not Imply Semi-Decidable

We already observed this: the Halting Problem is not decidable (Theorem 29.2) but it is semi-decidable (Theorem 36.3).

Semi-Decidable plus Co-Semi-Decidable Implies Decidable

Complementation is at the heart of the relationship between decidable and semi-decidable. The following is the most important fact relating decidable and semi-decidable .

36.5 Theorem. *A is decidable if and only if both A and $\overline{A}$ are semi-decidable.*

Proof. For the the left-to-right direction of the theorem, suppose A is decidable. Then A is semi-decidable by Lemma 36.4. Furthermore we have that $\overline{A}$ is decidable as well by Theorem 35.1. So by Lemma 36.4 again, $\overline{A}$ is semi-decidable.

For the other direction, suppose that A is semi-decidable and $\overline{A}$ is semi-decidable . In order to show that A is decidable we will build a decision procedure p for A. We have by hypothesis two programs p_1 and p_2 which accept A and $\overline{A}$ respectively. Build p as follows:

On input w
For $t = 0$ to ∞
if t is even simulate p_1 on w for t steps ; if $p_1[w] \downarrow 1$ then return 1
if t is odd simulate p_2 on w for t steps ; if $p_2[w] \downarrow 1$ then return 0

This p returns either 0 or 1 on *all* inputs w because we are guaranteed that exactly one of p_1 or p_2 will return 1 on w. So we just need to note that the inputs where p returns 1 are precisely the inputs where p_1 halts, that is, the strings in A. ///

36.2 Semi-Decidable Decision Problems

Semidecidability is a property of a *language.* But in the usual way, since decision problems are really just a different manner-of-speaking for describing languages, we will often refer to a decision problem as being semi-decidable, when what we mean is that the associated language is a semi-decidable language.

36.6 Example. The language

$$\{G \mid G \text{ is an ambigous context-free grammar}\}$$

is a semi-decidable language.

This is what Example 36.1 showed.

Of course, for all we know based just on what we have proven so far, the problem of testing that a CFG is ambiguous might actually be decidable; the argument in Example 36.1 leaves open the possibility that some other, more clever algorithm, could be written which would be a decision procedure. As a matter of fact, though, this is not the case, as we will prove later.

36.7 Example. The following problem is semi-decidable.

Halt On λ

INPUT: A C program p

QUESTION: Does p terminate normally on input λ?

What we mean here is that the language

$$\{p \mid p \text{ is a program that terminates normally on } \lambda\}$$

is a semi-decidable language.

This amounts to the claim that one can write a program *TermTst* which halts with success on input p if and only if p codes a program which would terminate on λ. To emphasize: if given an input program p that does not halt normally on λ, *TermTst* is allowed to return some answer other than 1 OR to not return and answer at all.

Here is simple pseudocode for such a semidecision procedure *TermTst*:

> *on input p;*
> *(decode p as a program and) simulate p on* λ*;*
> *if and when this simulation halts, return 1.*

For emphasis, we note that if the program p does not terminate on input λ, our program *TermTst* as we have described it will not halt on input p. This simply means that the argument above is an argument for the *semi*-decidability of the given problem, and does *not* count as an argument that the problem is decidable.

By the way, there is nothing in this example that depends on the input being the empty string, we could have chosen *any* bitstring and had exactly the same argument.

There is perhaps a danger in the above example that you may confuse *TermTst* with the C program being tested. Don't! *TermTst* is a program-testing program; it processes *any* string you give to it.

In fact, Rice's Theorem (Chapter 34) tells us that the *Halt On* λ problem is undecidable. So this is another example of a problem that is semi-decidable but not decidable.

The Dovetailing Trick

Before giving some more, less-obvious, examples of semidecidability, we describe a certain trick which we will use often. It is routine as a programming device but it is worth describing carefully once and for all so that proofs that use it are clear.

Often we have two or more computations we want to simulate and combine in some way. It might be that we have several (even infinitely many) programs that we want to run on a string. It might be that we have several (even infinitely many) strings that we want to run on a certain program on. Or it might be both: we have several programs and several strings and we want to run each program on each string.

We need to be careful not to do these computations *in sequence* due to the fact that one of them might not terminate, and so the others would get "starved" and we couldn't inspect them. The trick, of course, is to do the computations, intuitively, in parallel. Our present job is to make that precise, without having to develop a complex notion of how to implement processes or threads, but keeping to simple programming intuitions.

If p is a program, x is an input string, and n is a natural number, let us agree that there is a well-defined notion of running the computation $p[x]$ for n steps. Though nothing we do will really depend what exactly we count as a step, for concreteness, let us assume that our programs have been compiled to some sort of assembly code, and a "step" is the execution of one such instruction.

The construction The setting is: we have a sequence of programs $p_1, p_2, \ldots p_k$ and a sequence inputs $x_1, x_2, \ldots$, and we would like to run them all is such a way that no $p_i[x_i]$ starves other computations from running, and each $p_i[x_i]$ gets all the number of steps it requires. There are finitely many programs; there can be either finitely many or infinitely many inputs.[1]

When we refer to a "dovetailing" computation of $p_1, p_2, \ldots p_k$ on inputs $x_1, x_2, \ldots$ we mean the following. It is a (potentially infinite) computation that that runs in stages. Specifically:

> At each stage s,
> run $p_1[x_1], p_1[x_2], \ldots, p_k[x_s]$
> each for s steps, or until it halts, whichever comes first.

Now, this is written so as to be maximally general, and not have special cases as to whether there are finitely or infinitely many input strings. So let us agree that if there are only finitely many, say n, strings that we care about then just ignore the references to strings x_{n+1}, x_{n+2}, etc.

The key things to note are

- at each stage we only do finitely much computation, and yet
- for each program p_i and for each string x_j and for each number of steps n, we do simulate $p_i[x_j]$ for at least n steps (namely, at any stage s such that s is as big as the maximum of i, j, n)

Here is an example of this technique in action.

36.8 Example (Program Non-Emptiness)**.** Recall the question, "does a given program p halt on any strings at all?" Note that to say that there exists an input that program p halts on is the same as saying that $\mathsf{pfn}\, p$ is not the empty function.

Written as a decision problem

> *Program Non-emptiness*
> INPUT: a program p
> QUESTION: is $\mathsf{pfn}\, p \neq \emptyset$?

[1] Actually it is not hard to tweak this technique to handle infinitely many programs. But it would complicate the notation, and we almost never need that generality, so we skip it.

Written as a language:

$$\mathsf{NonEmp} \stackrel{\text{def}}{=} \{p \mid p \text{ is a program and } \mathsf{pfn}\, p \neq \emptyset\}$$

We showed this earlier to be to be undecidable. Here we show it to be semi-decidable. It will be an application of the dovetail trick (where we have only one program, but infinitely many strings).

Proof. Here is a program that semi-decides NonEmp.

> *on input p;*
> *Using dovetailing, consider each computation* $p[x_1], p[x_2], \ldots p[x_n] \ldots$
> *as* x_i *ranges over all strings.*
>
> *If and when any of these halts,*
> *return 1*

If there are any strings x such that $p[x]\downarrow$ then the above procedure will discover this, and return 1. If there are no strings x such that $p[x]\downarrow$ then the above procedure will run forever.

That is, this program returns 1 on input p if and only if $p \in \mathsf{NonEmp}$, as desired. ///

36.3 Reducibility and Semidecidability

In fact we can use reducibility to tell us things about semidecidability as well.

36.9 Theorem. *Suppose A is reducible to B. If B is semi-decidable then A is semi-decidable.*

Proof. Suppose that $A \preceq B$ via function f; let p_f be a program that computes f. Now let p be a program with the property that $x \in B$ iff $p[x]\downarrow 1$. The the following is a program q has the property that $w \in A$ iff $q[w]\downarrow 1$.

> on input w:
> compute $p_f[w]$; run p on the result
> (return that answer)

Yes, that's the same code as in the proof of Theorem 40.8. This time the program doesn't necessarily halt on all w, but we don't need that this time. If $w \in A$ then $p_f[w]$ will be in B, and so p will return 1; while if $w \notin A$ then $p_f[w]$ will be not be in B, and so p will either loop forever or return something other than 1. So we have shown A to be semi-decidable. ///

The following is immediate from the theorem.

36.10 Corollary. *Suppose A is reducible to B. If A is not semi-decidable then B is not semi-decidable.*

36.4 Emptiness Is Not Semi-Decidable

That result about Program NonEmptiness now gives us more information about Program *Emptiness.*

36.11 Theorem. Emp *is not semi-decidable.*

Proof. We have seen that the complement of Emp, namely, $\mathsf{NonEmp} \stackrel{\text{def}}{=} \{p \mid \mathsf{dom}(\mathsf{pfn}(p)) \neq \emptyset\}$, is semi-decidable. So if Emp were semi-decidable, we could conclude that it was actually decidable, using the fundamental fact (Theorem 36.5) relating decidable and semi-decidable. ///

36.5 Beyond Semi-decidable

Some languages are not even semi-decidable.

Recall that $\mathsf{Emp} = \{p \mid p \text{ is a program with } \mathsf{pfn}\, p = \emptyset\}$.

- This language in not decidable (we can use Rice's Theorem to show this.)
- But the complement of Emp is NonEmp, which we just showed to be semi-decidable.

This implies that Emp is not only not decidable, it is not even semi-decidable. Since: if it were, then, given that its complement NonEmp *is* semi-decidable, that would make Emp decidable! (For that matter, it would make NonEmp decidable too.)

This is a strange phenomenon. If you have a language X that you know to be undecidable, then a way to show that X is even *more* complex, that is not even semi-decidable, is to show that the complement, $\overline{X}$, is "not so bad", namely, is semi-decidable.

Finally, you might be wondering if things can be even worse than that, namely, whether there can be languages X that are not semi-decidable, yet whose complements are not semi-decidable either. The answer is yes, and in fact there is a natural example: the language consisting of the always-halting programs. We don't have the tools to prove that fact yet; we do provide a little discussion in Remark 37.5.

36.6 Closure Properties of the Semi-Decidable Languages

We already saw that the decidable languages are closed under most of the set-theoretic operations one would want: complement, union, intersection, concatenation, and Kleene star.

The semi-decidable languages are closed under these as well (**except** for complement!) but since semi-decidable languages are accepted by programs that are not guaranteed to halt, the proofs are more delicate than the ones for decidable languages.

The next theorem collects the main results.

36.12 Theorem. *The semi-decidable languages are closed under the operations of*

1. *intersection,*
2. *union,*
3. *concatenation, and*
4. *Kleene closure (the star operation).*

Proof. 1. Suppose A and B are semi-decidable languages; we wish to show that $A \cap B$ is semi-decidable. Let p_A be a program such that $L(p_A)$ is A, and let p_B be a program such that $L(p_B)$ is B. Our goal is to show that there exists a program r such that $L(r)$ is $A \cap B$. We exhibit pseudocode for such a program as follows:

> *on input w;*
> *run p_A on w;*
> *run p_B on w;*
> *if each of these runs halts with 1, then return (1),*

To defend the claim that this program suffices we must argue that $L(r)$ is $A \cap B$. Note that we are making no claims that r halts on all inputs! We only need to establish that if w is indeed in both A and B then program r will halt with return (1), and if w is not in both A and B then r will not halt with 1. This is clear from inspection of the code.

2. Suppose A and B are semi-decidable languages; we wish to show that $A \cup B$ is semi-decidable. Let p_A be a program such that $L(p_A)$ is A, and let p_B be a program such that $L(p_B)$ is B. Our goal is to show that there exists a program r such that $L(r)$ is $A \cup B$. *Here we have to be clever.* We cannot simply use the same pseudocode as we did for the decidable-language case. The problem is that if we run p_A first on an input w then this might never return, yet w might be in

B, hence in $A \cup B$, but we never get a chance to verify this. The solution is easy though: we simply run dovetail p_A and p_B :

on input w;

dovetail $p_A[w]$ and $p_B[w]$;
if and when either of these runs halts with 1, then return (1)

To defend the claim that this program suffices we must argue that $L(r)$ is $A \cup B$. This merely amounts to observing that r will halt with 1 w precisely if at least one of $p_A[w]$ or $p_B[w]$ halts with 1.

3. Next is concatenation. Suppose A and B are semi-decidable languages; we wish to show that AB is semi-decidable. Let p_A be a program such that $L(p_A)$ is A, and let p_B be a program such that $L(p_B)$ is B. Our goal is to show that there exists a program r such that $L(r)$ is AB. We exhibit pseudocode for such a program as follows:

on input w;
Using dovetailing, consider each pair of strings w_1, w_2 such that $w_1 w_2 = w$:
run p_A on w_1;
run p_B on w_2;
if each of these runs halts with 1, then return (1)

To defend the claim that this program suffices we must argue that $L(r)$ is AB. But as in the decidable-language case the fact that $L(r)$ is AB is straight from the definition of concatenation: a string w is in AB if and only if there are strings $w_1 \in A$ and $w_2 \in B$ such that $w = w_1 w_2$.

Note the need for dovetailing here. For any *given* pair w_1, w_2 such that $w = w_1 w_2$, we can test whether this pair witnesses w to be in AB by virtue of having $w_1 \in A$ and $w_2 \in B$ without having to dovetail these tests. Since: if the simulation of p_A on w_1 fails to halt then that's fine, $w_1 \notin A$ and we don't expect to accept this pair. But we need to be able to check *all* pairs w_1, w_2 such that $w = w_1 w_2$, without having some "bad" pair prevent us from getting to test a potentially "good" pair. So we test all the ways that w might be divide in two simultaneously.

4. Finally, for Kleene closure. Suppose A is a semi-decidable language; we wish to show that A^* is semi-decidable. Let p_A be a program such that $L(p_A)$ is A. Our goal is to show that there exists a program r such that $L(r)$ is A^*. We exhibit pseudocode for such a program as follows:

on input w;
Using dovetailing, consider each sequence of strings $w_1, w_2, \ldots w_n$ such that $w_1 w_2 \ldots w_n = w$:

run p_A *on* w_1*;*
run p_A *on* w_2*;*
...
run p_A *on* w_n*;*
if each of these runs halts with 1, then return (1)

The argument that this suffices should by now be familiar to you. You should note again the need for dovetailing for essentially the same reason as for the case of concatenation.

///

Note carefully that we did not claim the result above that if A is semi-decidable then $\overline{A}$ is semi-decidable. If it *were* true that whenever A were semi-decidable then $\overline{A}$ were semi-decidable as well, then it would follow that whenever A is semi-decidable then in fact A is decidable. And that is certainly not true.

36.7 Problems

285. LangVsPfn

Give a concrete example showing that we can have $L(p) = L(q)$ yet $\mathsf{pfn}(p) \neq \mathsf{pfn}(q)$

286. SDSubset

1. Prove or disprove: If L is semi-decidable and $K \subseteq L$ then K is semi-decidable.
2. Prove or disprove: If L is semi-decidable and $L \subseteq K$ then K is semi-decidable.

287. UseHypotheses

Let L_1 and L_2 be languages. We know that if we have the following conditions then we can conclude that both L_1 and L_2 are decidable.

(i) L_1 and L_2 are each semi-decidable,

(ii) $L_1 \cup L_2 = \{0,1\}^*$, and

(iii) $L_1 \cap L_2 = \emptyset$.

In this problem we want to really drive home how those conditions work.

1. Give a concrete example to show that if L_1 and L_2 satisfy (i) and (ii), but not (iii), then L_1 is not necessarily decidable.
2. Give a concrete example to show that if L_1 and L_2 satisfy (i) and (iii), but not (ii), then L_1 is not necessarily decidable.

288. Taxonomy

For each of the following situations, either give an example of a language L fitting the description, or explain why the situation is impossible.

1. L is decidable and $\overline{L}$ is decidable.
2. L is decidable and $\overline{L}$ is semi-decidable but not decidable.
3. L is decidable and $\overline{L}$ is not semi-decidable.
4. L is semi-decidable but not decidable and $\overline{L}$ is decidable.
5. L is semi-decidable but not decidable and $\overline{L}$ is semi-decidable but not decidable.
6. L is semi-decidable but not decidable and $\overline{L}$ is not semi-decidable.
7. L is not semi-decidable and $\overline{L}$ is decidable.
8. L is not semi-decidable and $\overline{L}$ is semi-decidable but not decidable.

289. SDPartition

Suppose $L_1, L_2, \ldots L_k$ are semi-decidable languages over the alphabet Σ_2 such that

1. $L_i \cap L_j = \emptyset$ when $i \neq j$, and
2. $L_1 \cup L_2 \cup \ldots L_k = {\Sigma_2}^*$.

Prove that L_1 is decidable. Indicate clearly how each of the hypotheses (1) and (2) are used in your argument.

Hint. Think about the case $k = 2$, and the theorem that says that a language is decidable if and only it it and its complement are semi-decidable.

290. SDSplitting

Suppose that L is semi-decidable but not decidable. (So $\overline{L}$ is not semi-decidable...) Consider the language

$$L' \stackrel{\text{def}}{=} \{0x \mid x \text{ is in } L\} \cup \{1x \mid x \text{ is not in } L\}$$

Can you say for certain (without knowing more about L) whether L' is decidable, semi-decidable, or non-semi-decidable? Justify your answer.

291. WhichSemiDec

Let X be decidable and let Y be semi-decidable but not decidable. Define

- $Z_1 = X - Y$
- $Z_2 = Y - X$

1. Exactly one of Z_1 or Z_2 is guaranteed to be semi-decidable. Which is it?
2. For the Z_i which is guaranteed to be semi-decidable, prove it. Here you may quote without proof any closure properties you know.
3. For the Z_i which is not guaranteed to be semi-decidable, give a decidable language X and an semi-decidable language Y which demonstrate this.

292. HaltSomewhere

Define K to be $\{p \mid \text{ for some input } x,\ p[x] \downarrow \lambda\}$. Show that K is semi-decidable.

293. DecClosureAgain

In this problem we will develop a different proof that the decidable sets are closed under intersection and union. It is not a *better* proof than the direct one in the text, but it will be good proof-practice for you.

In the proof below we will assume (only) that we have established the following results

- X is decidable if and only if $\overline{X}$ is decidable.
- If A and B are semi-decidable then $A \cup B$ is semi-decidable.
- If A and B are semi-decidable then $A \cap B$ is semi-decidable.

a) Now, to prove: If A and B are decidable then $A \cup B$ is decidable.

I've started the proof for you ...

Proof. It suffices to show that (i) $A \cup B$ is semi-decidable, and (ii) $\overline{A \cup B}$ is semi-decidable.

To prove (i) : *...fill in*

To prove (ii) : Since A and B are decidable, we have $\overline{A}$ and $\overline{B}$ are semi-decidable. So *...fill in the rest, ..., playing a trick with DeMorgan's Laws.* ///

b) Similar to the previous part, let's prove that if A and B are decidable then $A \cup B$ is decidable, using the same three assumptions there.

294. SemidecRed

Suppose A is semi-decidable and $A \preceq \overline{A}$. Prove that A is decidable.

295. NotSemiDec

We have already proved that the set AlwaysHalts of always-halting programs is not decidable. Prove that it is not even semi-decidable.

Chapter 37

Enumerability

Sometimes semi-decidable languages are called *recursively enumerable,* especially in older texts and papers. That term "recursively *enumerable*" seems odd given our definitions. There does not seem to be anything being "enumerated" at all. But an equivalent—and sometimes very convenient—characterization of "semi-decidable" clarifies things.

37.1 Definition. A program *e enumerates* a language A if, when executed with an empty input string, it generates a sequence of strings comprising precisely the elements of A.

For concreteness we may say that (i) the language A is defined over an alphabet not including the newline symbol, and (ii) the elements of A are written to standard output with newlines separating them.

It is not required that an enumerator generate the strings of A in any particular order. It is also not required that an enumerator generate strings only once. That is, an enumerator e for a language A may generate the same x from A many times. It is only required that each element of A occur *at least once* in the output of e (and, of course, that nothing not in A is output).

Note that an enumerator may halt after generating a finite number of strings; this means that it enumerates a finite language. But an enumerator may never halt, that's fine as well. So it is possible to enumerate an infinite language. Note that even if it doesn't halt it may enumerate only a finite language (think about an enumerator which prints the string "abracadabra" over and over again.)

37.1 Enumerability is Equivalent to Semidecidability

The main theorem of this section is that a language is semi-decidable if and only if it is enumerated by some program. Hence the traditional name "recursively enumerable" is not so dumb after all.

37.2 Theorem. *A language $A \subseteq \Sigma_2^*$ is semi-decidable if and only if there is a computer program which enumerates A.*

Proof. The "if" direction of the proof is easy: if A is enumerated by program e then it is easy to build a program p with such that $A = L(p)$. Here is pseudocode for p:

> *On input x;*
> *Start the enumeration from e;*
> *If x is ever output by e, return 1.*

The other direction is slightly tricky. Suppose A is semi-decidable, so that there is a program p with $L(p) = A$. We seek a program e to generate a list of the elements of A. The following idea almost works: under our standard enumeration of all strings $x_0, x_1, x_2, \ldots$, consider each x_i in turn as input to p. Whenever p accepts x_i, add x_i to the output of e.

The trouble with idea, of course, is that if we are not careful we might get caught in an infinite loop trying to decide whether p accepts some particular x_i and never get a chance to consider x_{i+1}, x_{i+2}, etc.

The trick is to run all the tests on all the x_i using the dovetailing trick of Section 36.2.

Here is the argument.

Suppose that p that semi-decides A. Here is a program e which enumerates A. Let $x_0, x_1, \ldots$ be an enumeration of all the string in Σ_2^*,

> dovetail $p[x_0], p[x_1], p[x_2], \ldots$;
> if and when and x_i is accepted by p, `print(`x_i`)` and continue;

///

Note that for the enumeration program e we built in the second part, each x in A actually is printed by e infinitely many times. What does the output of e look like in the case that A is a finite set?

37.2 Always-Terminating Programs Can't be Enumerated

At this point one can imagine someone making the following complaint.

> *All this stuff about programs halting is fine, but in real life we don't really care about programs which don't halt. I want a theory of computability which takes* ***always-halting*** *programs as the fundamental objects of study, and investigates these directly.*

This is a perfectly reasonable point of view. There is one problem, though. *It is impossible to carry out such a research plan.* The reason for that is that it is impossible to recognize these programs! That is, the set of programs which always halt is not only not a decidable language, it is not even semi-decidable.

37.3 Definition. A program p is terminating if it halts on all inputs.

Equivalently, program p is terminating if its associated partial function $\mathsf{pfn}(p)$ has domain all of Σ_2^*.

This is a potentially confusing terminology, since in other contexts, "terminating" can be used to refer to a particular computation, whereas here we apply the term to a program if *all* of its computations terminate.

We are going to show that not only is there no decision procedure to answer whether or not an arbitrary program is terminating, there cannot even be an algorithmic way of *listing* all of the terminating programs. The argument for this is illuminating: it gives a bit more insight into what is going on with the self-referential aspect of the proof that the halting problem is not decidable.

37.4 Theorem. *The following problem is not semi-decidable.*

Termination

INPUT: *A program p*

QUESTION: *Is p terminating; that is, does p halt on all inputs?*

Another way to state the Theorem is as follows.

$$\mathsf{Term} = \{p \mid p \text{ halts on all inputs } \} \quad \text{is not semi-decidable.}$$

Proof. Recall that a language A is semi-decidable iff there is an effective enumeration of A. So suppose for the sake of contradiction that there were a computer program *DecList* which generates a list

$$t_0, t_1, t_2, \ldots$$

such that every t_i is in Term and every program in Term occurs at least once in the list.

Recall that we have a specified an enumeration of all Σ_A strings as $x_0, x_1, x_2, \ldots$. Now construct the following program t^*.

> on input x;
> let i be the first number such that x_i is x;
> using *DecList*, generate t_i and simulate it on x_i;
> let y be the result of this computation; return $y1$ (the result of appending 1 to the end)

The main point is that the result of t^* on x_i differs from the result of t_i on x_i. Note that this implies that t^* computes a different function from each one of the t_i (for example t^* doesn't compute the same function as t_{173} because they differ at least on input x_{173}).

But the description above *does* defines a program, because we have assumed that we can generate the ith terminating program for any i, and once it is generated we can proceed to simulate it.

Furthermore, the program t^* we've defined is itself terminating. This is because each t_i is guaranteed to be terminating, so the tests in the code above always return.

But this is a contradiction: since t^* does not compute the same function as any of the t_i, our original listing was not complete after all.

///

37.5 Remark. Even though we haven't developed the tools to prove it, it is worth stating the following result. Recall that in Section 36.5 we saw that we could prove a language X to be non-semi-decidable by (i) proving that it was no decidable and (ii) proving that its complement $\overline{X}$ *was* semi-decidable. The language of always-halting program, that we were just talking about

$$Tot \stackrel{\text{def}}{=} \{p \mid p \text{ halts on all inputs}\}$$

is not semi-decidable, but its complement $\overline{Tot}$ is not semi-decidable either! In this sense, *Tot* is the most complex language we have seen so far.

Even without a proof you should be able to feel intuitively by now that $\overline{Tot}$ is not semi-decidable: to say that p is in $\overline{Tot}$ is to say that p runs forever on at least one input, and it is hard to imagine how one could verify this by some semi-decision procedure, isn't it?

37.3 Problems

296. NoReps

Suppose that e is an enumerator for a language A. Show that there exists an enumerator e for the same language A such that e prints each element of A exactly once, that is, with no repetitions. *Hint: you have unbounded memory at your disposal!*

As prelude to the next two questions, let us recall that, we can order the set Σ_2^* of strings, for example, by simple alphabetical ordering with the empty string as w_0, the strings of length one coming next, the strings of length two after those, etc. This naturally induces a linear ordering on strings, namely $x < y$ if x comes before y in the above ordering.

297. EnumDec1

We know that if A is decidable then it is semi-decidable, so that implies that if A is decidable then there is an enumerator e for A. We might expect, though, that if A is decidable then we can expect something extra nice about our enumerator. Indeed: prove the following.

Theorem. Let A be decidable. Then there is an enumerator e which generates the elements of A *in increasing order.* This is with respect to the standard ordering on strings: shorter strings first, then use alphabetical order. The string "strictly" here means that we do not allow repetitions.

298. EnumDec2

Show the converse of the previous question. That is, prove the following.

Theorem. Suppose there is an enumerator e which generates the elements of A *in strictly increasing order.* Then A is decidable.

Hint. First notice that if the language A of elements enumerated by e is finite, then A is certainly decidable. So it remains to consider the case when A is infinite — surprisingly, this makes things easier!

(As a matter of fact, the problem as stated is true even if we remove the word "strictly" from the assumption, which is to say that we allow repetitions in the enumeration. But focusing on strict enumeration makes the essential insight clearer.)

299. InfiniteUnion

Consider the following claim.

If $A_1, A_2, \ldots$ is a countably infinite set of semi-decidable languages, then their union $A_1 \cup A_2 \cup \ldots$ is a semi-decidable language.

1. Show that this claim is false.

   Hint. This is easy: remember that any singleton set is a semi-decidable language!

2. What is wrong with the following supposed "proof" of the claim?

   > Each A_1 is semi-decidable, so for each i there is a program p_i such that A_i is $L(p_i)$. Here is a program p such that $L(p)$ is $A_1 \cup A_2 \cup \ldots$
   >
   > > on input x;
   > > dovetail all the $p_i[x]$;
   > > if and when any of the $p_i[x]$ return 1,
   > > return 1.

   This looks like a proof that $A_1 \cup A_2 \cup \ldots$ is semideciable. But we had a counterexample in the previous part. What is going on? (The problem is *not* that we are dovetailing infinitely many programs. That is a straightforward generation of the finite-many-programs dovetailing technique.)

   Hint. This is a subtle problem. If you have an easy explanation, it's unlikely to be correct.

Chapter 38

Post's Correspondence Problem

38.1 The *PCP* Puzzle

Here is a puzzle. Consider the following "dominoes", which are just pairs of strings that we have chosen to write one on top of the other.

$$\begin{pmatrix} b \\ bbb \end{pmatrix} \qquad \begin{pmatrix} babbb \\ ba \end{pmatrix} \qquad \begin{pmatrix} ba \\ a \end{pmatrix}$$

The puzzle is: supposing you have as many copies of each domino as you like, can you lay them side-by-side so that the bit string made from the top row is the same as the bit string made from the bottom row?

The answer—in this case—is yes. If we concatenate copies of the second, the first, the first again, then the third, we get

$$\begin{pmatrix} babbb \\ ba \end{pmatrix}\begin{pmatrix} b \\ bbb \end{pmatrix}\begin{pmatrix} b \\ bbb \end{pmatrix}\begin{pmatrix} ba \\ a \end{pmatrix}$$

You can check that the first row and the second both make the string babbbbbba.

38.2 Examples

38.1 Example. Here is another such problem. We take our dominoes to be

$$\begin{pmatrix} b \\ bbb \end{pmatrix} \qquad \begin{pmatrix} ba \\ babbb \end{pmatrix} \qquad \begin{pmatrix} ba \\ aa \end{pmatrix}$$

Can we arrange these side-by-side, so that the bit string made from the top row is the same as the bit string made from the bottom row? Certainly not. Whenever we use any of the first two dominoes, the lower bit string is longer than the upper one. And we can't just use copies of the third domino, since the first bits in the upper and lower halves are different. So the top can never catch up to the bottom.

38.2 Example. Suppose our dominoes are

$$\binom{baa}{bb} \qquad \binom{ba}{babb} \qquad \binom{aba}{abb}$$

Is there a solution?

The answer here is again no. Can you prove that?

38.3 Example. One more example. Suppose our dominoes are

$$\binom{a}{b} \qquad \binom{b}{abb} \qquad \binom{abb}{a}$$

Is there a solution? Swing away from these notes for a little while and try to find one...

...You're back? If you didn't find one did you come away with the impression that there couldn't be one? Well, there is a solution, but you need to string together 44 dominoes, and that is the shortest solution.[1]

We have been playing with the *Post Correspondence Problem*, invented by the mathematician Emil Post in 1946. We can phrase the problem as a decision problem, namely, given a set of dominoes, decide, yes or no, whether or not there exists a way to arrange them side-by-side so that the bit string made from the top row is the same as the bit string made from the bottom row.

Let's make all this precise, by defining a decision problem.

> *Post's Correspondence Problem*
>
> INPUT: An alphabet Σ and a list $\binom{x_1}{y_1}, \ldots, \binom{x_n}{y_n}$ of ordered pairs of elements of Σ^*
>
> QUESTION: Does there exist a sequence $s = i_1, \ldots, i_m$ with $1 \leq i_j \leq n$ for all i,
>
> $$x_{i_1} \cdots x_{i_m} = y_{i_1} \cdots y_{i_m}$$

The Post's Correspondence Problem is often abbreviated *PCP*.

38.3 *PCP*, Turing Machines, and Programs

Recall that Turing machines are a mathematical model of computation, expanding on *DFAs*, that have, provably, the same computational power as C and other programming languages that you are familiar with.

[1] Here and elsewhere is this section I am relying on the work of Ling Zhao, who has the excellent website http://webdocs.cs.ualberta.ca/~games/PCP

The big reason why Turing machines are studied as a model of computing is that they are so simple: this allows them to be connected to other things that don't seem like computers, and so we can transfer computational ideas to other realms. The *PCP* is a great example.

38.4 Theorem. *There is an algorithm* PCPtoTM *that behaves as follows*

Given an arbitrary program p and string x it produces a PCP instance $\mathcal{P}$ such that p halts on x if and only if $\mathcal{P}$ has a solution .

The proof is complicated, we won't give it here. But the main ingredients are important to know about.

1. Using the fact that any program p can be transformed into a Turing machine t, it suffices to show that we can build a *PCP* instance corresponding to any Turing machine.
2. The heart of the proof is showing how to represent Turing machines configurations in terms of dominos, in such a way that passing from one machine configuration to the next in Turing machine land can be mirrored by successfully playing *PCP* dominos. And one sets things up so that when (and only when) the Turing machine halts, the *PCP* has completed successfully.

The reason all this works is that Turning Machines are so simple in their mechanics that one can simulate an arbitrary Turing Machine by building a clever *PCP* instance.

There are a zillion details to be worked through, but it's a classic result and so easy to find the details if you care to consult a text or do a web search.

38.4 Undecidability of *PCP*

With Theorem 38.4 in hand this next is easy.

38.5 Theorem. *Post's Correspondence Problem is undecidable.*

Proof. For sake of contradiction suppose we had a decision procedure q that determined whether *PCP* instances were solvable. Then the following would be a solution to the Halting Problem:

> on input p and w, use the algorithm *PCPtoTM* to build a *PCP* instance $\mathcal{P}$;
> run q on that; and return the same answer.

///

Let's make clear what this theorem says. It says that there can be no computer program that

- takes arbitrary instances of *PCP* as input
- always gives a yes/no answer
- answers yes if and only if there is a solution to the given instance.

Needless to say, the fact that *PCP* is undecidable as a decision problem does not mean that one cannot analyze individual instances, or even certain families of instances. But the theorem say that there cannot exists a program that always works.

38.5 Who Cares?

The *PCP* is a cool little puzzle, and it is amusing to know that no algorithm can solve it, but why should it matter to you as a Serious Computer Scientist?

The answer is that, somewhat amazingly, many decision problems L that really do arise in practice are reducible to the *PCP*, in the sense that we can exhibit a construction transforming instances of L into instances of the *PCP*. This means that *if* we could decide L *then* we could decide *PCP*. And so—since we *cannot* decide *PCP*, we cannot decide L.

Chapter 39 shows how this works, on a couple of examples in the context of context-free grammmars.

38.6 Problems

300. PCPWarmUp

For each instance of the *PCP* below, either give a solution or argue why no solution exists.

1. $$\begin{pmatrix} abc \\ ab \end{pmatrix} \begin{pmatrix} aca \\ ca \end{pmatrix} \begin{pmatrix} b \\ acab \end{pmatrix}$$

2. $$\begin{pmatrix} abc \\ ab \end{pmatrix} \begin{pmatrix} abba \\ c \end{pmatrix} \begin{pmatrix} c \\ ccc \end{pmatrix} \begin{pmatrix} bbba \\ cbbb \end{pmatrix} \begin{pmatrix} abcc \\ aab \end{pmatrix}$$

3. $$\begin{pmatrix} c \\ cb \end{pmatrix} \begin{pmatrix} b \\ aba \end{pmatrix} \begin{pmatrix} a \\ bab \end{pmatrix} \begin{pmatrix} aba \\ b \end{pmatrix} \begin{pmatrix} bab \\ a \end{pmatrix}$$

4. $$\begin{pmatrix} ab \\ a \end{pmatrix} \begin{pmatrix} aa \\ bab \end{pmatrix} \begin{pmatrix} b \\ aa \end{pmatrix} \begin{pmatrix} ba \\ ab \end{pmatrix}$$

301. PCPInf

Show that is a *PCP* instance P has a solution, then it has infinitely many solutions. (This is easy)

302. PCPEqLength

Suppose P is a *PCP* instance, and consider a domino having top and bottom strings of the same length, but not the same as strings.

True or false: this domino can never be played in a solution.

303. PCPNeqLength

Suppose P is a *PCP* instance with no dominoes having identical top and bottom strings.

True or false: a necessary condition for a *PCP* instance to have a solution is that there be one domino whose top string has length longer than that of its bottom string and another domino whose bottom string length is longer than that of its top string.

304. PCPUnary

Show that the *PCP* is decidable if the alphabet has only one symbol. (Give pseudocode for a decision procedure.)

305. PCPBinary

Explain how to algorithmically transform any *PCP* problem $\mathcal{P}$ over an arbitrary finite alphabet Σ into a problem $\mathcal{P}'$ over the alphabet $\{0,1\}$ such that $\mathcal{P}'$ has a solution if and only if $\mathcal{P}$ has a solution.

The moral of this is that, for the purposes of investigating which *PCP* problems have solutions, we may without loss of generality restrict our attention to the alphabet $\{0,1\}$.

306. PCPSubset

a) Prove or disprove: if P is a solvable *PCP* instance and P' consists of a nonempty subset of the dominoes of P then P' is solvable.

b) Prove or disprove: if P is an unsolvable *PCP* instance and P' consists of a nonempty subset of the dominoes of P then P' is unsolvable.

307. PCPDouble

Let P be a *PCP* instance, say

$$\binom{x_1}{y_1}, \ldots, \binom{x_n}{y_n}$$

Let P^2 denoted the *PCP* instance obtained by "doubling" every string appearing in P. That is P^2 has the dominos

$$\binom{x_1x_1}{y_1y_1}, \ldots, \binom{x_nx_n}{y_ny_n}$$

True or false: if P has a solution then P^2 has a solution.

Give a careful proof or a concrete counterexample.

308. PCPReverseAll

Let P be a *PCP* instance, say

$$\binom{x_1}{y_1}, \ldots, \binom{x_n}{y_n}$$

Let P^R denoted the *PCP* instance obtained by reversing every string appearing in P. That is, we obtain

$$\binom{x_1^R}{y_1^R}, \ldots, \binom{x_n^R}{y_n^R}$$

Prove that if P has a solution then P^R has a solution. (This isn't hard, but be careful and explicit in your solution)

309. PCPReverseOne

A variation on Problem 308. Prove or disprove: if P is a solvable *PCP* instance and we create P' by choosing *one* domino of P and reversing the top and bottom strings there, then P' is solvable.

Hint. As ever, a proof must be a general argument, a disproof should be a concrete example.

310. PCPBound

Fix the alphabet $\Sigma = \{0, 1\}$, and consider *PCP* instances over this alphabet.

Let us say that the "total size" of a *PCP* instance is the sum of the lengths of all the strings appearing in the dominoes. If a string appears in several places, count its length that many times.

Convince yourself that for each n there are only finitely many different *PCP* instances of total size n. (Easy.)

Now, if P is a *solvable PCP* instance, let $l(P)$ be the length of a shortest solution: the length of the sequence s described in the definition of the *PCP*. Let $bound : \mathbb{N} \to \mathbb{N}$ be the following function.

$bound(n) \stackrel{\text{def}}{=}$ the largest value of $l(P)$ among all the solvable *PCP* instances of length n

Note that f is well-defined because there are only finitely many instances of size n, and consequently only finitely many *solvable* instances.

Prove that *bound is not a total computable function.*

Hint. *Show that if bound were computable then the PCP would be solvable.*

311. PCPTwo

(Hard.) Show that the subclass of *PCP* instances that have only 2 dominoes is decidable. (Give pseudocode for a decision procedure.)

Feel free to assume that the alphabet is $\{0, 1\}$, per Problem 305.

This is a hard problem. It was first proved in [EKR82].

By the way, it is known that the subclass of *PCP* instances with 5 or more dominoes is undecidable. Decidability is unknown for $k = 3$ or 4.

Chapter 39

Undecidability Results about *CFGs*

39.1 PCP meets CFG

If P is an instance of the PCP there are two context-free grammars easily derived from P, which we will call G_T and G_B. The T and B are for "top" and "bottom".

Each of G_T and G_B are *CFGs* with terminal alphabet $\Sigma \cup \{d_1, d_2, \ldots, d_n\}$, where Σ is the alphabet the PCP is defined over and the d_i are fresh alphabet symbols (one corresponding to each domino).

Each of G_T and G_B will have a single variable (which is of course their start variable): for G_T this variable is S_T and for G_B this variable is S_B.

Suppose P has dominos $\binom{x_1}{y_1}, \ldots, \binom{x_n}{y_n}$ over alphabet Σ. Here are the rules for G_T.

$$\begin{aligned} S_T &\rightarrow x_1\, S_T\, d_1 \mid x_1 d_1 \\ &\vdots \\ S_T &\rightarrow x_n\, S_T\, d_n \mid x_n d_n \end{aligned}$$

The rules for G_B are similar

$$\begin{aligned} S_B &\rightarrow y_1\, S_B\, d_1 \mid y_1 d_1 \\ &\vdots \\ S_B &\rightarrow y_n\, S_B\, d_n \mid y_n d_n \end{aligned}$$

The idea is that the grammar G_T generates precisely the strings across the top that could be generated by playing dominos from P, together with a record (the d_i) of which dominos were played . Of course this record has to be read backwards, but that won't matter.

And of course the grammar G_B generates precisely the strings across the bottom that could be generated by playing dominos.

Please note that the grammars G_T and G_B depend on which PCP instance we started with, so we should really show that in the notation, and call them something like G_T^P and G_B^P, but that's ugly, so we won't do it. Just keep in mind that whenever we speak of a G_T or G_B we always have a specific PCP instance in the background.

39.1 Check Your Reading. *Do Problem PCPGrammars now.*

The point to defining G_T and G_B is that they make a bridge between the world of PCP and the world of grammars. We'll cross that bridge back-and-forth in the next few sections.

39.1.1 Useful Facts about G_T and G_B

The grammars G_T and G_B are pretty simple, and they satisfy two properties that will be useful in proving the undecidability of *CFG* Ambiguity and *CFG* Universality respectively. Let's state them here, even before we need them, so that the flow of Section 39.3 and Section 39.4 will be simpler.

In Section 39.3 we will use the following fact.

39.2 Lemma. *For every PCP instance P, the grammars G_T and G_B are unambiguous.*

Proof. This is easy to see. The strings derivable in either grammar are all of the form

$$z\, d_{i_m} \cdots d_{i_1}$$

where z is some string over the PCP alphabet and the d_i are symbols unique to each rule. The only way to generate d_j in the output is to apply the jth rule, and the order of the ds says exactly what the order of the rules applied were. So two different derivations cannot generate the same string. This observation applies to both G_T and G_B. ///

In Section 39.4 we will use the following fact. We know that it is not the case that every context-free language has a context-free complement. But the grammars G_T and G_B happen to always generate languages whose complement *is* context-free.

39.3 Lemma. *For every PCP instance P, we can construct grammars G'_T and G'_B that generate the complements of the languages $L(G_T)$ and $L(G_B)$.*

Proof. This is somewhat tricky, and we don't give the proof here. A proof (using pushdown automata) can be found in [HMU06].

///

39.2 Context-Free Language Intersection

Here we show that it is undecidable whether two given context-free grammars generate any strings in common.

39.4 Lemma. *Let P be a PCP instance. Then P has a solution if and only if $L(G_T) \cap L(G_B) \neq \emptyset$.*

Proof. If P has a solution, this means that there is a sequence $s = i_1, \ldots, i_m$ with

$$x_{i_1} \cdots x_{i_m} = y_{i_1} \cdots y_{i_m}$$

But if $x_{i_1} \cdots x_{i_m}$ is the top string generated by the play $i_1, \ldots, i_m$, then G_T generates the string

$$x_{i_1} \cdots x_{i_m} \, d_{i_m} \cdots d_{i_1}$$

Similarly, G_B generates the string

$$y_{i_1} \cdots y_{i_m} \, d_{i_m} \cdots d_{i_1}$$

And if $x_{i_1} \cdots x_{i_m} = y_{i_1} \cdots y_{i_m}$ (that's what it means to have a solution to the PCP!) those two generated strings are equal.

Conversely, the only way there could be a string generated by both grammars is for it to represent a winning play in the PCP game. This is essentially because every string generated by either grammar loks like $z \, d_{i_m} \cdots d_{i_1}$ where the sequence of ds records a play of dominos. ///

Now we get our undecidability result immediately.

39.5 Theorem. *The following problem is undecidable.*

Context-Free Grammar Intersection

INPUT: *Context-Free Grammars G_1 and G_2*

QUESTION: *Is $L(G_1) \cap L(G_2) \neq \emptyset$?*

Proof. For sake of contradiction, suppose there were a decision procedure $\mathcal{D}_\cap$ for this problem. Then the following would be a decision procedure for the Post Correspondence Problem.

- Given instance P of the PCP;

- Build G_T and G_B from P;
- Call $\mathcal{D}_\cap$ on these two grammars;
- If $\mathcal{D}_\cap$ returns YES, return YES, else return NO

The fact that this would be a correct decision procedure for the PCP is a consequence of Lemma 39.4. Since there can be no decision procedure for the PCP, this contradiction shows that there can be no decision procedure for the Context-Free Grammar Intersection problem. ///

39.3 Context-Free Grammar Ambiguity

Here we show that it is undecidable whether a given context-free grammar is ambiguous.

Let P be a PCP instance. We create another grammar G_A from P. This time we start with G_T and G_B derived from P as before, but now combine them into one grammar that generates the union of their languages. Namely, add one new variable S, and add

$$S \to S_T \mid S_B$$

to the productions for G_T and G_B.

What strings are generated by G_A? Precisely those strings

$$a_1 a_2 \ldots a_k d_{i_m} \cdots d_{i_1} \qquad \text{// here the } a_j \text{ are } \Sigma\text{-symbols}$$

such that, if the dominos are played according to the sequence $d_{i_1} \cdots d_{i_m}$, then $a_1 a_2 \ldots a_k$ is *either* the string along the top *or* the string along the bottom. This is because the first step in the derivation of $a_1 a_2 \ldots a_k d_{i_m} \cdots d_{i_1}$ has to be either $S \Longrightarrow S_T$ or $S \Longrightarrow S_B$, and once that move is made we can only apply rules that originated from G_T or G_B respectively.

Now we are ready to prove what we want.

39.6 Lemma. *Let P be a PCP instance. Then P has a solution if and only if the grammar G_A is ambiguous.*

Proof. If P has a solution, this means that there is a sequence $s = i_1, \ldots, i_m$ with

$$x_{i_1} \cdots x_{i_m} \;=\; y_{i_1} \cdots y_{i_m}$$

Just as in Lemma 39.4 this means that G_T generates the string

$$x_{i_1} \cdots x_{i_m}\, d_{i_m} \cdots d_{i_1}$$

and G_B generates the string

$$y_{i_1} \cdots y_{i_m}\, d_{i_m} \cdots d_{i_1}$$

As we noted before, these are the same string, since $x_{i_1} \cdots x_{i_m} = y_{i_1} \cdots y_{i_m}$. This string has two parse trees in G_A, one that has the step $S \Longrightarrow S_T$ and the start and one that has $S \Longrightarrow S_B$ at the start.

We need to show the converse, that if G_A is ambiguous then P has a solution. This is where we need the observation that each of G_T and G_B are unambiguous on their own (Lemma 39.2). Given that, the only way for a string w to have two parse trees in G_A is for the first steps to be different, that is, to have $S \Longrightarrow S_T \Longrightarrow^* w$ and also $S \Longrightarrow S_B \Longrightarrow^* w$. But as we argued in the proof of Lemma 39.4 this means that P has a solution. ///

The proof of the next theorem follows the same pattern as the proof of Theorem 39.5. We have used the same wording as much as possible, to emphasize this.

39.7 Theorem. *The following problem is undecidable.*

Context-Free Grammar Ambiguity

INPUT: *A Context-Free Grammar G*

QUESTION: *Is G ambiguous?*

Proof. For sake of contradiction, suppose there were a decision procedure $\mathcal{D}_{ambig}$ for this problem. Then the following would be a decision procedure for the Post Correspondence Problem.

- Given instance P of the PCP;
- Build G_A from P;
- Call $\mathcal{D}_{ambig}$ on G_A;
- If $\mathcal{D}_{ambig}$ returns YES, return YES, else return NO

The fact that this would be a correct decision procedure for the PCP is a consequence of Lemma 39.6. Since there can be no decision procedure for the PCP, this contradiction shows that there can be no decision procedure for the Context-Free Grammar Ambiguity problem. ///

39.4 Context-Free Language Universality

When we considered the Universality problem for *DFAs* (does the given *DFA* accept all strings?) the decision procedure was an easy tweak of the decision procedure for *DFA* Emptiness. That was because swapping accepting and non-accepting *DFA* states was an easy way to complement a accepted language.

When we pass to *CFGs*, we have a decision for the Emptiness problem. But we do not have an easy way to "complement a grammar," indeed we know that if K is a context-free language then the complement of K may or may not be context-free.

And in fact: it is undecidable whether a *CFG* generates all strings.

Let P be a PCP instance. We create yet one more another grammar G_U from P. Here is where we use the fact about G_T and G_B (Lemma 39.3) that we can construct grammars G'_T and G'_B with $L(G'_T) = \overline{L(G_T)}$ and $L(G'_B) = \overline{L(G_B)}$. Then we let G_U be the grammar built in the standard way to capture the union of those, that is, $L(G_U) = L(G'_T) \cup L(G'_B) = \overline{L(G_T)} \cup \overline{L(G'_B)}$.

39.8 Lemma. *Let P be a PCP instance. The P has a solution if and only if the grammar G_U does **not** generate all strings over its terminal alphabet.*

Proof. We showed in Lemma 39.4 that P has a solution if and only if $L(G_T) \cap L(G_B) \neq \emptyset$. Taking complements of each side and doing some basic set theory we have:

$$\begin{aligned} P \text{ has a solution} \quad &\text{if and only if } \overline{L(G_T) \cap L(G_B)} \neq \Sigma^* \\ &\text{if and only if } \overline{L(G_T)} \cup \overline{L(G_B)} \neq \Sigma^* \\ &\text{if and only if } L(G'_T) \cup L(G'_B) \neq \Sigma^* \end{aligned}$$

which is what we wanted to show.

///

Again the proof of the next theorem follows the same pattern as the proof of Theorem 39.5.

39.9 Theorem. *The following problem is undecidable.*

Context-Free Grammar Universality

INPUT: *A Context-Free Grammar G with terminal alphabet Σ*

QUESTION: *Is $L(G) = \Sigma^*$?*

Proof. For sake of contradiction, suppose there were a decision procedure $\mathcal{D}_{univ}$ for this problem. Then the following would be a decision procedure for the Post Correspondence Problem.

- Given instance P of the PCP;
- Build G_U from P;
- Call $\mathcal{D}_{univ}$ on G_U;
- If $\mathcal{D}_{univ}$ returns YES, return NO, else return YES

The fact that this would be a correct decision procedure for the PCP is a consequence of Lemma 39.8. Since there can be no decision procedure for the PCP, this contradiction shows that there can be no decision procedure for the Context-Free Grammar Universality problem. ///

39.5 Problems

312. PCPGrammars

Choose some *PCP* problems, and write down the corresponding grammars G_T and G_B. Play some dominos in the *PCP* game. Then write down the derivation in G_T and the derivation in G_B that correspond to that play.

313. CFGEquivalence

Show that the following problems is undecidable.

> *CFG Equivalence*
> INPUT: Two *CFGs* G_1 and G_2
> QUESTION: Is $L(G_1) = L(G_2)$

Hint. This isn't hard. Use Theorem 39.9.

314. CFGContainment

Show that the following problems is undecidable.

> *CFG Equivalence*
> INPUT: Two *CFGs* G_1 and G_2
> QUESTION: Is $L(G_1) \subseteq L(G_2)$

Hint. Use Theorem 39.9.

315. CFGRegContainment

Consider the following two decision problems.

> *CFG Contains DFA*
> INPUT: A context-free grammar G and a DFA M
> QUESTION: Is $L(M) \subseteq L(G)$?
>
> *DFA Contains CFG*
> INPUT: A context-free grammar G and a DFA M
> QUESTION: Is $L(G) \subseteq L(M)$?

Exactly one of these problems is undecidable. Which one is it? Prove it.

316. CFGDecUndec

For each of the following problems, figure out whether it is decidable. Then either prove it undecidable or give an algorithm solving it.

Caution. As you do these problems, don't make the blunder of assuming that you do "complement constructions" or "intersection constructions" on grammars. It is just not true that for each *CFG* G there is a G' generating the complement of $L(G)$' a similar remark holds for intersections.

a)

CFG Complement Universal

INPUT: A *CFG* $G = (\Sigma, V, P, S)$

QUESTION: Is $\overline{L(G)} = \Sigma^*$?

b)

CFG Complement Empty

INPUT: A *CFG* $G = (\Sigma, V, P, S)$

QUESTION: Is $\overline{L(G)} = \emptyset$?

c)

CFG Finiteness

INPUT: A *CFG* $G = (\Sigma, V, P, S)$

QUESTION: Is $L(G)$ finite ?

d)

CFG Covering

INPUT: Two *CFGs* G_1 and G_2

QUESTION: Is $L(G_1) \cup L(G_2) = \Sigma^*$

So we are trying to check whether G_1 and G_2 collectively generate all strings.

317. CFGNoInt

We know that there is no algorithm taking two *CFGs* G_1 and G_2 and returning a *CFG* G such that $L(G) = L(G_1) \cap L(G_2)$. But suppose you didn't know that fact.

Explain how you could *conclude* that fact from Theorem 39.5.

Chapter 40

Proving Languages Undecidable

We have a lot of techniques to show languages to be decidable, closure under various operations and so forth. And we have some examples of undecidability, proved on a case-by-case basis. In this chapter we study catalog and explore *general* techniques for proving undecidability.

40.1 Direct Approaches

First we showed the Halting Problem to be undecidable, directly.

Then in Chapter 33 we showed, directly, that determining whether a given program *always* halted was undecidable.

Here is the language corresponding to this decision problem:

$$\mathsf{AlwaysHalts} \stackrel{\text{def}}{=} \{p \mid \ p \text{ is a program that always halts.}\}$$

So another way to state the result of Chapter 33 is: the language AlwaysHalts is undecidable.

40.1 Theorem. *The language* AlwaysHalts *is undecidable.*

40.2 Exploiting Closure Properties

We can sometimes use closure properties to deduce that a language is undecidable. You explored this in Problem 280.

40.3 Reducibility: The Idea

The most typical proof of undecidability is structured as follows. To show some language X to be undecidable, we (i) first cleverly choose some language U that we know to be undecidable, then (ii) give an argument if *if* X were decidable then U would be decidable. This contradiction establishes that X cannot be decidable after all.

This was precisely what we did in proving Rice's Theorem in Chapter 34: we showed that *if* a give functional property were decidable *then* SelfHalt would be decidable.

This kind of reasoning can be formalized, and in doing so we get good insight into the *structure* of all possible languages. In this section we give such a formalization.

The technique is called *reducibility*.

40.2 Definition. A language A is *reducible* to language B, written $A \preceq B$, if there is a computable total function $f : \Sigma_2^* \to \Sigma_2^*$ such that for every w,

$$w \in A \text{ if and only if } f(w) \in B$$

40.3 Remark. Reducibility is a somewhat overloaded word in computer science. There are several kinds of "reducibility" that are studied. For example, the contemporary notion of polynomial-time reducibility in complexity theory, which underlies the theory of NP-completeness for example, is a variant of our reducibility in which the reducing function is required to be efficiently computable.

Just as reducibility is the standard technique for showing problems to be decidable or undecidable, this refined notion, of polynomial-time reducibility, is the most common technique for establishing complexity results. You can read more about this in any textbook about complexity.

For this text we always just use "reducible" in the sense of Definition 40.2.

40.4 Example. For any language A we have $A \preceq A^R$. The function is simply this:

$$f(x) \stackrel{\text{def}}{=} x^R$$

It is clear that f is total and computable, and that $x \in A$ if and only if $f(x) \in A^R$.

40.5 Example. Let $A = \{w \mid |w| \text{ is even}\}$, let $B = \{w \mid |w| \text{ is odd}\}$. Then $A \preceq B$. Here is a f that works: $f(x) = x1$. [that is, append a "1"]

40.6 Example. Let $A = \{a^n b^n c^n \mid n \geq 0\}$, let $B = \{a^{2n} b^n \mid n \geq 0\}$. Then $A \preceq B$. Here is a reduction function f that works:

$f(x) =$ the result of (i) removing all the cs in x, and (ii) replacing each a by a aa

40.7 Check Your Reading. *Show that in Example 40.6 we also have $B \preceq A$. (This is not typical!)*

The next easy observation is the crucial fact about reducibility. It is our universal tool for proving things undecidable.

40.8 Theorem. *Suppose A is reducible to B. If B is decidable then A is decidable.*

In symbols: if $A \preceq B$ and B is decidable then A is decidable.

Proof. Suppose that $A \preceq B$ via function f; let p_f be a program that computes f. Now let p_B be a program that halts on all inputs and decides B. The the following is a program that halts on all inputs and decides A:

> *on input w:*
> *compute $p_f[w]$; run p_B on the result;*
> *return that answer;*

This halts on all w since p_f and p_B both halt on all inputs. And it decides A by definition of the fact that f reduces A to B. ///

A couple of notes:

- it is crucial in the above proof that the function f be total, *i.e.* that p_f always halts, otherwise we would not be able to make our decision procedure for A apply to all inputs.
- it is crucial in the above proof that the function f be computable, since we had to build a *program* for deciding membership in A by passing to B.

The following is immediate from the theorem.

40.9 Corollary. *Suppose A is reducible to B. If A is undecidable then B is undecidable.*

In symbols: if $A \preceq B$ and A is undecidable then B is undecidable.

Corollary 40.9 is an invaluable tool for showing languages to be undecidable.

40.10 Example. Here is a silly example; silly because it is not a decision problem that anyone would care about, but worth seeing as a hint of how to use reducibility to show things undecidable. More significant examples will come soon.

Consider the language $\mathsf{SelfHalt}^R$, the set of reversals of strings in SelfHalt. (Remember that SelfHalt, defined as $\{p \mid p[p]\downarrow\}$, is an undecidable language.) The language $\mathsf{SelfHalt}^R$ is clearly undecidable; here is what a proof based on reducibility looks like.

Proof. To show that the set $\mathsf{SelfHalt}^R$ is undecidable, Corollary 40.9 says that all we have to do is to prove $\mathsf{SelfHalt} \preceq \mathsf{SelfHalt}^R$. For that it suffices to construct a computable total function f with $x \in \mathsf{SelfHalt}$ if and only if $f(x) \in \mathsf{SelfHalt}^R$. We define f by: $f(x) = x^R$. ///

More generally

40.11 Example. For any A, A is undecidable if and only if A^R is undecidable.

Since: If A is undecidable then $A \preceq A^R$ shows that A^R is undecidable. If A^R is undecidable then $A^R \preceq (A^R)^R = A$ shows that A is undecidable.

40.12 Example. Let A be decidable and let B be any language which is not $\emptyset$ and which is not Σ_2^*. Then $A \preceq B$.

Note that we do not assume anything about the decidability of B.

Proof. Let p be an algorithm deciding membership in A; let z_1 be some string in B and let z_0 be some string not in B. The following is an algorithm for a computable function reducing A to B.

> on input x;
> if $p[x]$ returns 1 return z_1 else return z_0

Clearly this function is total and computable, since p is always-terminating. ///

This is a dangerous example! It might encourage a misconception about how reductions can be defined. We've already stressed this point (in an earlier "Caution!") but let's be clear: don't let that test for membership in A suggest that you can do that whenever you are trying to define a reduction. In most interesting situations, the reduction function f has to be defined without any knowledge of A, because f has to be computable. This is a very special case, when A is decidable.

Two Potential Gotchas

The direction of the reduction matters. Suppose you have $A \preceq B$. If you know that A is decidable, then *you can conclude nothing* about whether B is decidable or not. Similarly if you know B to be undecidable, you know nothing about the decidability of A.

The reduction function must be total and computable If you are asked to prove A is reducible to B you must avoid the temptation to let your reduction function f make reference to membership in A, except for the uninteresting special case when A is decidable. In all interesting situations, the reduction function f has to be defined without any knowledge of A, because f has to be computable.

To put this more concretely: if you are trying to prove $A \preceq B$ then you *cannot* say something like, "the definition f on input x is: if $x \in A$ then ...something ... but if $x \notin A$ then ... something else ...". This pseudocode will be bogus *unless* you know A is decidable!

Transitivity of $\preceq$

This result is easy but it is used constantly.

40.13 Lemma. *Suppose $A \preceq B$ and $B \preceq C$. Then $A \preceq C$.*

Proof. Let f witness $A \preceq B$ and let g witness $B \preceq C$. Claim: the function $(g \circ f)$ witnesses $A \preceq C$.

Since: $(g \circ f)$ is certainly computable since it is the composition of computable functions. Now for any w, if $w \in A$ then $f(w) \in B$ and so $g(f(w)) \in C$; furthermore if $w \notin A$ then $f(w) \notin B$ and so $g(f(w)) \notin C$. This show that $w \in A$ iff $(g \circ f)(w) \in C$. ///

40.4 Reduction by Program Transformation

In the last section we did dumb reducibility examples like $A \preceq A^R$, just to get familiar with the concept. To do real applications of the idea we need to be more subtle.

Suppose X and Y are two languages that are sets *of programs,* and we would like to show that $X \preceq Y$ (presumably because we already know X is undecidable and we want to prove that Y is undecidable). What we can do is to define a reduction function f that takes as input a string p and returns a string $f(p)$ with the property that **if** the input string happened to be the source code of some program in X **then** $f(p)$ will be the source code of some program in Y, **otherwise** $f(p)$ will not be the source code of some program in Y.

Such an f is a *program transformer:* given a string w, thought of as a program, it constructs the text $f(w)$ of a new program. When the f we use is a computable function then this counts as a reduction $X \preceq Y$.

This needs lots of examples. The technique is tricky to absorb, but in using it, it really is the same idea over and over.

The Hello World Problem

Hello-World

INPUT: A program p

QUESTION: Does p return "Hello World!" on every input?

Here is the language corresponding to this decision problem:

$$\mathsf{HelloWorld} \stackrel{\text{def}}{=} \{p \mid \ p \text{ is a program that always returns "Hello World!".}\}$$

You might think it is easy to decide whether a given program prints "Hello World!" on every input, just be looking at the code. But suppose the program consisted of some crazy complex while-loop just in front of a `return ``Hello World!''` statement? Then the question of whether this program always returns "Hello World!" really comes down to the question of whether that while loop always terminates, right? And that could be hard to decide.

Indeed we have the following theorem. We explain the proof in great detail since it is very subtle.

40.14 Theorem. HelloWorld *is undecidable.*

Proof. We will show that SelfHalt $\preceq$ HelloWorld. Consider the following transformation of programs: for a given program p we construct a new program p' that behaves as follows:

on input x;
simulate p on p;
if this simulation halts, return "Hello World!"

Yes, that *"simulate p on p"* line has nothing to do with the actual input to p'. It's just a time-waster. But—and this is the point—if p is a program that doesn't halt on itself, this is an **infinite** time-waster! We have:

p	p'
halts on itself	always returns "Hello World!"
fails to halt on itself	halts on no inputs

Then $p \in$ SelfHalt if and only if p' always returns "Hello World!"

We're basically done: having seen how to build a program p' based any given program p, we define our program transformer f to simply be the function that builds p' from p. But since f is supposed to be a total function we need to deal with the fact that not every string that f has to act on is a legal program, but this is only a mild annoyance. If the input w to f doesn't parse as a program, then f can leave it alone, since this w will not be in SelfHalt and it won't be in HelloWorld either.

So here is pseudocode for f

$$f(w) = \begin{cases} p' \text{ as described above} & \text{if } p \text{ parses as a program} \\ w & \text{otherwise} \end{cases}$$

Thus SelfHalt $\preceq$ HelloWorld via this f. ///

This was a hard proof to understand. But it is worth the effort, since this basic trick actually proves lots of seemingly different results with no extra effort. Read on.

The Identity Problem

Consider the decision problem of determining whether a given program computes the identity function.

Program Identity
INPUT: A program p
QUESTION: Is $\mathsf{pfn}(p)$ the identity function?

Here is the language corresponding to this decision problem:

$$\mathsf{Id} \stackrel{\text{def}}{=} \{p \mid \ p \text{ is a program computing the identity function}\}$$

The proof of the next theorem will be almost word-for-word the same as the last proof.[1] Read carefully, and notice the tiny places where it differs.

40.15 Theorem. *The language* Id *is undecidable.*

Proof. We will show that SelfHalt $\preceq$ Id. Consider the following transformation of programs: for a given program p we construct a new program p' that behaves as follows:

[1] except that we will chat less

on input x;
simulate p on p;
if this simulation halts, return x

We have:

p	p'
halts on itself	returns its input
fails to halt on itself	halts on no inputs

Then $p \in \mathsf{SelfHalt}$ iff p' computes the identity function.

Here is pseudocode for f. Again we need to handle the case where the input string for f isn't even a program. But if the input w to f doesn't parse as a program, then f can leave it alone, since this w will not be in SelfHalt and it won't be in Id either.

$$f(w) = \begin{cases} p' \text{ as described above} & \text{if } p \text{ parses as a program} \\ w & \text{otherwise} \end{cases}$$

Thus $\mathsf{SelfHalt} \preceq \mathsf{Id}$ via this f. ///

The Everywhere-Halting Problem

Consider the decision problem of determining whether a given program halts on *all* of its inputs. Note that this is the same asking whether $\mathsf{pfn}(p)$ is a total function.

We showed this to be undecidable in Chapter 33. We prove it again here as an application of the reducibility technique.

Always-Halting
INPUT: A program p
QUESTION: Does p halt on all inputs?

Here is the language corresponding to this decision problem:

$$\mathsf{AlwaysHalts} \stackrel{\text{def}}{=} \{p \mid \ p \text{ is a program that always halts.}\}$$

40.16 Theorem. *The language* AlwaysHalts *is undecidable.*

Proof. We will show that SelfHalt $\preceq$ AlwaysHalts.

And now can use **exactly the same** f as we used in the proof of Theorem 40.14.

$$f(w) = \begin{cases} p' \text{ as described in 40.14} & \text{if } p \text{ parses as a program} \\ w & \text{otherwise} \end{cases}$$

Why does this work? Because the function that always returns "Hello World!" happens to be a total function , and the function that halts on no inputs isn't. So when this f reduced SelfHalt to HelloWorld, it just so happens that it also reduces SelfHalt to AlwaysHalts.

For that matter, in this example we could also have used the same function that we used in Theorem 40.15. Check that for yourself. ///

Program Non-Emptiness

Consider the decision problem of determining whether a given program halts on *any* of its inputs. Note that to say that a program p halts on some inputs is the same as saying the $\mathsf{pfn}(p)$ is not the empty function.

Program Non-Emptiness

INPUT: A program p

QUESTION: Is $\mathsf{pfn}(p) \neq \emptyset$?

Here is the language corresponding to this decision problem:

$$\mathsf{NonEmp} \stackrel{\text{def}}{=} \{p \mid \mathsf{pfn}(p) \neq \emptyset\}$$

40.17 Theorem. *The language* NonEmp *is undecidable.*

Proof. We leave this proof as an exercise for you (Problem 333). ///

By the way, since the decidable languages are closed under complement, we have the following corollary.

40.18 Corollary. *The following language is undecidable.*

$$\mathsf{Emp} \stackrel{\text{def}}{=} \{p \mid \mathsf{pfn}(p) = \emptyset\}$$

Proof. If Emp were decidable, then its complement NonEmp would be decidable, contradicting the theorem just proved. ///

One you work through the proofs in this section—or even a few of them—you can't help but think, "since all these results can be proved by essentially the same trick, there must be one theorem that captures them all at once." You'd be right. That theorem is Rice's Theorem, in Chapter 34.

40.5 The Generalization Trick

As our store of known undecidable languages increases it becomes easier to show new things undecidable. This is what reducibility is all about; but there is an another, easier, way of leveraging known undecidability results that works sometimes.

This is: to recognize that the problem you want to prove undecidable is a *more general version* of a problem you already know to be undecidable.

We have already used this trick, when we concluded that the undecidability of the Self-Halting problem entailed the undecidability of the full Halting problem. Here is another example.

The Program Equivalence Problem

Consider the following fundamental problem:

> *Program Equivalence*
> INPUT: Two programs p_1 and p_2.
> QUESTION: Does $\mathsf{pfn}(p_1) = \mathsf{pfn}(p_2)$?

This is undecidable. And it is *easy* to show that, using generalization, given the work we did earlier in the section. To be consistent with the other results in the chapter we will phrase the result in terms of a language.

Theorem. *The following language is undecidable.*

$$\mathsf{Equiv} \stackrel{def}{=} \{\langle p_1, p_2\rangle \mid \mathsf{pfn}(p_1) = \mathsf{pfn}(p_2)\}$$

Proof Idea The intuition behind the proof is that if we could decide whether *any* two programs are equivalent in the sense of computing the same function, then we should be able to decide whether an arbitrary given program is equivalent to specific fixed program.

Proof. We proved earlier that the language

$$\mathsf{Id} \stackrel{\text{def}}{=} \{p \mid \ p \text{ is a program that computes the identity function}\}$$

is undecidable. Let p be some particular program that computes the identity function.

Here is a function that witnesses $\mathsf{Id} \preceq \mathsf{Equiv}$:

$$f(w) \stackrel{\text{def}}{=} \langle w, p \rangle$$

To see that this works: certainly f is computable and total. If w is a program that computes the identity function then w and p compute the same function, that is, $f(w) = \langle w, p \rangle$ is in Equiv.

And if w is a not program that computes the identity function then $f(w) = \langle w, p \rangle$ is not in Equiv.

This completes the proof. ///

40.6 Problems

318. EvenUndec

If A is a language, let A_{even} be the set of strings in A of even length.

a) Show that if A is decidable then A_{even} is decidable.

b) True or false? If A is undecidable then A_{even} is undecidable.

319. ClosureUndec

1. True or false? For all A and B, if A and B are undecidable then $A \cup B$ is undecidable. True or false? For all A and B, if A and B are undecidable then $A \cap B$ is undecidable.

320. 0pUndec

Prove that the following language is not decidable.

$$A \stackrel{\text{def}}{=} \{0p \mid p \in \mathsf{SelfHalt}\}$$

This is the set of all programs in SelfHalt but with the single symbol "0" prepended to each one.

Hint. Show that $\mathsf{SelfHalt} \preceq A$.

321. CarefulMred

These are about reading the definition of $\preceq$ carefully.

1. Suppose $A \preceq \emptyset$. What can you say about A? Prove your answer.
2. Suppose $A \preceq \Sigma_2^*$. What can you say about A? Prove your answer.
3. Suppose $\emptyset \preceq A$. What can you say about A? Prove your answer.
4. Suppose $\Sigma_2^* \preceq A$. What can you say about A? Prove your answer.

322. ConstantUndec

Prove that the following language is undecidable.

$$\mathsf{Id} \stackrel{\text{def}}{=} \{p \mid \ p \text{ is a program computing the constant function returning } 101\}$$

323. MredIntersection

Prove or disprove: For all A and B, $A \preceq (A \cap B)$

324. MredPreorder

The relation $\preceq$ is almost a partial order. We proved that it is transitive.

1. Prove that $\preceq$ is reflexive: we always have $A \preceq A$
2. Show by example that $\preceq$ is not anti-symmetric: $A \preceq B$ and $B \preceq A$ does not imply $A = B$.

This last is the sense in which $\preceq$ is not a partial order; relations that are reflexive and transitive but not necessarily anti-symmetric are called "preorders".

325. MredComplement

Prove that If $A \preceq B$ then $\overline{A} \preceq \overline{B}$. (super easy.)

326. DecMred

Let *Odd* be the set of all bit strings of odd length. Show that $Odd \preceq \mathsf{SelfHalt}$

Hint. Remember that we emphasized that (in general) in proving $X \preceq Y$ for some languages X and Y by building a function f, we couldn't necessarily let f test whether its input is in X, since f has to be computable.

The trick in this problem is to realize that since we are working with *Odd*, clearly a decidable language, our function f to reduce *Odd* to SelfHalt can, if we wish, involve a test for membership in *Odd*.

327. MredOdd

Let A be any language; then let A_{odd} be the set all odd length strings in A. Prove that if A_{odd} is not decidable then A is not decidable.

Hint. You need a little observation to start: explain why there must be at least one string w not in A. Then this w will be handy in the rest of the proof.

For bragging rights: give two proofs, one using reducibility, one using closure properties.

328. Helper

This problem is here mainly to be useful in problem 329

Construct a language X such that

- X is undecidable, and
- every string in X has odd length.

Hint. You can start your proof by saying, "Let Y be any undecidable language. Now do a little construction based on Y that gives you a set X of odd-length strings with $Y \preceq X$.

329. OddMredConverse

Prove or disprove: If A is not decidable then A_{odd} is not decidable. (Compare Problem 327)

330. 2SelfHalt

Prove that the following language is not decidable.

$$2\,\mathsf{SelfHalt} = \{pp \mid p \in \mathsf{SelfHalt}\}$$

Note that $2\,\mathsf{SelfHalt}$ is not the same as the concatenation $\mathsf{SelfHalt}\,\mathsf{SelfHalt}$

331. 2AMred

For any A, define

$$2A = \{xx \mid x \in A\}$$

Note that $2A$ is not the same as the concatenation AA

1. Prove that for any A we have $A \preceq 2A$.
2. Explain why it is *not* true that $2\Sigma_2^* \preceq \Sigma_2^*$
3. Prove that if $A \neq \Sigma_2^*$ we have $2A \preceq A$.
   Hint. You will need to identify a particular element w_0 known to not be in A.

332. AMredAA

An interesting contrast to problem 331. Do we always have $A \preceq AA$? The answer is no. In fact it is possible to have

- A undecidable, but
- AA decidable

Give an example.

333. NonEmpUndec

Prove Theorem 40.17.

Hint. You can re-use a reduction function from Section 40.4! But write out the argument in complete detail to make sure you understand it.

334. UnclusionUndec

Prove that the following problem is undecidable.

> *Program Inclusion*
> INPUT: Two programs p_1 and p_2.
> QUESTION: Is $\mathsf{pfn}(p_1) \subseteq \mathsf{pfn}(p_2)$?

Remember that a function, partial or not, is a set of ordered pairs. So the question "$\mathsf{pfn}(p_1) \subseteq \mathsf{pfn}(p_2)$?" really does make sense.

Hint. Use the result in Section 40.5. (But don't try to do a reduction.)

Chapter 41

Undecidability Results about Arithmetic

In this section we describe some decidability and undecidability results about familiar mathematical structures. This is a huge subject so we will just focus on a couple of results, designed to convey some essential facts, impart some basic intuition, and hopefully spur you to explore more on your own.

Formalities We want to work with numbers and polynomials as strings. Elsewhere in these notes we have presented a one-to-one correspondence between the natural numbers and the finite strings over $\{0,1\}$. We will also assume—without presenting boring details—that each polynomial can be represented as a finite string of symbols.

In this way various sets of natural numbers are identified with sets of strings and various sets of polynomials are also sets of strings. In particular sets of natural numbers are *languages*, as are sets of polynomials. And so it really does make sense to speak of sets of numbers or sets of polynomials as being decidable, semi-decidable, etc, ... or not.

We will be interested in the following structures

- $\mathbb{N}$, the natural numbers: $\{0,1,2,\dots\}$.
- $\mathbb{Z}$, the integers: $\{\dots,-2,-1,0,1,2,\dots\}$.
- $\mathbb{Q}$, the rational numbers: $\{p/q \mid p,q \in \mathbb{Z}, \quad p/q \text{ in lowest terms}\}$.
- $\mathbb{R}$, the real numbers.

There is one important subtlety: arbitrary real numbers—in contrast to integers or rationals—cannot be viewed as finite objects, and so cannot be encoded as strings. One has to be careful when speaking of algorithmic questions concerning the real numbers!

41.1 Polynomial Solvability

Consider the problem of deciding whether a polynomial—perhaps with more than one variable—has roots (that is, values that make the polynomial evaluate to 0). The answer to this question can certainly depend on *where* we want the roots to be, whether we are thinking about integers, real numbers, etc. For example if we ask whether $x^2 - 2$ has any roots, the answer is "no" if we ask about roots in $\mathbb{Z}$ or $\mathbb{Q}$, but the answer is "yes" if we ask about roots in $\mathbb{R}$.

Polynomial solvability is a fundamental question about ordinary mathematics over various structures, and so understanding its decidability (or undecidability) is fundamental to understanding the limits of computation.

41.2 Polynomials over the Integers and Natural Numbers

Here is the decision problem we care about.

> PolyInt: *Polynomial solvability over* $\mathbb{Z}$
>
> INPUT: a multi-variable polynomial $p(\vec{x})$ whose coefficients are in $\mathbb{Z}$
>
> QUESTION: are there values $\vec{a}$ drawn from $\mathbb{Z}$ such that $p(\vec{a}) = 0$ holds?

The phrasing "does the equation $p(\vec{x}) = 0$ have roots in $\mathbb{Z}$?" is a little shorthand for the question above.

We are going to explore whether this problem is decidable, that is, we will ask whether there is a decision procedure which will take as input an arbitrary $p(\vec{x})$ over $\mathbb{Z}$ and answer yes or no whether $p(\vec{x})$ has any roots in $\mathbb{Z}$.

Here is a variation which will be convenient:

> PolyNat: *Polynomial solvability over* $\mathbb{N}$
>
> INPUT: a multi-variable polynomial $p(\vec{x})$ whose coefficients are in $\mathbb{Z}$
>
> QUESTION: are there values $\vec{a}$ drawn from $\mathbb{N}$ such that $p(\vec{a}) = 0$ holds?

Note that in this problem we allow the coefficients to range over the integers but we ask for solutions drawn from the natural numbers. This problem turns out to be more convenient technically than the POLYINT problem, but it is not really any easier (or harder!), as we now show.

PolyInt is no harder than PolyNat

Suppose we are given a polynomial p and we wish to know whether it has solutions over $\mathbb{Z}$. Build a new polynomial p' by replacing each integer variable x in p by the difference $(x_1 - x_2)$ of two new natural-number variables.

Then p has a solution over $\mathbb{Z}$ if and only if p' has a solution over $\mathbb{N}$.

PolyInt is no harder than PolyNat

Suppose we are given a polynomial q and we wish to know whether it has solutions over $\mathbb{N}$. Build a new polynomial q' by replacing each natural-number variable x in q by an expression $(x_1^2 + x_2^2 + x_3^2 + x_4^2)$ using new integer variables. This works by virtue of the theorem of Lagrange that says that any natural number can be written as the sum of 4 squares.

Then q has a solution over $\mathbb{N}$ if and only if q' has a solution over $\mathbb{Z}$.

So, even though PolyInt arises more naturally mathematically, for the purpose of studying decidability we might as well study PolyNat. This turns out to be more convenient, since it is easier to connect $\mathbb{N}$ with our known computability results.

41.1 Check Your Reading. *Explain why it would not be very interesting to consider the polynomial solvability problem where both coefficients and solutions were restricted to $\mathbb{N}$. (By the way, using subtraction is tantamount to allowing coefficients over $\mathbb{Z}$!)*

Semidecidability

Before proving the next result we note that for each k, is not hard to effectively enumerate the set of all k-tuples of natural numbers. One method is the following. For any fixed value n, consider the set of all k-tuples over $\mathbb{N}$ whose largest element is n. Clearly there are only finitely many such k-tuples, and we can enumerate these effectively. So to enumerate *all* k-tuples: first enumerate the k-tuples whose largest element is 0 (there is just one!); then enumerate the k-tuples whose largest element is $1; \ldots$, and so on.

Now we can show

41.2 Lemma. *Polynomial solvability over $\mathbb{Z}$ is semi-decidable.*

Proof. It will be sufficient, and more convenient, to show that polynomial solvability over $\mathbb{N}$ is semi-decidable. Here is a semi-decision procedure:

on input $p(\vec{x})$;
let k be the number of variables in p;
let $\vec{n}_0, \vec{n}_1, \ldots$ be an enumeration of all k-tuples over $\mathbb{N}$;
evaluate each $p(\vec{n}_i)$ in turn;
if and when any of these evaluates to 0, return 1.

///

This is a good time to acknowledge the following issue. We have phrased polynomial solvability as a decision problem, but of course what one *really* wants to do, given a polynomial, is *find* roots, not just get a yes/no answer as to whether roots exist.

But if you know a polynomial has roots, then, as the above proof makes clear, you can always find roots, just by searching. Of course in practice one would like a more efficient method, and of course such methods are well-studied. The important point here is that if we are able to prove that a given *decision problem* is undecidable then of course there will be no hope of actually *computing solutions*.

Diophantine Sets

Our main technique will be to construct a connection between polynomials and semi-decidable sets. The crucial definition is the next one.

41.3 Definition. A set $A \subseteq \mathbb{N}$ is a *Diophantine set*[a] if there is a polynomial $p(z, x_1, \ldots, x_k)$ with coefficients in $\mathbb{Z}$ such that

for every n, $n \in A$ if and only if the polynomial $p(n, x_1, \ldots, x_k)$ has a root.

[a] Diophantus was a 3rd century Greek mathematician who studied polynomial equations

By the way, it makes perfect sense to speak of Diophantine sets of ordered pairs, ordered triples, etc. We just use several variables $z_1, z_2, \ldots$ instead of just z. In this way we defined Diophantine *relations*, not just sets. For example, the relation "less than or equal" is Diophantine in this sense, using the polynomial $z_1 - z_2 + x$. We won't use these in this brief treatment of the subject, so we will always use "Diophantine" in the one-variable sense of Definition 41.3.

41.4 Examples.

1. The set of odd numbers is Diophantine; the polynomial is $z - (2x_1 + 1)$
2. The set of divisors of 100 is Diophantine; the polynomial is $zx_1 - 100$

3. The set of squares is Diophantine; the polynomial is $z - x_1^2$
4. The set of non-primes greater than 1 is Diophantine; the polynomial is $z - (x_1 + 2)(x_2 + 2)$

More examples are Problem 339

To make the connection between polynomial solvability and (un)decidability, remember that we have a one-to-one correspondence between the natural numbers and the finite strings over $\{0,1\}$. If $S = \{n_1, n_2, \dots\}$ is a set of natural numbers, we will write $\{b_{n_1}, b_{n_2}, \dots\}$ to refer to the corresponding set of bitstrings.

41.5 Check Your Reading. *Explain why, if a set $S = \{n_1, n_2, \dots\}$ of natural numbers is Diophantine then the corresponding language $\{b_{n_1}, b_{n_2}, \dots\}$ is semi-decidable.*

41.3 The DPRM Theorem

The following amazing theorem is the key result for us. The converse of the observation that every Diophantine set is semi-decidable was conjectured in 1953 by Martin Davis, and was worked on for years by many mathematicians, chiefly Davis, Hilary Putnam, Julia Robinson, and Yuri Matiyasevich; the final step in the proof was achieved by Matiyasevich in 1970. To reflect their collective efforts this theorem is often named using their initials:

41.6 Theorem (The DPRM Theorem)**.** *A set $S = \{n_1, n_2, \dots\}$ of natural numbers is Diophantine if and only if the corresponding language $\{b_{n_1}, b_{n_2}, \dots\}$ is semi-decidable.*

41.4 Using DPRM

Once we know Theorem 41.6 we can draw our undecidability conclusion.

41.7 Theorem. *Polynomial solvability over $\mathbb{Z}$ is undecidable, and polynomial solvability over $\mathbb{N}$ is undecidable.*

Proof. We will prove the result that polynomial solvability over $\mathbb{N}$ is undecidable. By the work we did in Section 41.1, specifically sub-Section 41.2, this will imply that polynomial solvability over $\mathbb{Z}$ is undecidable as well.

Choose a language that is semi-decidable but not decidable; for concreteness here let us consider SelfHalt. The DPRM Theorem says that there is a polynomial $p(z, x_1, \ldots, x_k)$ such that for every b_n:

$$b_n \in \text{SelfHalt} \text{ if and only if the polynomial } p(n, x_1, \ldots, x_k) \text{ has a root.}$$

So if there were a decision procedure d for polynomial solvability, the following would be a decision procedure for membership in SelfHalt:

on input b_n;
ask d whether the polynomial $p(n, x_1, \ldots, x_k)$ has a root;
return this answer

///

Where does the undecidability arise? A reasonable question is: if we bound the number of variables in our polynomials, or perhaps the degree of the polynomial, can we get decidability? Some results are known. Let's use n to count the number of variables in a polynomial, and d to stand for the total degree of a polynomial.

- It is important that we are considering *multi-variable* polynomials: Problem 338 asks you to prove that one-variable Diophantine solvability is decidable.
- Polynomial solvability is known to be undecidable when the number of variables is 11 or greater. It is not known whether this bound can be improved.
- Polynomial solvability is undecidable for polynomials of degree 4 or greater (indeed, any polynomial is equivalent to one of degree at most 4).
- Polynomial solvability is decidable for degree 2.

Polynomials over the Real Numbers

Let's turn to the question of whether a given polynomial has real-valued roots. Polynomial solvability over the reals is crucially important in many applications, most recently in robotics. For a nice treatment of some ways that solving polynomials is useful in robotics, see the great textbook *Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra,* by Cox, Little, and O'Shea [CLO92].

Now, since we are asking for *algorithmic* solutions to this question, there is an important subtlety in the definition of our problem. Specifically, we do *not* work with polynomials with real-valued coefficients, because such polynomials cannot, in general, be represented in a computer. This is because real numbers do not have finite representations (some do, but only countably many). Thus arbitrary polynomials with real coefficients can't even be given as input to a program!

So what we work with are polynomials with *rational number* coefficients, while we search for solutions in $\mathbb{R}$. This seems odd at first, but the coefficients-in-$\mathbb{Q}$ constraint is natural in practice and the answers-in-$\mathbb{R}$ allowance is the right one if we are reasoning about, for example, Euclidean space.

Polynomial solvability over $\mathbb{R}$

INPUT: a multi-variable polynomial $p(\vec{x})$ whose coefficients are in $\mathbb{Q}$

QUESTION: are there values $\vec{a}$ drawn from $\mathbb{R}$ such that $p(\vec{a}) = 0$ holds?

41.8 Theorem. *Polynomial solvability over* $\mathbb{R}$ *is decidable*

In fact this is an immediate corollary of a much stronger (and more important) theorem, which we give below as Theorem 41.10.

Polynomials over the Rational Numbers

No one knows whether there is a decision procedure to solve polynomials over the rationals!

Open Problem. Is polynomial solvability over $\mathbb{Q}$ decidable?

41.5 Logical Truth

Here is a significant generalization of polynomial solvability: the question of the truth or falsity of complex statements using arithmetic operators, such as "every even number is the the sum of two prime numbers."

Formally we speak of *arithmetic sentences* [1]. These are first-order logic sentences, using addition, multiplication, and ordering, and logic connectives and quantifiers such as $\wedge, \vee, \forall,$ and $, \exists$. For example

$$\forall x \ \exists y \ . \ (x < y) \wedge (y < x + 1)$$

is false in $\mathbb{N}$ and $\mathbb{Z}$, and is true in $\mathbb{Q}$ and $\mathbb{R}$.

On the other hand

$$\forall x \ (\ (0 < x) \ \rightarrow \ \exists y \ (y * y = x) \ )$$

is true in $\mathbb{R}$ and false in the other three structures.

[1] accent on the *third* syllable of "arithmetic," since it is used as an adjective.

The main observation for now is that the problem(s) of polynomial solvability are special cases of the problem(s) of logical truth. A concrete example will make this clear. To ask

$$\text{does } 2x^2 + 17y - 1 \text{ have a root in (for example) } \mathbb{Z}?$$

is the same as asking whether the arithmetic sentence

$$\exists x \, \exists y \ . \, 2 * (x * x) + 17y - 1$$

is a true sentence about (for example) $\mathbb{Z}$.

Logical Truth over the Natural Numbers and Integers

Logical Truth over $\mathbb{Z}$

INPUT: An arithmetic sentence S

QUESTION: Is S true about $\mathbb{Z}$?

The following is an immediate corollary of Theorem 41.7

41.9 Theorem. *Logical truth over $\mathbb{Z}$ is undecidable; logical truth over $\mathbb{N}$ is undecidable.*

Proof. In each case the result follows from the fact that polynomial solvability is a special case of logical truth. ///

In fact, the set of arithmetic sentences that are true in $\mathbb{N}$ is not only not decidable, it is not even semi-decidable, nor co-semi-decidable. To say in a precise way just how rich it is would require a long digression into hierarchies of logical complexity, which we will resist doing here.

Logical Truth over the Real Numbers

Things are quite different over the real numbers.

The real numbers are, individually, much more complicated beasts than integers are; for example a typical real number doesn't even have a finite representation.

So, an important first thing to note is that it is not immediately obvious that polynomial solvability over $\mathbb{R}$ is even semi-decidable. In contrast to $\mathbb{Z}$ and $\mathbb{N}$, there can be no "generate and test" semidecision procedure to check for $\mathbb{R}$-solvability of a polynomial. We cannot enumerate all possible solutions, again because individual real numbers are infinite objects.

But, amazingly, the space $\mathbb{R}$ of real numbers has much nicer logical behavior than that of $\mathbb{Z}$. The following famous theorem goes even beyond the result the polynomial solvability is decidable.

41.10 Theorem. *Logical truth over $\mathbb{R}$ is decidable.*

This was first proved by Alfred Tarski [Tar48]. As we hinted earlier, algorithms for deciding the truth of sentences about the reals are used in several applications, notably including robotics, and so improving the efficiency of such algorithms is still an area of research.

We immediately get:

41.11 Corollary. *Polynomial solvability over $\mathbb{R}$ is decidable.*

Logical Truth over the Rational Numbers

When it comes to logical truth, the rationals behave more like the integers than like the reals.

Logical Truth over $\mathbb{Q}$
INPUT: An arithmetic sentence S
QUESTION: Is S true about $\mathbb{Q}$?

41.12 Theorem. *Logical truth over $\mathbb{Q}$ is undecidable.*

As we mentioned above, the decidability of the (perhaps simpler?) problem of polynomial solvability over the rationals is unknown.

41.6 Summary

- Over $\mathbb{N}$:
  - Polynomial solvability is undecidable.
  - Therefore, logical truth is undecidable.
- Over $\mathbb{R}$:
  - Polynomial solvability is decidable.
  - Indeed, logical truth is decidable.
- Over $\mathbb{Q}$:
  - It is an open problem whether polynomial solvability is decidable.
  - Logical truth is undecidable.

A final remark We've talked about the integers, the rational numbers, and the reals. We haven't talked about the complex numbers $\mathbb{C}$. What about polynomial solvability there? Well, if you know about $\mathbb{C}$ at all you know that every polynomial has roots over $\mathbb{C}$, indeed that's the crucial fact about $\mathbb{C}$ in the first place. So the decision problem for polynomial solvability over $\mathbb{C}$ is trivial: the answer to every instance of the problems is "yes."

But what about the richer question of logical truth? The situation is the same as for logical truth over $\mathbb{R}$: it is decidable. Tha is to say, there is an algorithm that determines the truth or falsity of any arithmetic sentence when interpreted over $\mathbb{C}$.

41.7 Problems

335. EnumTuples

If we cared to, we could effectively enumerate all the k-tuples over $\mathbb{Z}$. Give a method to do this.

336. ZQPractice

a) Suppose we want to ask whether the polynomial

$$p = 3x^{17}y^3 - 47x^{12}z + 111xy^7 + 99$$

has solutions over $\mathbb{Z}$.

We described in the text a way to build a polynomial p such that p has roots in $\mathbb{Z}$ if and only if q has roots in $\mathbb{N}$. What is that q?

b) Suppose we want to ask whether the polynomial

$$p = 3x^{17}y^3 - 47x^{12}z + 111xy^7 + 99$$

has solutions over $\mathbb{N}$.

We described in the text a way to build a polynomial q such that p has roots in $\mathbb{N}$ if and only if q has roots in $\mathbb{Z}$ What is that q?

337. DivideConstant

This problem is about one-variable polynomials.

Prove: if p is $c_n x^n + c_{n-1}x^{n-1} + \ldots c_1 x + c_0$ and a is an integer root of p, then a divides c_0.

Hint. Under the assumption that a is a root of p, do some basic algebra to arrive at an equation that looks like $ab + c_0 = 0$ for some integer quantity b; argue that this means that a divides c_0.

Note. This is a simplified version of a stronger theorem, which states that if a/b is a *rational number* in lowest terms with $p(a/b) = 0$ then a divides c_0 and b divides c_n.

338. OneVarDec

Show that the problem of solvability of *one-variable* polynomials over $\mathbb{Z}$ is decidable.

Hint. Use Problem 337

339. DiophantineExamples

Each of the following polynomials $p(z,x_1,\ldots x_n)$ defines an easy-to-describe (Diophantine) set $A \subseteq \mathbb{N}$, by

$$A = \{n \in \mathbb{N} \mid \exists x,\ldots,x_k : p(n,x_1,\ldots x_k) = 0\}$$

Describe each set. You needn't prove your answer, just generate enough instances so that you are confident that you know the set.

a) $(z-17)(x_1+1)$

b) $z+x_1 = 10$

c) $z-7x_1$

d) $(z-2x_1)(z-3x_2)$

e) $z-(2x+3)(y+1)$

f) $x_1^2 - z(x_2+1)^2 - 1$ *Hint. This one isn't easy. Note for starters that z can't be a perfect square...(why?)*

340. FiniteDiophantine

Let $X = \{a_1,\ldots,a_k\}$, $k \geq 0$, be a finite set. Show that X is Diophantine.

341. CofiniteDiophantine

Let X be a finite set. Show that the complement $\mathbb{N} - X$ is Diophantine.

342. IntersectionDiophantine

a) Suppose $p(z,x_1,\ldots x_k)$ and $q(z,y_1,\ldots y_p)$ are polynomials.

Explain why, for any n and tuples $(a_1,\ldots,a_k)$ and $(b_1,\ldots,b_p)$,

$$p(n,a_1,\ldots a_k) = 0 \text{ and also } q(n,b_1,\ldots b_p) = 0$$

if and only if

$$(p(n,a_1,\ldots a_k))^2 + (q(n,b_1,\ldots b_p))^2 = 0$$

b) Using this, prove (without quoting Theorem 41.6) that the intersection of two Diophantine sets is Diophantine.

343. UnionDiophantine

Prove (without quoting Theorem 41.6) that the union of two Diophantine sets is Diophantine.

Hint. Proceed in the spirit of Problem 342, that is, find an appropriate way to combine polynomials.

344. ComplementDiohpantine

Prove or disprove: The complement of a Diophantine set is Diophantine.

Bibliography

[Ard61] Dean N Arden. Delayed-logic and finite-state machines. In *Proceedings of the Second Annual Symposium on Switching Circuit Theory and Logical Design*, pages 133–151. IEEE, 1961. http://dx.doi.org/10.1109/FOCS.1961.13.

[CLO92] David Cox, John Little, and Donal O'Shea. *Ideals, varieties, and algorithms*, volume 3. Springer, 1992.

[EKR82] Andrzej Ehrenfeucht, Juhani Karhumäki, and Grzegorz Rozenberg. The (generalized) Post Correspondence Problem with lists consisting of two words is decidable. *Theoretical Computer Science*, 21(2):119–144, 1982.

[HMU06] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[Koz97] Dexter C. Kozen. *Automata and Computability*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1997.

[Rog67] Hartley Rogers. *Theory of recursive functions and effective computability*, volume 5. McGraw-Hill New York, 1967.

[Sip96] Michael Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1st edition, 1996.

[Sud97] Thomas A. Sudkamp. *Languages and Machines: An Introduction to the Theory of Computer Science*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[Tar48] Alfred Tarski. *A Decision Method for Elementary Algebra and Geometry*. University of California Press, 1948.

[Tur37] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London mathematical society*, 2(1):230–265, 1937.