Notes on the Foundations of Computer Science

Dan Dougherty
Computer Science Department
Worcester Polytechnic Institute

October 5, 2018 – 17 : 06

Contents

Introduction

The notes are designed for teaching various courses in the foundations of computer science.

There is a canonical set of topics that appears reliably in every such course: finite automata, context-free grammars, Turing machines and undecidability. These notes explore each of these topics as well. There is some novelty in the presentation here in that in addition to treating "foundations" per se we try to provide the student with an intellectual *toolkit* for the design and analysis of systems. This is a running theme of the course.

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

Part I

Mathematics Background

1 Relations and Functions

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

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 *range* 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]

1.1 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.

[1] Unfortunately some authors use the word "range" for what we have called the "codomain." You have to be careful when you read.

1.3 Properties of Composition

- 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.

1.2 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!)

1.4 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.

1.4.1 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.

1.3 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.")

1.4 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.*

1.5 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 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 define a one-sided inverse on the other side.

1.5 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. 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: $f(g(b)) = id_B(b) = b$.

   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.

2. 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 $a_1 = a_2$.

   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 range of A, choose $g(b)$ to be any a such that $f(a) = b$. If b is not in the range of A, choose $g(b)$ to be a_0. It is easy to check that $g \circ f = id_A$.

///

1.5.1 "Pointless" Reasoning

It can be efficient and satisfying to reason about functions as algebraic objects rather than working with "points" in sets. Theorem 1.5 is a great tool for this. Here is a series of examples.

The following result is often quite useful.

1.6 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.

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.

1.6 Characteristic Functions and Subsets

Characteristic functions are a different way of thinking about the idea of "subset."

Fix a set 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

1.7 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.*

1.8 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 1.7. 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 slogan

> *Subsets and characteristic functions are the same things, just in different notation.*

But beware, these are pure mathematical functions, and there need not be any "algorithm" to compute them. We are certainly not claiming that for any given subset there is actually a computer program that computes the characterisitc function!

A remark about notation. In mathematics one often sees Y^X to denote the set of all functions from a set X to a set Y. That is, mathematicians sometimes extend the exponent notation that you are familiar with to the setting of sets and functions between them. This is a "pun", which can be confusing. But the last part of Exercise 1 explains why this notation is suggestive.

In fact some authors use the notation 2^U to denote the set $Pow(U)$ of all subsets of U. Here is why. People sometimes use "2" as syntact sugar for the set $\{0,1\}$. So when we write 2^U this is just syntactic sugar for $\{0,1\}^U$. That is, when we write 2^U we are really denoting the set of characteristic functions over U. The relationship between "the set of characteristic functions over U" and "the set of all subsets of U is so strong that some authors use the same notation, 2^U, to denote them both.

1.7 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.

1.9 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.

1.8 Composition and Inverse on Relations

1.10 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 $R \circ P$ *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 $(x,y) \in R$ and $(y,z) \in P$.*

1.11 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 is no trouble (Definition 1.11) in defining the inverse of R. But suppose f is a *function* from A to B. Then, since f is a (special kind of) relation, sure, there is an inverse of f. *But the inverse of f, in the relational sense, might not be a function!*

1.12 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 1.10) really is the same the composition as functions (that is, via Definition 1.1).*

1.9 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.)

1.13 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.

1.9.1 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 $\mathcal{P} = \{A_1, A_2, \ldots\}$ 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.

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

1.9.2 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.

1.10 Exercises

***Exercise* 1.** For this exercise, 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?

***Exercise* 2.** 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 $g(x) = 2^x$

***Exercise* 3.** 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 $g(x) = 2^x$

***Exercise* 4.** The goal of this exercise 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.

Now repeat the previous exercise, but work with finite sets rather than $\mathbb{N}$. That is, for each of the four combinations of "injective and surjective, or not", find examples of functions $f : A \to B$ where you can choose the A and B.

In each case, try to find the *smallest* sets A and B that support an example.

***Exercise* 5.** Note that in Exercise 4 you were *not* asked to find examples that are functions from a finite set to itself. For which of the four parts of that exercise could we have asked for a function $f : A \to A$, with A finite, and for which would it be impossible?

Each part of the following exercise can be proved by a rather tedious argument that traces through the effect of functions on points. But once we have proved the algebraic results in Theorem 1.5 we have nice tidy "calculational" proofs available. Two of them are done for you.

***Exercise* 6.** Let $f : A \to B$ and $g : B \to C$.

1. If f and g are injective then $(g \circ f) : A \to C$ is injective.

2. If $(g \circ f) : A \to C$ is injective then f is injective. (It does not follow that g is injective, however.)

3. If f and g are surjective then $(g \circ f) : A \to C$ is surjective.

4. If $(g \circ f) : A \to C$ is surjective then g is surjective. (It does not follow that f is injective, however.)

Proof.

1. Suppose f and g are injective, we want to prove that $(g \circ f)$ is injective. We first have to handle the trivial special case when $A = \emptyset$: the result follows from the fact that $(g \circ f)$ is automatically injective since the domain is $\emptyset$. For the non-trivial case it suffices to find an $h : C \to A$ such that $h \circ (g \circ f) = id_A$. By our assumptions about f and g we have h_1 and h_2 such that $h_1 \circ f = id_A$ and $h_2 \circ g = id_B$. *Draw a picture!* We can define our desired h to be $h_1 \circ h_2$. To see that this works, we calculate:

   $$h \circ (g \circ f) = (h_1 \circ h_2) \circ (g \circ f) = (h_1 \circ (h_2 \circ g) \circ f) = (h_1 \circ id_B \circ f) = (h_1 \circ f) = id_A.$$

2. Assume $(g \circ f) : A \to C$ is injective. We want to show that f is injective. Again the case when $A = \emptyset$ is easy since f is a function with domain $\emptyset$. Otherwise it suffices to exhibit some h such that $h \circ f = id_A$. By the fact that $(g \circ f)$ is injective there is an h_1 such that $h_1 \circ (g \circ f) = id_A$. But then we may take our desired h to be $(h_1 \circ g)$. *Again, draw a picture to make this clear.*

3. Left for you.

4. Left for you.

///

Next, 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 left inverse g_1 and a right 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:

***Exercise* 7.** Prove: For an arbitrary $f : A \to B$, if there exists g_1 and g_2 with $g_1 \circ f = id_A$ and $f \circ g_2 = id_B$, then f has an inverse. In fact, in this case, g_1 and g_2 must be the same function, which is indeed the inverse.

Hint. Use pointless reasoning!

***Exercise* 8.** Suppose $f_1 : A_1 \to B_1$ and $f_2 : A_2 \to B_2$ are each injective. Exhibit an injective function $f : (A_1 \times A_2) \to (B_1 \times B_2)$; explain why it is injective.

Define f by: $f(a_1, a_2) = (f_1(a_1), f_2(a_2))$.

To show this is injective: suppose $f(a_1, a_2) = f(a_1', a_2')$; we want to show that $(a_1, a_2) = (a_1', a_2')$.

If $f(a_1, a_2) = f(a_1', a_2')$, this means that $(f_1(a_1), f_2(a_2)) = (f_1(a_1'), f_2(a_2'))$; which in turn means that $(f_1(a_1)) = (f_1(a_1'))$ and also $(f_2(a_2)) = (f_2(a_2'))$. Since f_1 is injective we conclude that $a_1 = a_1'$ and since f_2 is injective we conclude that $a_2 = a_2'$. Therefore we have $(a_1, a_2) = (a_1', a_2')$ as desired.

***Exercise* 9.** Suppose $f : A \to B$ and $g : B \to C$ are functions.

As a review for yourself, make a full taxonomy of the implications you can make concerning injectivity and surjectivity and composition. That is, for each way of filling in the blanks below with the words "injective" or "surjective", decide whether the resulting statement is either true or false. If a statement is true give a proof, ideally using left- and right-inverses. If it is false give a specific pair of functions f and g making the statement false (don't forget to say what A, B, and C are).

1. If $(g \circ f)$ is ______ then g has this property as well.
2. If $(g \circ f)$ is ______ then f has this property as well.
3. If g is ______ then $(g \circ f)$ has this property as well.
4. If f is ______ then $(g \circ f)$ has this property as well.

***Exercise* 10.** A two-sided inverse for a function $f : A \to B$ is a function $g : B \to A$ such that $(g \circ f)$ is the identity on A and $(f \circ g)$ is the identity on B. Now, suppose we had a situation where f had a left inverse and also had a right inverse. Does that automatically mean that f has a two-sided inverse?

Prove that the answer is yes. In fact prove that if g_1 is a left inverse for f and g_2 is a right inverse for f, then in fact $g_1 = g_2$.

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 $g_1 = g_2$.

***Exercise* 11.** 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.

***Exercise* 12.**

1. List all the binary relations on the set $\{1,2\}$
2. How many binary relations are there on a set with n elements?

***Exercise* 13.** Give an example to show that relation composition is not commutative.

***Exercise* 14.** For each relation R from Examples 1.9 determine which of the above properties it enjoys.

***Exercise* 15** (Functions and equivalence relations)**.** 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 $f(a_1) = f(a_2)$. A moment's thought shows 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

***Exercise* 16.** 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.

***Exercise* 17.** 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.

***Exercise* 18.** 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.)

***Exercise* 19.** 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.

2 Strings, Languages, and Regular Expressions

2.1 Definition. *An* ***alphabet*** *is any finite set. We typically use* Σ *(the Greek capital letter "Sigma") to denote an alphabet, and often use the word "character" 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^**. The unique string with length 0, the empty string, is denoted* λ*. Sometimes strings are called "words."*

A ***language*** *over alphabet* Σ *is any set of strings over* Σ*. That is, a language over alphabet* Σ *is a subset of* Σ^**. The empty set is a language, denoted* $\emptyset$*.*

Examples: The binary alphabet Σ_2 is just $\{0,1\}$. Here are some strings over Σ_2: 01, 111110, λ, 101010101010101010. The ASCII alphabet is the set of 128 symbols you are presumably familiar with. A text file is just a string over the ASCII alphabet.

The main operation on strings is **concatenation.** Just as with multiplication on numbers we typically denote concatenation by juxtaposition: if x and y are strings then xy is their concatenation. For this reason, we usually don't parenthesize concatenations.

The analogy between concatenation and multiplication goes further. Concatenation is associative: $(xy)z$ is the same string as $x(yz)$. Concatenation has an identity element, namely the empty string: $\lambda x = x\lambda = x$. But concatenation is not commutative: usually $xy \neq yx$.

Let x and y be strings. We say that x is a *prefix* of 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 $z_1 x z_2 = y$.

Do not confuse the language $\emptyset$ with the string λ! They are not even the same type of object: the latter is a string, the former is a set of strings. It can help your intuition to think of any set S as being like a box, that has the elements of S inside. In particular a language is like a box, with a collection of 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.

2.1 Why Do We Care About Languages?

The notion of "language" as we have defined it sounds somewhat narrow at first. 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 $\Sigma = \{0,1\}$
- A program itself, its source code, is a file. Thus a program is a string over $\Sigma =$ the ASCII alphabet.

  Of course we could think of a program source as a sequence of bits, too. There's more than one way to formalize a concept. But 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.

2.1.1 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 a thermostat, or a robot, 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.

2.2 Ordering Strings

Let Σ be an alphabet. Let us order Σ in some arbitrary way, as $\Sigma = [a_1, \dots, a_n]$. Once we do that, there is a natural way to order the set Σ^* of strings.

2.2 Definition. *(Lexicographic Order on Strings) If the alphabet Σ is totally ordered, this induces the* lexicographic order $\prec$ *on Σ^* 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.*

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.

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

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

This puts Σ^* is a natural bijective correspondence with $\mathbb{N}$, and so we can refer to the "nth string w_n if it is convenient.

***Exercise* 20.** Show that $x = y$ if and only if $x \not\prec y$ and $y \not\prec x$.

2.3 Example. (The Most Important Example) The case when $\Sigma = \{0, 1\}$, with $0 < 1$, works out particularly nicely. Take a string $x \in \Sigma^*$, and suppose that x is the nth string in the above ordering, that is, x is w_n. If we append a 1 to the *left* of x, then view the resulting string $1x$ as a number encoded in binary, then the encoded number will be precisely $n + 1$.

For example, the empty string is w_0, the 0th string in the encoding, since when we prepend a 1, this is the number 1 in binary. If we wanted to know where 010 occurs in the ordering, we prepend a 1 to get 1010, decode that in binary to get the number 10, which tells us that 010 is the w_9, the 9th string.

What is w_{10}, the 10th string? The encoding of 11 without leading 0s is 1011, so the 10th string is 011.

2.3 Operations on Languages

Suppose A and B are languages. Then we define the languages $A \cup B$, $A \cap B$, and $\overline{A}$ (union, intersection, and complement) in the usual way, as with all sets. But since A and B are sets of strings, there are two other operations that make sense:

- AB is the **concatenation** of A and B, the result of taking all possible concatenations of the strings from A then from B:
$$AB \overset{\text{def}}{=} \{xy \mid x \in A,\ y \in B\}$$
- A^* is the **Kleene-closure**[2] or **Kleene-star** of A, the result of taking all possible concatenations of any finite number of strings from A:
$$\begin{aligned} A^* &\overset{\text{def}}{=} \{x_1x_2\ldots x_n \mid n \geq 0;\ \text{each } x_i \in A\} \\ &= \{\lambda\} \cup A \cup AA \cup AAA \cup \ldots \end{aligned}$$

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 characters considered one-element strings, then the Σ^* notation does the right thing even when viewed as a Kleene-closure.

[2]named for Stephen Kleene, a pioneer of the subject. Everyone pronounces his last name "cleany." Except Kleene himself, who 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?

2.3.1 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

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

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

2.4 Regular Expressions and Pattern Matching

Regular expressions are a formalism used for *matching*. A regular expression E can be seen as a search pattern: given such an expression and a string x we can ask whether x matches E. A classical use case is that x is a text file and E captures a search we want to perform. A more recent use case is when x is a DNA or protein sequence.

Below we define the syntax of regular expressions over a given alphabet Σ. Each regular expression E determines a language $L(E)$. These are the strings that "match" E.

2.4 Definition. *Let Σ be an alphabet. The regular expressions over Σ and the languages they denote are defined inductively as follows.*

- *For each $a \in \Sigma$, a is a regular expression.*
  It denotes the one-element language consisting of the length-1 string a: $L(a) = \{a\}$.
- *λ is a regular expression.*
  It denotes the one-element language consisting of the empty string λ: $L(\lambda) = \{\lambda\}$.
- *$\emptyset$ is a regular expression.*
  It denotes the empty language: $L(\emptyset) =$ the empty set.
- *If E_1 and E_2 are regular expressions then $(E_1 \cup E_2)$ is a regular expression.*
  It denotes the union of the languages of E_1 and E_2: $L(E_1 \cup E_2) = L(E_1) \cup L(E_2)$.
- *If E_1 and E_2 are regular expressions then (E_1E_2) is a regular expression.*
  It denotes the concatenation of the languages of E_1 and E_2: $L(E_1E_2) = L(E_1)L(E_2)$.
- *If E is a regular expressions then E^* is a regular expression.*
  It denotes the Kleene closure 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, above, $L(E_1E_2) = L(E_1)L(E_2)$, the expression E_1E_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-closure.

When we write $L(E_1 \cup E_2) = L(E_1) \cup L(E_2)$, the expression $E_1 \cup E_2$ 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.

To help see the difference notice that there can be many different regular *expressions* denoting the same language (for example $E \cup \emptyset$ will certainly denote the same language as E). In fact Section 2.6 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 $\cup$ and concatenation; (ii) concatenation has higher precedence than $\cup$. 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

$$((ab)(c^*)) \cup d$$

we may write

$$abc^* \cup d$$

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

- a^* denotes the set of all finite sequences of as (including the empty string)
- $(a \cup b)^*$ denotes the set of all finite strings over $\{a, b\}$
- $a^* \cup b^*$ denotes the set of all bitstrings that are either all-a or all-b.
- $(a \cup b)^* b (a \cup b)^* b (a \cup b)^*$ denotes the set of all bitstrings with at least two occurrences of b.
- $((a \cup b)(a \cup b))^*$ denotes the set of all even-length bitstrings.
- $(a \cup b)^* aab (a \cup b)^*$ denotes the set of all bitstrings with the string aab as a substring.
- $(ab \cup b)^* (\lambda \cup a)^*$ denotes the set of bitstrings 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. See Section 11.5.2 for a discussion, after we've explored regular expressions a bit more.

2.5 Languages vs Expressions

There are two points to be aware of here, each very important.

Expressions Are Not the Same as Languages There is a difference between a *language,* which is a (possibly infinite) set of strings, and an *expression,* which is little finite chunk of syntax. Expressions are what we can write down, send to one another, use as input to programs, etc. Languages live in the mathematical universe and, if they are infinite, cannot be directly examined, transmitted, or computed over.

Some Languages Cannot be Named by Expressions Every regular expression names a language, but we will see later that *not every language* can be described in this simple inductive way, that is, not 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.

But it makes perfect sense to talk about, for example, XY even when X and Y are arbitrary languages.

For example, let $P = \{10, 11, 101, 111, \dots\}$ be the set of bit strings coding a prime number. This P is a perfectly good language, but one can show that there is no regular expression you can write down that matches P.

2.6 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: $(XY)Z = X(YZ)$.

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

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

and the rules

$$F \cup EG \subseteq G \text{ implies } E^*F \subseteq G$$
$$F \cup GE \subseteq G \text{ implies } FE^* \subseteq G$$

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 answer is that *any* equation about languages that is universally true follows (in a way that can be made formal, omitted here) from this set of laws. A proof of that is beyond the scope of this course.

If we rewrite a regular expression E into another regular expression E' using the equations corresponding to the laws in then we can be sure that $L(E) = L(E')$.

For example we know that $a^*(b \cup (ac))$ and $a^*b \cup a^*ac$ define the same language, due to the language equality $X(Y \cup Z) = XY \cup XZ$

2.7 Example. Suppose x is any string over some Σ, say $x = c_1c_2 \dots c_n$ where each $c_i \in \Sigma$. Then x itself can be viewed 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. Of course this *RegExp* denotes the singleton language $\{x\}$.

To be perfectly precise, when we view x as a regular expression we are really speaking of the fully-parenthesized iteration of concatenations $(\dots((c_1c_2)c_3)\dots c_n)$. But there is no need to be *that* pedantic: we just write $c_1c_2 \dots c_n$.

A takeaway from Example 2.7 is that we could have taken as one of the base-cases for the definition of *RegExps* that *for any string $x \in \Sigma^*$, x is a regular* *expression. And* $L(x) = \{x\}$. Would that be better? It's a matter of taste. Some people might find that definition more natural (we don't tend to care much about 1-element strings, do we?) But some people might be drawn to having a "minimal" set of basic things, just single chars from Σ, and getting more complex things automatically.

In that spirit, see Exercise 35.

2.7 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.

2.8 Exercises

***Exercise* 21.** What's the difference between an alphabet and a language?

***Exercise* 22.** Is $\{\lambda\}$ a language? Is $\{\emptyset\}$ a language? Is $\emptyset$ a language?

***Exercise* 23.** Consider the union operation on languages. Is it associative? Is it commutative? Does it have an identity element, ie a language X such that for all languages A we have $X \cup A = A$ and $A \cup X = A$?

Answer the same questions for intersection. Answer the same questions for concatenation.

***Exercise* 24.** Suppose that: Σ is an alphabet, x and y are strings over Σ, A and B are languages over Σ. Recall that λ denotes the empty string.

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.

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

***Exercise* 25.** 1. Can a language be infinite?

2. Can an element of a language $A \subseteq \Sigma^*$ be infinite?
3. Does it make sense to talk about taking the union of two strings?

***Exercise* 26.** Let $A = \{aa, bb\}$ and $B = \{\lambda, b, ab\}$

1. List the strings in the set AB
2. List the strings in the set 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.

***Exercise* 27.** Under the lexicographic order of the strings in $\{0,1\}^*$, what number string is 101? How about 0001?

What is the 99th string?

***Exercise* 28.** Prove or disprove

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

2. $A(B \cup C) \supseteq AB \cup AC$

3. $A(B \cap C) \subseteq AB \cap AC$

4. $A(B \cap C) \supseteq AB \cap AC$

Hint. Three of those are true and one is false.

***Exercise* 29.** 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

***Exercise* 30.** Let A and B be arbitrary languages.

1. Prove $(A \cup B)^* = A^*(BA^*)^*$

2. Prove or disprove: $(A \cup B)^* = B^*(AB^*)^*$

***Exercise* 31.** Fix $\Sigma = \{a, b\}$.

For each given regular expression E and string x decide whether $x \in L(E)$. Defend your answer.

(This exercise 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.)

1. E is $a^*ba^*b(a \cup b)^*$
   (a) Is ab in $L(E)$?
   (b) Is $aaabbaaa$ in $L(E)$?
   (c) Is $babab$ in $L(E)$?
2. E is $a^*(a^*ba^*ba^*)^*$
   (a) Is ab in $L(E)$?
   (b) Is $aaabbaaa$ in $L(E)$?
   (c) Is $babab$ in $L(E)$?
3. E is $(b \cup ab)^*(\lambda \cup a)$
   (a) Is ab in $L(E)$?
   (b) Is $aaabbaaa$ in $L(E)$?
   (c) Is $babab$ in $L(E)$?

***Exercise* 32.** For each of the following regular expressions, give two strings which are in the language they define, and give two strings which are in not the language they define. Assume that the alphabet Σ is always $\{a, b\}$.

1. a^*b^*
2. $a(ba)^*b$
3. $a^* \cup b^*$
4. $(aaa)^*$

5. $aba \cup bab$

6. $(\lambda \cup a)b$

***Exercise* 33.** For each of the following regular expressions α give a shortest non-λ string in $L(\alpha)$. (There may be more than one answer to a given question...)

1. $(a \cup bb)a^*b$.

2. $(bb \cup (ab \cup ba)(aa \cup bb)^*(ab \cup ba))^*$

3. $((aa \cup bb)^* \cup (aab \cup bba)^*)^*$

***Exercise* 34.** Fix $\Sigma = \{a,b\}$. For each language $K \subseteq \Sigma^*$ described below, construct a regular expression E with $L(E) = K$.

1. The set of strings of odd length.

2. $\{x \mid x \text{ does not contain } ba\}$

3. $\{x \mid x \text{ has odd length and contains } ba\}$

4. $\{x \mid x \text{ does not contain } aaa\}$

5. $\{x \mid x \text{ contains } aa \text{ at least twice}\}$ (Be careful about the string *aaa*: we do want to include this.)

6. The set of strings that contain exactly three occurrences of b.

7. The set of strings starting with a and ending with bb.

8. The set of strings that contain the substring abb.

***Exercise* 35.** Show that we actually don't need to allow λ as a basic regular expression. That is, there is a regular expression E constructible using the other regular expression operators, such that $L(E) = \{\lambda\}$

***Exercise* 36.** Fix $\Sigma = \{a,b,1,2,-\}$. (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.

***Exercise* 37.** Fix $\Sigma = \{0, 1, 2, \ldots 9, -, .\}$. 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.

***Exercise* 38.** The syntax for regular expressions does not explicitly provide for intersection nor for complement. But 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. In this exercise 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 2.5, give a regular expression for the complement of that language.
2. For each pair of languages in Examples 2.5, 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!

***Exercise* 39.** *Adapted from Kozen [Koz97]* For each pair of regular expressions below, say whether they are equal (i.e., denote the same language). If so, give a proof. If not, give an example of a string in one but not the other.

1. $(0 \cup 1)^*$ and $0^* \cup 1^*$
2. $0(120)^*12$ and $01(201)^*2$
3. $\emptyset^*$ and λ^*
4. $(0^*1^*)^*$ and $(0^*1)^*$
5. $(01 \cup 0)^*0$ and $0(10 \cup 0)^*$

***Exercise* 40.** *Adapted from Kozen [Koz97]* 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.

1. $\emptyset^*$ and λ^*
2. $(a \cup b)^*$ and $a^* \cup b^*$
3. $(ab)^*a$ and $a(ba)^*$
4. $(bba)^*bb$ and $bb(abb)^*$
5. $(a^*b^*)^*$ and $(a^*b)^*$
6. $(ab \cup a)^*$ and $(ba \cup a)^*$
7. $a^*ba^*b(a \cup b)^*$ and $(a \cup b)^*b(a \cup b)^*b(a \cup b)^*$
8. $a^*ba^*b(a \cup b)^*$ and $a^*(a^*ba^*ba^*)^*$

***Exercise* 41.** *Closure under reverse*

Prove that if L is regular then so is L^R (the set of reversals of string in L), using regular expressions. That is, suppose you are given a regular expression denoting L and show how to write a regular expression denoting L^R. Your construction should be a recursive algorithm over regular expressions.

***Exercise* 42.** 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.

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

Concatenation Testing

INPUT: a string w

QUESTION: is $w \in AB$?

***Exercise* 43.** 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 $w \in A^*$?

***Exercise* 44.** 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 $A^* \subseteq B$.

***Exercise* 45** (*A useful equation*)**.** Let A, B and C be arbitrary languages. Prove:

> if $B \cup AC \subseteq C$ 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 forall k, $A^kB \subseteq C$. Now, prove that statement by induction on k.

***Exercise* 46.** The following result is known as Arden's Lemma [Ard61].

Theorem. *Let A be a language such that $\lambda \notin A$. Then the equation $X = AX \cup B$ has the unique solution $X = A^*B$.*

We will prove this result a little later, when we need to use it. For now, as good practice,

1. Verify the result for yourself in several cases (make up sample A and B and check the solution).
2. Suppose we removed the condition "$\lambda \notin A$". Would it still be true that A^*B is a solution to the equation? Give an example to show why the condition is needed.

3 Cardinality

Everyone knows what it means to say that A and B "have the same size" when A and B are finite sets. The big idea in this section is to develop a sensible way of talking about this idea when A and B are allowed to be *infinite* sets. The important thing is that the theory developed here applies *uniformly* to both finite and infinite sets.

The ideas are due to the mathematician Georg Cantor. Do yourself a favor and have a look at the first few paragraphs of the Wikipedia article on Cantor: http://en.wikipedia.org/wiki/Georg_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.

3.1 Key Definitions

3.1 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 *$B \preceq A$.*

  This is equivalent to saying 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" infinite set.

3.2 Definition (Countable and Uncountable)**.** *A set A is* countable *if $A \preceq \mathbb{N}$. A is* uncountable *otherwise.*

Note that any finite set is countable. Sometimes we may say that a set A is *countably infinite* to mean that it is countable, and not finite. [3]

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.

3.2 Basic Tools

The following lemma is a super-convenient tool. We proved it in Lemma 1.6, but it is worth repeating here.

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. A.*

This means that when A is not empty, $A \preceq B$ if and only if there is a surjective function from B to A.

Notice that the set A is infinite precisely if $\mathbb{N} \preceq A$.

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.

[3]Some authors reserve the word countable for infinite sets that are $\preceq \mathbb{N}$, in other words, for them, finite sets are not called "countable. But the more inclusive terminology here is more typical.

3.3 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 (Exercise 47).

The next section, 3.3, will give lots of examples of sets X such that $X \preceq \mathbf{N}$, that is, countable sets. We do **not** have any examples yet of infinite sets A and B with $A \prec B$, that is, of two infinite sets A and B where there is no possible injective function from B to A. If there were not such examples, this whole subject of cardinality would be dumb. But it takes a really deep idea to exhibit such a phenomenon: this was Cantor's brilliant insight. You can skip ahead to Section 3.4 below to be reassured that we aren't spinning our wheels, or you can tackle Section 3.3 to get more practice with cardinality first.

3.3 Some Examples

1. Very easy examples: $\{a,b,c\} \approx \{1,3,5\}$; $\{a,b,c\} \preceq \{1,3,5,100\}$; indeed $\{a,b,c\} \prec \{1,3,5,100\}$; $\{1,2,\ldots 2^{10}\} \prec \mathbb{N}$.

2. $\mathbb{N} \preceq \mathbb{Z}$ Proof: the function $f : \mathbb{N} \to \mathbb{Z}$ defined by $f(n) = n$ is injective.

   More generally, whenever $A \subseteq B$ then we have $A \preceq B$, via the function that maps any $a \in A$ to itself.

3. Let $E = \{0,2,4,\ldots\}$ be the set of even natural numbers. Then $E \approx \mathbb{N}$. Proof: we have to show $E \preceq \mathbb{N}$ and $\mathbb{N} \preceq E$. The first is true since $E \subseteq \mathbb{N}$. The second is witnessed by the function $f : \mathbb{N} \to E$ defined by $f(n) = 2n$.

4. $\mathbb{Z} \preceq \mathbb{N}$ Proof : 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.

   Thus: $\mathbb{Z}$ is countable.

5. $\mathbb{N} \approx \mathbb{Z}$. This is a re-statement of two facts we just established, that $\mathbb{N} \preceq \mathbb{Z}$ and $\mathbb{Z} \preceq \mathbb{N}$.

6. $\mathbb{N}^2 \approx \mathbb{N}$. (Here $\mathbb{N}^2 = \mathbb{N} \times \mathbb{N}$ denotes the set of ordered pairs (a,b) of natural numbers.)

   Proof: First, to see that $\mathbb{N} \preceq \mathbb{N}^2$, we consider the function $f : \mathbb{N} \to \mathbb{N}^2$ defined by $f(z) = (z,0)$. [Of course there is nothing special about the use of "0" here ...]. This is clearly injective.

   Next, to see that $\mathbb{N}^2 \preceq \mathbb{N}$ we 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.

   The fact that $\mathbb{N}^2 \preceq \mathbf{N}$ says that $\mathbb{N}^2$ is countable.

7. We can generalize the previous example. Let $\mathbb{N}^k = \mathbb{N} \times \mathbb{N} \times \cdots \times \mathbb{N}$ denote the set of ordered k-tuples. Then $\mathbb{N}^k \approx \mathbb{N}$.

   Certainly $\mathbb{N} \preceq \mathbb{N}^k$, by the injective function mapping n to $(n,0,\ldots 0)$. 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.

8. $\mathbb{Z}^2 \preceq \mathbb{N}$.

   Proof: we have shown that $\mathbb{Z} \preceq \mathbb{N}$. By Exercise 8 we conclude that $\mathbb{Z}^2 \preceq \mathbf{N}^2$ So $\mathbb{Z}^2 \preceq \mathbf{N}^2 \preceq \mathbb{N}$ using part 6, and so by Lemma 3.3 we have $\mathbb{Z}^2 \preceq \mathbb{N}$.

   Thus: $\mathbb{Z}^2$ is countable.

9. $\mathbb{Z}^2 \approx \mathbb{Q}$.

   First, to show $\mathbb{Q} \preceq \mathbb{Z}^2$. It will be easiest to exhibit a *surjective* function from $\mathbb{Z}^2$ to $\mathbb{Q}$. Suppose we have a pair $(p,q) \in \mathbb{Z}^2$. We can think of this as the rational number $\frac{p}{q}$. Be careful: two different fractions $\frac{a}{b}$ and $\frac{b}{c}$ can denote the same rational number, namely if $ac = bd$. This means that the function $f : \mathbb{Z}^2 \to \mathbb{Q}$ defined by $f(p,q) = \frac{p}{q}$ is not injective. But that doesn't matter, it is clearly surjective. So this function f witnesses the fact that $\mathbb{Q} \preceq \mathbb{Z}^2$.

Now, to show that $\mathbb{Z}^2 \preceq \mathbb{Q}$, we could try to construct explicitly an injective function from $\mathbb{Z}^2$ to $\mathbb{Q}$ (or a surjective function from $\mathbb{Q}$ to $\mathbb{Z}^2$). **Or** we can be clever (= lazy), and chain together things we already know. We We already showed above that $\mathbb{Z}^2 \preceq \mathbb{Z}$. And we certainly know that $\mathbb{Z} \preceq \mathbb{Q}$, because $\mathbb{Z} \subseteq \mathbb{Q}$. So $\mathbb{Z}^2 \preceq \mathbb{Z} \preceq \mathbb{Q}$ and we can apply Lemma 3.3 to conclude $\mathbb{Z}^2 \preceq \mathbb{Q}$.

The fact that $\mathbb{Q} \preceq \mathbb{N}$ says that $\mathbb{Q}$ is countable

3.4 Example (The set of non-negative rational numbers is countable)**.** Let $\mathbb{Q}^+$ stand for the set of all non-negative rational numbers. Note that each $r \in \mathbb{Q}^+$ can be written as as fraction $\frac{p}{q}$ where p and q are natural numbers. Of course a single r can be written as a fraction in many ways: $\frac{1}{2}$ and $\frac{2}{4}$ and $\frac{17}{34}$ all denote the same rational number, for example, but this will not matter to us.

Claim. $\mathbb{Q}^+$ is countable.

Proof. To show that the set $\mathbb{Q}$ of rational numbers is countable, *i.e.*, to show that $\mathbb{Q}^+ \preceq \mathbb{N}$, it suffices to construct an surjective function $h : \mathbb{N} \to \mathbb{Q}^+$. Define h by

$$h(n) = \begin{cases} \frac{p}{q} & \text{if } n \text{ is of the form } 2^p 3^q \\ 0 & \text{otherwise} \end{cases}$$

Note that the definition of h makes sense because every n has a unique prime factorization, so that it is well-defined whether n is of the form $2^p 3^q$ or not, and if it is, then p and q are uniquely determined. Thus h really is a function. And it is surjective by virtue of the fact that *every* non-negative rational can be written as $\frac{p}{q}$ for some p and q. ///

The fact that a given r can be written as a fraction in many ways means that the function h is not injective (which is fine).

3.4 Uncountable sets

It is time for some examples of uncountable sets.

Let $\mathcal{B} = \{0,1\}^{\mathbb{N}}$ denote the set of all functions from $\mathbb{N}$ to $\{0,1\}$. Such a function is conventionally called an "infinite bitstring."

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

Proof. The proof technique is called "diagonalization."

We prove that any function $g : \mathbb{N} \to \{0,1\}^{\mathbb{N}}$ must fail to be surjective. Let an arbitrary $g : \mathbb{N} \to \{0,1\}^{\mathbb{N}}$ be given. Note that for any $i \in \mathbb{N}$, $g(i)$ is an infinite string. Claim: the following infinite string β is not in the range 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

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

To see that β is not in the range of g, we show that for each n, β cannot be $g(n)$. But it is easy to check that β differs from $g(n)$ in the nth place: β was built expressly to satisfy this condition. ///

3.6 Check Your Reading. *Prove that the* $\{0,1\}^*$ *of all* finite *bitstrings is countable.*

Now notice that an infinite bitstring is nothing more than a characteristic function whose domain is $\mathbb{N}$, that is, characteristic function for a subset of $\mathbb{N}$. So what we proved just then was the following.

3.7 Corollary. *The set* $Pow(\mathbb{N})$ *of all subsets of the natural numbers is uncountable.*

So what we have so far is a hierarchy, with finite sets having smaller cardinality than $\mathbb{N}$, and then $\mathbb{N}$ having 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, $Pow(A)$.

3.8 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 range of g. Here is the definition of one such set, call it C (in honor of Cantor):

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

To justify that this works, we must argue that for any $a \in A$, the set $g(a)$ is not C. So let an arbitrary a be given, we want to show that $g(a) \neq C$. Well, either we have $a \in g(a)$ or not:

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

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

It is absolutely essential to note that the above proof works uniformly for any set A whatsoever, in particular whether A is finite or infinite. For example, Cantor's theorem is the tool to prove that for any set A at all, there can be no surjective function from A to the set of subsets of A.

3.9 Check Your Reading. *Make a specific (small) finite set A. Write down A and $Pow(A)$ on a sheet of paper. 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 range of g.*

3.10 Check Your Reading (Very Highly Recommended). *Look at Cantor's Theorem in the special case where A is $\mathbb{N}$. Make sure you see that the proof of Cantor's Theorem is* the same proof idea *as the proof of Theorem 3.5 (keep in mind the association between characteristic functions and subsets).*

As a immediate consequence of Cantor's Theorem we have

3.11 Corollary. *If A is infinite then $Pow(A)$ is uncountable.*

Proof. Cantor's Theorem says that $A \prec Pow(A)$. If $Pow(A)$ were countable then we would have $Pow(A) \preceq \mathbb{N}$. Thus $A \prec \mathbb{N}$, which is a contradiction since we assumed A infinite. ///

3.5 Tools for Showing Countability

Suppose you'd like to show a certain set A to be countable. Here is a menu of useful facts, collected as Lemmas.

3.12 Lemma. *A subset of a countable set is countable.*

Proof. Suppose $A \subseteq B$ and suppose B is countable. Then there is an injective $f : B \to \mathbb{N}$. The inclusion function $i : A \to B$ is injective. The composition of injective functions is injective, so $f \circ i : A \to \mathbb{N}$ shows A to be countable. ///

3.13 Lemma. *If $A_1, A_2, A_3, \ldots A_k$ is a* finite *family of countable sets then the Cartesian product $A_1 \times A_2 \times A_3 \times \cdots \times A_k$ is countable.*

Proof. To show $A_1 \times A_2 \times A_3 \times \cdots \times A_k$ countable we use the result of part 7 of Section 3.3. That is, we know that $\mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \cdots \times \mathbb{N}$ is countable, so all we have to do is find an injection g from $A_1 \times A_2 \times A_3 \times \cdots \times A_k$ 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 $f : (A_1 \times A_2 \times A_3 \times \cdots \times A_k) \to (\mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \cdots \times \mathbb{N})$ 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. ///

3.14 Lemma.

1. *If $A_1, A_2, A_3, \ldots$ is a family of countable sets indexed by the natural numbers then the union $\bigcup\{A_1 \cup A_2 \cup A_3 \cup \ldots\}$ is countable.*
2. *If $A_1, A_2, A_3, \ldots A_k$ is a finite family of countable sets then the union $\{A_1 \cup A_2 \cup A_3 \cup \cdots \cup A_k\}$ 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 *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 is not 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 $h_i : A_i \to \mathbb{N}$, namely $h_i(a) = p_i^{f_i(a)+1}$. 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. (Do you see why we set $h_i(a) = p_i^{f_i(a)+1}$ rather than simply $h_i(a) = p_i^{f_i(a)}$?)

So now we can define $h : \bigcup\{A_1 \cup A_2 \cup A_3 \cup \dots\} \to \mathbb{N}$ by $h(a) = h_i(a)$ where i is least such that $a \in A_i$. This is an injection, so $\bigcup\{A_1 \cup A_2 \cup A_3 \cup \dots\}$ is countable.

The second part is conceptually just a special case of part 1. Strictly speaking it isn't a special case but the very same proof works, we just don't need to use *all* the prime numbers in the argument.

///

3.15 Check Your Reading. *Suppose $A \subseteq B$ and A is uncountable. Explain why B is uncountable.*

3.6 Cardinality and Formal Languages

In this section we apply the ideas of cardinality (*cf.* Section 3) in the setting of formal languages.

Recall the construction we did in Section 2.2. For any finite alphabet Σ we put Σ^* into a natural bijective correspondence with $\mathbb{N}$. This proves:

3.16 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 3.5.

This leaves one question unexplored: what about the set of finite strings over a 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 Exercise 55.

Summarizing, we have that

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

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

3.17 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

$$x_0, x_1, x_2, \dots$$

This is a consequence of Theorem 3.16: 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 range 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 range 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 3.17 and the proof of Cantor's Theorem (Theorem 3.8) are really the same proof, just applied in a slightly different setting. Just keep in mind the association between characteristic functions and subsets.

3.7 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. You can do math under the assumption that CH is true or you can 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 CH, while $\neg P$ is proven under the assumption that CH is false. So you can choose whether to believe P or to believe $\neg P$!

3.8 Summary

What you need to know to solve most problems are the following two techniques.

- *To prove a set countable:* To prove that a set X is *countable*, it suffices to do one of the following things
  - Define some *injective* function $f : X \to C$, where C is some set that you have already shown to be countable, or

  - Define some *surjective* function $f : C \to X$, where C is some set that you have already shown to be countable

  If you can take the set C to be **N**, the natural numbers, you're always good, since **N** is countable *by definition.*

- *To prove a set uncountable:* To prove that a set Y is *uncountable*, it suffices to do one of the following things

  - Define some *injective* function $f : U \to Y$, where U is some set that you have already shown to be uncountable, or
  - Define some *surjective* function $f : Y \to U$, where U is some set that you have already shown to be uncountable

  There is not a canonical, "by definition" uncountable set, but two standard choices for U are (i) the set $Pow(\mathbb{N})$ of subsets of the natural numbers, and (ii) $\mathbb{R}$, the set of real numbers. These sets are well-known to be uncountable.

3.18 Check Your Reading. *[This is just a "replay" of the summary advice given above but it is repeated here to stress how important it is as a tool.]*

Fill in each of the blanks below with one of the words countable, uncountable, injective*, or* surjective.

- *To prove that a set X is* countable*, it is sufficient to define some _____ function $f : S \to X$, where S is some set already known to be _____.*
- *To prove that a set Y is* uncountable*, it is sufficient to define some _____ function $f : S \to Y$, where S is some set already known to be _____.*
- *To prove that a set X is* countable*, it is sufficient to define some _____ function $f : X \to S$, where S is some set already known to be _____.*
- *To prove that a set Y is* uncountable*, it is sufficient to define some _____ function $f : Y \to S$, where S is some set already known to be _____.*

3.9 Exercises

***Exercise* 47.** Prove that if $A \prec B$ and $B \prec C$ then $A \prec C$.

Hint. This isn't trivial: you can use Lemma 3.3 to make some progress but note that part of what you need to establish is that $C \preceq A$ fails!

***Exercise* 48.** What does Cantor's Theorem say when $A = \emptyset$? (What is $Pow(\emptyset)$ anyway?)

***Exercise* 49.** Prove that the set $\mathbb{Q}$ of *all* rational numbers is countable.

***Exercise* 50.** Let $(0,1)$ be the open interval of real numbers between 0 and 1 (not inclusive). Let $(1,\infty)$ be the open interval of all real numbers greater than 1. Prove that $(0,1) \approx (1,\infty)$. Hint: there is a familiar mathematical function that does the trick.

***Exercise* 51.** Show that $(0,\infty) \approx \mathbb{R}$. Hint: $(0,\infty) \preceq \mathbb{R}$ is easy (see a previous problem). To show $\mathbb{R} \preceq (0,1)$, there is a familiar mathematical function that works.

Show that $(0,1) \approx \mathbb{R}$.

***Exercise* 52.** Show that $Pow(\mathbb{N}) \approx [0,1]$. Here $[0,1]$ is the closed real interval from 0 to 1. *Hint.* Think characteristic functions for $Pow(\mathbb{N})$ and base-2 notation for $[0,1$.

***Exercise* 53.** Prove that the set of all *finite* subsets of $\mathbb{N}$ is countable.

Hint. One way to do this is to associate with each finite subset of $\mathbb{N}$ a finite string over (say) $\{0,1\}$. If you can do this in an injective way this suffices, since $\{0,1\}^*$ is countable.

Another way to do this is to (injectively) associate with each finite subset of $\mathbb{N}$ a natural number. The 2^x3^y trick we used for showing the countability of $\mathbb{N}^2$ could provide inspiration...

***Exercise* 54.** For each of the following sets, decide if they are countable or uncountable. Then prove your answer (perhaps using the advice in 3.18).

1. $Pow(\mathbf{N})$, the set of subsets of $\mathbf{N}$,
2. $\{0,1\}^*$, the set of all finite strings over the alphabet $\{0,1\}$.
3. $\mathbf{N}^*$, the set of all finite sequences drawn from $\mathbf{N} = \{0,1,2,\dots\}$.

***Exercise* 55.** . Let A be a countably infinite set. Prove that the set A^* of all finite strings over A is countable.

Hint. Use Exercise 3. Don't reinvent the wheel.

4. The set S of all bitstrings, whether finite or infinite:

$$S = \{\sigma \mid \sigma \text{ is a finite bitstring or an infinite bitstring}\}$$

5. The set E of all strings in $\{0,1\}^*$ that have even length.

6. The set Y of all functions from $\mathbf{N}$ to $\mathbf{N}$. That is, $Y = \{f \mid f : \mathbf{N} \to \mathbf{N}\}$

***Exercise* 56.** 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 $\mathbf{e}$ denotes!

4 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.

4.1 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.

4.2 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 4.3.

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.

4.3 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:

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

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

4.6 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.

4.1 König's Lemma

Before reading this section you should do Exercise 61 at the end of this section.

When trees are known to be finite-branching, life is very different! Each of the statements in Exercise 61 is *true* under the hypothesis of finite-branching. It all comes down the following famous and important theorem:

4.7 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.

Do Exercise 62 now!

4.2 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.

4.8 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

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

4.9 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.

4.3 Exercises

***Exercise* 57.** The set of all bit strings, *i.e.*, the set of all finite sequences of 0s and 1s is a tree. Draw it.

***Exercise* 58.** 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.

***Exercise* 59.** 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.

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

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

***Exercise* 60.** Since the issue in Exercise 59 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.

***Exercise* 61.** Here are some exercises 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.

1. 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.*

2. 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 $\leq d$.*

3. 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 $\leq d$.*

4. 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.*

5. 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.*

6. 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.*

7. 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 $\leq n$.*

8. 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.*

9. 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.*

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

***Exercise* 62.** Prove, using König's Lemma, each of the statements in Exercise 61 under the hypothesis that tree T is finite-branching.

***Exercise* 63.** 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 4.9.

5 Induction: Defining, Computing, 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 exercises.

5.1 Defining

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

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

12. 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 $n \geq 0$

13. 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 $k < n$?

14. 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 $t(n)$.

15. 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 $b(3) = 2$."

5.2 Proving

16. Prove that $1+2+3+\cdots+n = \frac{1}{2}n(n+1)$
17. Prove that $2+4+6+\ldots 2n = n(n+1)$
18. Prove that for all $n \geq 4$, $\ 2^n < n!$.
19. Prove that $fib(0)+fib(1)+\cdots+fib(n) = fib(n+2)-1$
20. Refer to part 14 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.

21. 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.

22. By the way, do you see that in exercise 21 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).

23. 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 $n^2 - n + 41$. 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!

24. 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.

25. 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?

26. An example of needing a "strong" induction hypothesis.

    (a) 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.

(b) 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.*

27. 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.*

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

Part II

Regular Languages

6 Deterministic Finite Automata

6.1 Definition (Deterministic Finite Automaton)**.** *A **deterministic finite automaton** (DFA) is a 5-tuple $\langle \Sigma, Q, \delta, S, F\rangle$, where*

- Σ *is a finite set, called the* input alphabet
- Q *is a finite set, called the set of* states
- $s \in Q$ *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.

6.2 Definition. *Let $M = \langle \Sigma, Q, \delta, s, F\rangle$ be a DFA, and let $x = a_0 a_2 \dots, a_{n-1}$ be a string.*

*A **run** of M on x is a sequence of $n+1$ states $\langle q_0, \dots, q_n\rangle$ such that*

- *$q_0 = s$, the start state of M;*
- *for each $0 \le i \le n-1$, $\quad q_i \xrightarrow{a_i} q_{i+1}$ is a transition in δ*

Note that the concatenation of the symbols in the labelled transitions yields x

It is suggestive to write

$$q_0 \xrightarrow{a_0} q_1 \xrightarrow{a_1} \dots \xrightarrow{a_{n-1}} q_n \qquad or \qquad q_0 \overset{x}{\twoheadrightarrow} q_n$$

*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 $\langle s \rangle$.

6.3 Definition. *Let M be a DFA. The* ***language*** $L(M)$ accepted by *M is the set of strings accepted by M:*

$$L(M) \overset{def}{=} \{x \in \Sigma^* \mid \textit{ the run of } M \textit{ on } x \textit{ is accepting.}\}$$

Some authors say: "the language *accepted by M*" to mean the same thing as "the language accepted by *M*"

We will shortly consider a nondeterministic variation on *DFAs*, in which there can be more than one run, or no runs, on a given string. By phrasing the notion of acceptance the way we have, we will not need to change the formal definition when we consider the more general case.

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

$$q \xrightarrow{a_1} q_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} q_n \qquad \text{or} \qquad q \overset{x}{\twoheadrightarrow} q_n$$

in such a case. However we will only use the term "run" for a sequence that begins at the start state.

The $\hat{\hat{\delta}}$ function

Following up on the last remark, it is sometimes convenient to have a notation for the state that a *DFA* ends in, starting at some state q and processing string x.

6.4 Definition (The $\hat{\hat{\delta}}$ function)**.** *Let* $M = \langle \Sigma, Q, \delta, S, F \rangle$ *be a DFA. The function* $\hat{\hat{\delta}} : Q \times \Sigma^* \to Q$ *is defined by recursion:*

$$\begin{aligned} \hat{\hat{\delta}}(q, \lambda) &= q \\ \hat{\hat{\delta}}(q, xa) &= \delta(\hat{\hat{\delta}}(q, x), a) \qquad \textit{for } x \in \Sigma^*,\ a \in \Sigma \end{aligned}$$

If you read other books on automata, be aware that they may use the $\hat{\hat{\delta}}$ notation exclusively.[4] At any rate, these two notations mean **exactly the same thing:**

$$q \overset{x}{\twoheadrightarrow} q_n \qquad \text{and } \hat{\hat{\delta}}(q, x) = q_n$$

[4]The truth is that the $\hat{\hat{\delta}}$ notation is more convenient for *DFAs* but the "runs" notation is more convenient for *NFAs*, coming up shortly ...

6.1 Regular Languages

6.5 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 6.2. In Section 14 we will be able to give a complete answer to the question, “which languages are regular?”

6.6 Example. The set $A \overset{\text{def}}{=} \{x \in \{a,b\}^* \mid \text{ the length } |x| \text{ of } x \text{ is odd}\}$ is regular.

Here is a *DFA* that accepts A.

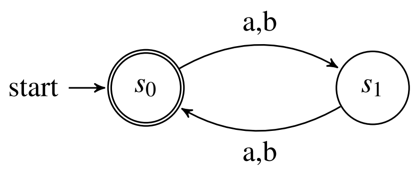
- Σ is (of course) $\{a,b\}$
- Q is $\{s_0, s_1\}$
- the start state s is s_0
- the set of accepting states is $\{s_1\}$
- the transition function δ is

$$\{((s_0,a),s_1), \quad ((s_0,b),s_1), \quad ((s_1,a),s_0), \quad ((s_1,b),s_0)\}$$

That’s a verbose way to write a *DFA*. Here is a more compact notation, in which we display the δ function as a lookup table, and indicate the start and accepting states as part of the table.

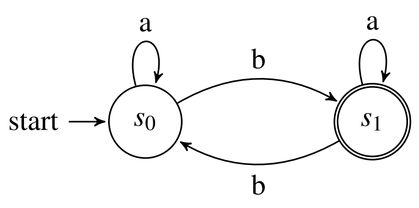
	a	b
start $\to s_0$	s_1	s_1
accepting s_1	s_0	s_0

But for small *DFAs*, we can capture the same information in a picture, like this:

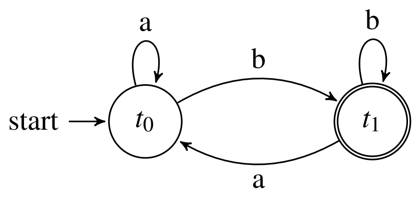


We have introduced a shorthand here: in pictures we can capture more than one arc between the same states by label a single arc with more than one symbol.

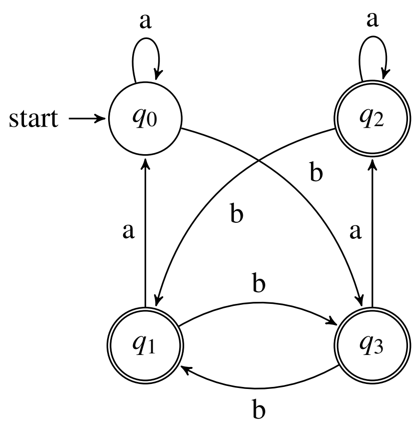
6.7 Example. The set $B \overset{\text{def}}{=} \{x \mid x \text{ has an odd number of } b\text{s}\}$ is regular; here is a picture of a *DFA* that accepts B.



6.8 Example. The set $C \overset{\text{def}}{=} \{x \mid x \text{ ends with a } b\}$ is regular; here is a *DFA* that accepts C.

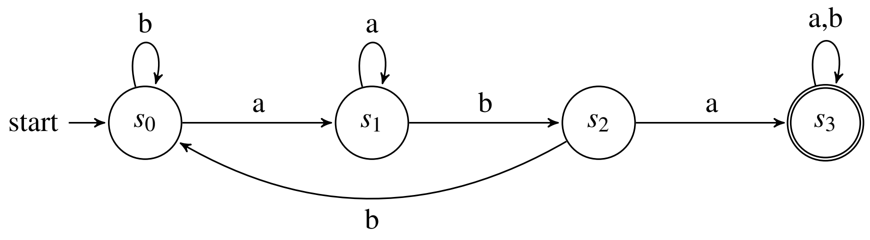


6.9 Example. The set $D \overset{\text{def}}{=} \{x \mid x \text{ has an odd number of } b\text{s or ends with a } b\}$ is regular; here is a *DFA* that accepts D.

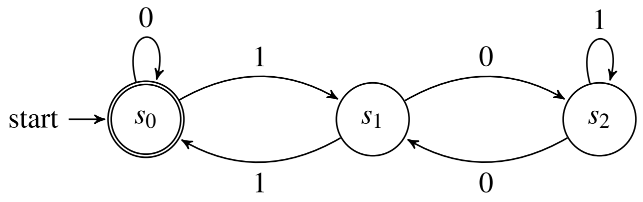


This is not too difficult an example, still you might wonder if there is a way to build *DFAs* that is somewhat systematic, relying less on inspiration. There are indeed some tricks, for example we will revisit this language in Example 7.5 later.

6.10 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.



6.11 Example. This example shows that *DFAs* can do counting in modular arithmetic. The set $F \overset{\text{def}}{=} \{x \mid$ the number that x 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.

To *justify* that this works, we establish a *stronger claim.* Namely

> for every bitstring x, if x codes a number that is equal to i mod 3 then the run of M on x will end in state q_i.

Now, if we prove this then we certainly will have proved our main assertion, since the main assertion is a special case of the claim. But the neat thing is that the second, stronger, claim above is *easier to prove* than the simpler, first claim.

6.2 Not All Languages are Regular

Not every language is 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.

6.12 Example. Fix 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 6.11 that *DFAs* can count in *modular arithmetic*...

6.3 A Peek Ahead: Regular Expressions and *DFAs*

We have already defined *regular expressions*. It would be pretty bad choice of terminology if the languages defined by "regular expressions" were not the same class as the "regular" languages, defined as as those defined by *DFAs*. In fact they are the same. But we have to do some work to prove this, so for now we will suffer with the anticipation.

Finite automata are concerned with *accepting* strings algorithmically. Regular expressions *describe* languages explicitly. In the forthcoming sections we compare finite automata to regular expressions. The main result is: regular expressions and finite automata have the same expressive power. Specifically we will show that

- Given any regular expression E there is a finite automaton M such that $L(M) = L(E)$. Furthermore there is an algorithm to build M given E.
- Given any finite automaton M there is a regular expression E such that $L(E) = L(M)$. Furthermore there is an algorithm to build E given M.

6.4 Exercises

***Exercise* 64.** For each of the following languages, make an *DFA* recognizing it.

1. Over the alphabet $\Sigma = \{a,b\}$: the set of all strings whose next-to-last symbol is a b.
2. Over the alphabet $\Sigma = \{a,b,c\}$: $\{a^n b^m c^p \mid n,m,p \geq 0\}\}$
3. Over the alphabet $\Sigma = \{a,b\}$: $\{w \mid w$ contains bb as a substring$\}$.
4. Over the alphabet $\Sigma = \{a,b\}$: $\{w \mid w$ does not contain bb as a substring$\}$.
5. Over the alphabet $\Sigma = \{a,b\}$: $\{w \mid w$ every odd position of w is the symbol b$\}$.
6. Over the alphabet $\Sigma = \{a,b\}$: $\{w \mid$ the third symbol from the end of w is an a$\}$.
7. Over the alphabet $\Sigma = \{a,b\}$: $\emptyset$, the empty set.
8. Over the alphabet $\Sigma = \{a,b\}$: $\{\lambda\}$. This is the language consisting of one string, the empty string λ.

***Exercise* 65.** In this exercise you will generalize Example 6.11.

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 $L(M) = L_{k,m}$.

It is simplest to agree that the empty string codes the number 0.

Note. You can't give a concrete *DFA* as an answer, of course. What this asks for is a general description of how to build such a *DFA* given k and m.

Hint. I suggest first solving the problem fixing $m = 2$ and letting k vary, then fixing $k = 2$ and letting m vary, then finally doing the general case.

***Exercise* 66.** *Induction practice I (from HMU)* 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.

***Exercise* 67.** *Induction practice II (from HMU)* 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 integer 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

$$\text{either } \{c\}^* \subseteq L(M) \quad \text{or} \quad \{c\}^* \cap L(M) = \emptyset$$

***Exercise* 68.** *Induction practice III* 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 $|y|$.

7 *DFA* Complement and Product

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.

7.1 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.

7.1 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:

7.2 Theorem. *When M' is constructed from M as in Definition 7.1, $L(M') = \overline{L(M)}$.*

Proof. It is obvious that for any $x \in \Sigma^*$ and any states p and q, $p \xrightarrow{x}\!\!\!\!\rightarrow q$ in M if and only if $p \xrightarrow{x}\!\!\!\!\rightarrow q$ 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.

7.3 Theorem. *If A is regular then $\overline{A}$ is regular.*

Proof. Directly from Theorem 7.2. ///

7.2 The Product Construction

The product construction is a central idea, that gives us both intersection and union for regular languages.

7.4 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.*

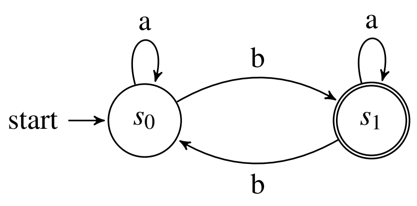
$$P = (\Sigma, (Q_1 \times Q_2),\ \delta_P,\ (s_1, s_2), (F_1 \times F_2)),\ \textit{where } \delta_P \textit{ is given by}$$

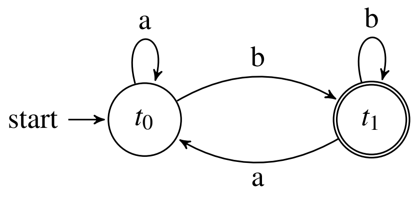
for each $c \in \Sigma$, and states p_1, p_2, q_1, q_2,

$$\begin{aligned} (p_1, p_2) \xrightarrow{c} (q_1, q_2) &\quad \textit{precisely when} \\ p_1 \xrightarrow{c} q_1 &\quad \textit{in } \delta_1 \textit{ and} \\ p_2 \xrightarrow{c} q_2 &\quad \textit{in } \delta_2. \end{aligned}$$

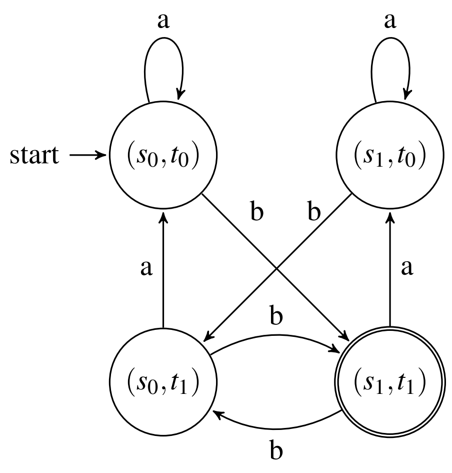
This construction builds a *DFA* that accepts the *intersection* of the languages accepted by the original *DFAs*. Before proving that, let's see an example.

7.5 Example. Recall Example 6.9 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 6.9, except that here the states have funny names!

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.[5]

Once you understand this construction it should be intuitively clear that the following proposition is true. But we give a careful proof of correctness, as a model for how such proofs are done.

7.6 Theorem. *When P is constructed as in Definition 7.4,* $L(P) = L(M_1) \cap L(M_2)$

Proof. We prove the following claim: for any $x \in \Sigma^*$ and states p_1, p_2, q_1, q_2,

$$(p_1, p_2) \xrightarrow{x} (q_1, q_2) \quad \text{if and only if}$$
$$p_1 \xrightarrow{x} q_1 \text{ and}$$
$$p_2 \xrightarrow{x} q_2.$$

Notice that this looks quite a lot like the *definition* of δ we gave for *P*. The

[5]Some authors even use the notation $M_1 \times M_2$ for our *P* above, but this is really asking for trouble.

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 Proposition, 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

$$L(M') = L(M_1) \cup L(M_2)$$

7.7 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 7.6. You only have to change a word or two!*

7.3 Regular Closure: Intersection and Union

These two constructions prove the following theorem.

7.8 Theorem. *If languages A and B are regular then so are $A \cap B$ and $A \cup B$.*

Proof. Immediate from Theorem 7.6 and the observation in 7.7. ///

7.3.1 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!

7.4 Exercises

***Exercise* 69.** Do several examples of the product construction, on small *DFAs* you are familiar with. Convince yourself that you did the construction correctly, by testing lots of input strings.

***Exercise* 70.** 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 $L(M) \cap \overline{L(N)}$.

***Exercise* 71.** 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.

***Exercise* 72.** 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. Not that $L(M) \cap L(N) = \emptyset$. Make the product construction on M and N, obtaining a *DFA* P whose language is $L(M) \cap M(N)$. Explain what is going on. (This isn't hard.)

8 Nondeterministic Finite Automata

Here is a generalization of *DFAs*.

8.1 Definition (Nondeterministic Finite Automaton)**.** *A **nondeterministic finite automaton** (NFA) is a 5-tuple $\langle \Sigma, Q, \delta, s, F \rangle$, where*

- Σ *is a finite set, called the* input alphabet
- Q *is a finite set, called the set of* states
- $s \in Q$ *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*.

8.1 Runs of an *NFA*

The definitions of *run* of an *NFA* is exactly as written for a *DFA*. We still say that an accepting run of M on x is a run whose last state is an accepting state. 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, as the machine "tries" to construct a run on x, it finds itself in a state q while reading a symbol a and there is simply not 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 is is possible that there are *no* runs of N on x: this will happen if every attempt to make a run blocks at some point. (Of course it follows that there are no accepting runs on x.)

So we have to be a bit more subtle in our notion of acceptance for an *NFA*.

With this in mind we make the following analogue of Definition 6.3.

8.2 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:*

$$L(M) \overset{def}{=} \{x \in \Sigma^* \mid \textit{there exists an accepting run of } M \textit{ on } x\}$$

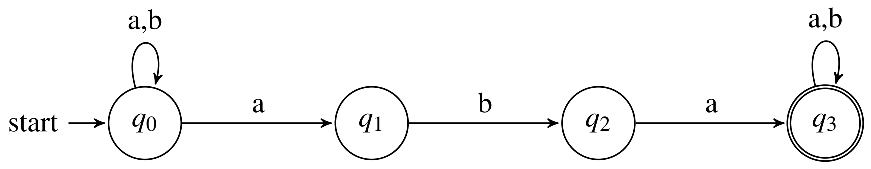
Later in this section we will prove the following fact:

> 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.

8.2 Examples

8.3 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!

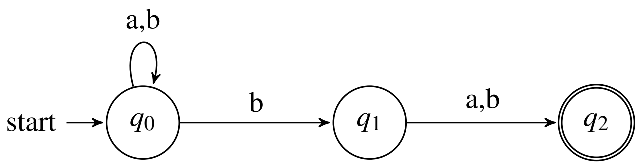
Another run is:

$$q_0 \xrightarrow{a} q_0 \xrightarrow{a} q_1 \xrightarrow{b} q_2 \xrightarrow{a} q_3$$

That's an accepting run. Since there *exists* an accepting run, we say that *aaba* is accepted by N, and so $aaba \in L(N)$.

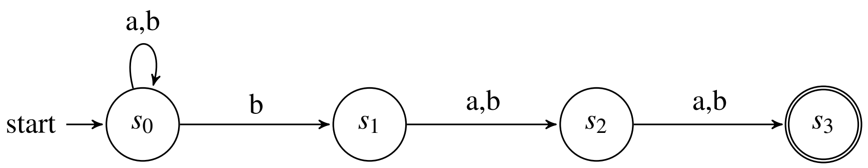
Earlier we saw a *DFA* that accepts the same language. But the *NFA* captures more directly what the specification of the language is.

8.4 Example. This *NFA* N accepts $\{x \in \{a,b\}^* \mid$ the next-to-last symbol in x is b.$\}$



It's easy enough to make a *DFA* for this language. But let's generalize this language:

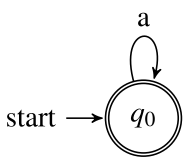
8.5 Example (The k^{th}-to-last symbol is...)**.** Let Σ be the language $\{a,b\}$ and let L be the set of strings whose 3rd-to-last symbol is an a. For example $bbaba \in L$, while $bbbbab \notin L_3$. It is easy to make an *NFA* recognizing L_3:



It can be shown that the smallest *DFA* for this language 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 Exercise 120, later.

Thus *NFAs* can be exponentially more succinct than *DFAs*.

8.6 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.

8.7 Check Your Reading. *Why didn't we make the remark about the ambiguity of that alphabet back when we drew pictures of DFAs?*

8.8 Example (Running Machines in Reverse)**.** If A is regular then so is A^R, the set of reversals of strings in A. A natural intuition about how to prove this would be to start with a *DFA* M such that $L(M) = a$, and then construct a new *DFA* M' which is like M but runs backwards, that is, all the transition arrows are reversed. But that doesn't work! Several of the requirements for being a *DFA* are violated. On the other hand this idea works just fine for *NFAs*. See Exercise 79.

8.3 From NFAs to DFAs: the Subset Construction

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, and behave better with respect to complement, 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* M recognizing the same language as N.

Let $Pow(Q)$ denote the set of subsets of Q. The key idea is that a state in M is a set of states from N. The details are in Algorithm 1

```
**Algorithm 1:** NFA to DFA
Input: a NFA $N = (\Sigma, Q, \delta, s, F)$
Output: a NFA $M$ such that $L(M) = L(N)$
The states of $M$ will be elements of $Pow(Q)$;
The start state of our $M$ 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 $\delta_M$ by

  $P \xrightarrow{c} R$ if there is some state $p \in P$ and some state $r \in R$ with $p \xrightarrow{c} r$ in $N$.

return $M = (\Sigma, Pow(Q), \delta', \{S\}, \mathcal{F})$
```

8.9 Theorem. *Let N be an NFA. The DFA M constructed in Algorithm 1 satisfies $L(M) = L(N)$.*

Proof. During this proof we will use capital letters to name states of the *DFA M* (as we did in the algorithm) to help remind you that such a state in M 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 $\twoheadrightarrow$ 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.

$$\textit{Claim:} \quad P \xrightarrow{x}\!\!\!\!\twoheadrightarrow_M R \quad \text{iff} \quad R = \{r \mid \exists p \in P,\ p \xrightarrow{x}\!\!\!\!\twoheadrightarrow_M r\}$$

Note that this looks just like the *definition* of $\longrightarrow$ in M, except that it is about the reflexive-transitive closure instead. 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 $\twoheadrightarrow$ is determined, and the claim is either true or false. Before we prove that it is true, let's note that it is all we need to show the correctness of our construction. Assuming the claim, we have that for any string x,

$$\{s\} \xrightarrow{x}\!\!\!\!\twoheadrightarrow \{r \mid \exists s \in \{s\}, s \xrightarrow{x}\!\!\!\!\twoheadrightarrow r \text{ in } N\} \qquad \text{in other words,}$$
$$\{s\} \xrightarrow{x}\!\!\!\!\twoheadrightarrow \{r \mid s \xrightarrow{x}\!\!\!\!\twoheadrightarrow r \text{ in } N\}$$

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 M accepts x.

It remains to prove the claim. We prove this claim by induction on the length of x.

When $x = \lambda$ each side of the iff holds because then we have $P = R$.

For the inductive step, suppose that $x = cy$ for $c \in \Sigma$ and $y \in \Sigma^*$. We want to show that

$$P \overset{cy}{\twoheadrightarrow} \{r \mid \exists p \in P,\ p \overset{cy}{\twoheadrightarrow} r \text{ in } N\}$$

Using the definition of $\longrightarrow$ we have

$$P \xrightarrow{c} \{p_1 \mid \exists p \in P,\ p \xrightarrow{c} p_1 \text{ in } N\} \quad \ldots\text{call this set } P_1$$

Using the induction hypothesis on y we have

$$P_1 \overset{y}{\twoheadrightarrow} \{r \mid \exists p_1 \in P_1,\ p_1 \overset{y}{\twoheadrightarrow} r \text{ in } N\}$$

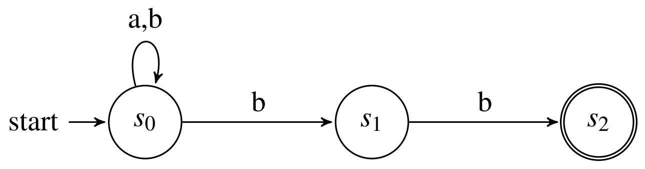
Putting these together we have

$$\begin{aligned} P \xrightarrow{cy} & \{r \mid \exists p \in P\, \exists p_1 \in P_1,\ p \xrightarrow{c} p_1 \overset{y}{\twoheadrightarrow} r \text{ in } N\} \\ = \quad & \{r \mid \exists p \in P\ p \overset{cy}{\twoheadrightarrow} r \text{ in } N\} \end{aligned}$$

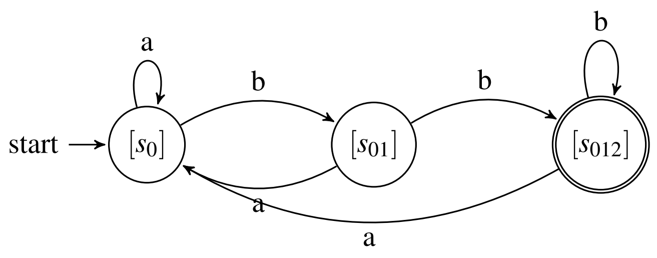
which is what we wanted.

///

8.10 Example. Here is an *NFA* accepting strings that end in *bb*.

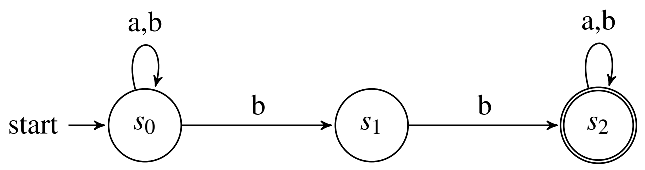


After the subset construction, we get

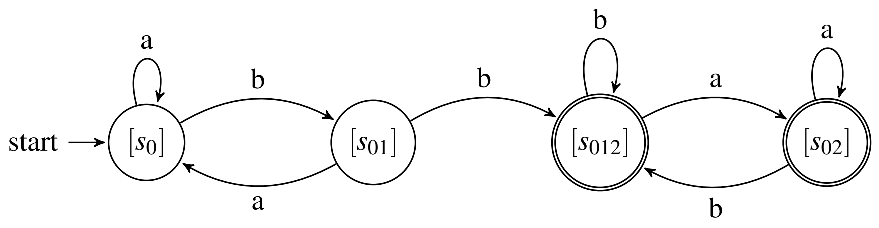


This is the *DFA* you would probably write directly if given this specification.

8.11 Example. Here is a slight variation on Example 8.10. Here we accept strings that *contain bb*.



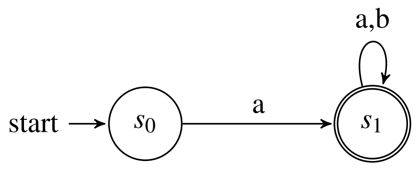
After the subset construction, we get



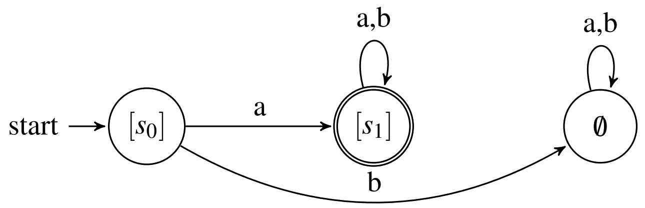
This is probably *not* the *DFA* you would probably write directly if given this specification. If you think about it you will see that this is not the smallest *DFA* we could write for this language. We'll come back to this point, in Section 13.

8.12 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 "fix" such machines 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



8.4 The Product Construction on *NFAs*

In Section 7.2 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 7.6 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*.

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*. See Exercise 78.

8.5 The Complement Construction on *NFAs*?

The complement construction 7.1 on *NFAs* fails, in the sense that swapping accepting and non-accpepting states in an *NFA* does *not* have the effect of complementing the accepted language. See Exercise 77.

8.6 Exercises

***Exercise* 73.** For each of the languages in Exercise 64, make an *NFA* recognizing it. Yes, you already built *DFAs*, but here you are invited to see if it is easier to build *NFAs* for them than *DFAs*. Convert your answer to a *DFA*, and compare to your answers you gave in Exercise 64.

***Exercise* 74.** Let K be any language containing a single string. Show that K is regular.

***Exercise* 75.** Let K be a finite language, that is, K is a language containing finitely many strings. Show that K is regular.

***Exercise* 76.** A language K is said to be *co-finite* if $\overline{K}$ is finite, that is, if only finitely many strings fail to be in K.

Prove that any co-finite language is regular.

***Exercise* 77.** *Complementation and automata* **An important problem!**

1. 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 $\Sigma^* - L$).

2. Show that if N is an *NFA* accepting the language L, then there is an *NFA* accepting the language $\overline{L}$.

3. Explain why part 2 does *not* contradict part 3.

***Exercise* 78.** 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 7.4, 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 ...).

***Exercise* 79.** *Reverse*

If L is a language let us write L^R for the language consisting of the reverses of the strings in L. Show that if L is regular so is L^R.

Hint: The basic idea was already given in the text. Start your careful proof like this:

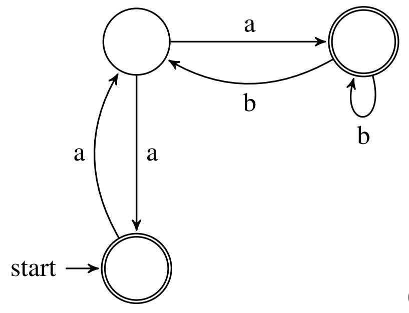
> Let $M = (\Sigma, Q, \delta, S, F)$ be an *NFA* such that $L(M) = L$. We define the automaton M' as follows ...

A formal proof that the machine M' really does accept L^R is not asked for here. What *is* being asked for is a precise definition of M'.

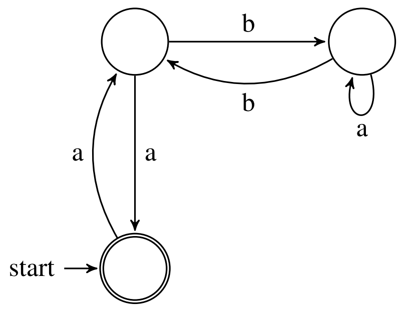
***Exercise* 80.** *Regular expressions and automata I*

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 exercise and the next are less ambitious, they ask you to explore the correspondence intuitively, to build your intuitions.

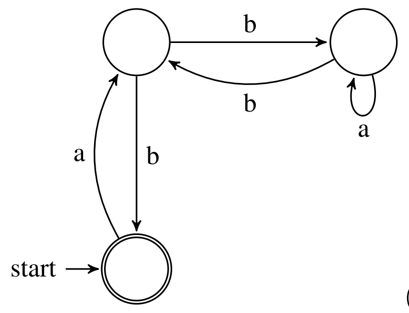
Match each *NFA* with an equivalent regular expression. (From [Koz97].)



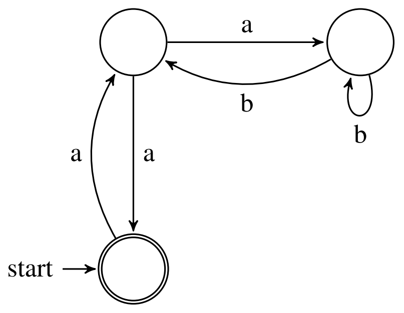
($\mathbf{M_1}$)



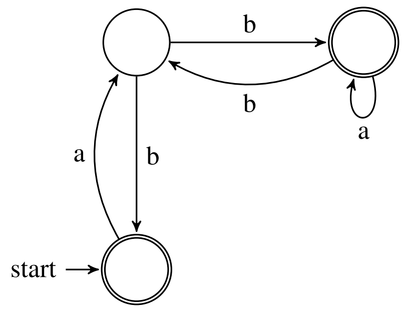
($\mathbf{M_2}$)



($\mathbf{M_3}$)



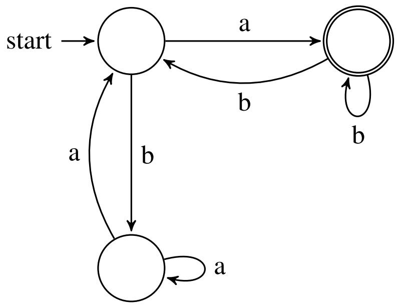
($\mathbf{M_4}$)



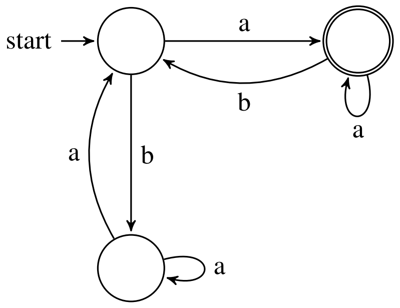
($\mathbf{M_5}$)

- $\lambda + a(ab^*b + aa)^*ab^*$
- $\lambda + a(ba^*b + ba)^*ba^*$
- $\lambda + a(ba^*b + aa)^*a$
- $\lambda + a(ab^*b + aa)^*a$
- $\lambda + a(ba^*b + ba)^*b$

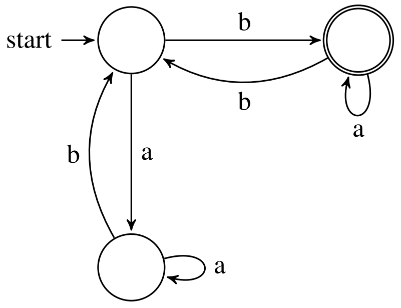
Exercise* 81** (Kozen).*** *Regular expressions and automata II* Match each *NFA* with an equivalent regular expression (From [Koz97].)



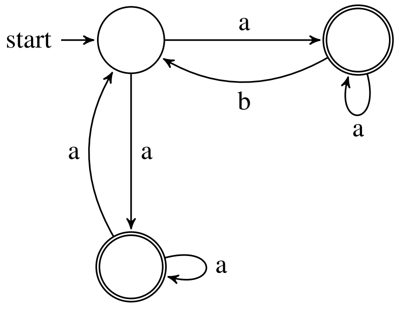
($\mathbf{N_1}$)



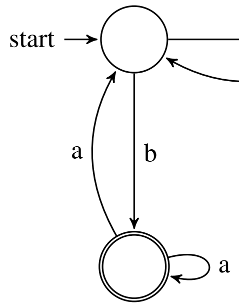
($\mathbf{N_2}$)



($\mathbf{N_3}$)



($\mathbf{N_4}$)



($\mathbf{N_5}$)

1. $(aa^*b+ba^*b)^*ba^*$
2. $(aa^*a+aa^*b)^*aa^*$
3. $(ba^*a+ab^*b)^*ab^*$
4. $(ba^*a+aa^*b)^*aa^*$
5. $(ba^*a+ba^*b)^*ba^*$

***Exercise* 82.** *NFA simulation* 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.

***Exercise* 83.** *"For-all" NFAs*

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.

***Exercise* 84.** *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

$$H(x,L) \overset{\text{def}}{=} \min\{H(x,y) \mid y \in L\}$$

Now for any language L over $\{0,1\}$ and any $k \geq 0$ we may define

$$N_k(L) \overset{\text{def}}{=} \{x \mid H(x,L) \leq k\}$$

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!

***Exercise* 85.** Let $\Sigma = \{0,1\}$, and let $\mathcal{Q} = \{q_0,q_1\}$.

1. How many *DFAs* are there that have input alphabet Σ and state set $\mathcal{Q}$?
2. (Hard) Some of the *DFAs* that you just wrote accept the same language. Give a few examples.

Now do the two things above but with arbitrary *NFAs* rather than *DFAs*.

***Exercise* 86.** *Prefixes*

A string x is a *proper prefix* of a string z if there is a non-empty string y such that $xy = z$. Suppose L is a language; we may define the following languages based on L:

- NoPrefix(L) = $\{w \in L \mid$ no proper prefix of w is in $L\}$.

- NoExtend(L) = $\{w \in L \mid w$ is not a proper prefix of any string of $L\}$.

Suppose L is regular. Is NoPrefix(L) guaranteed to be regular?

Suppose L is regular. Is NoExtend(L) guaranteed to be regular?

[This is a fairly hard problem.]

***Exercise* 87.** *Halves (from Sipser)*

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.]

9 *NFAs* with λ-Transitions

An *NFA*$_\lambda$ is an *NFA* that, in addition to transitioning from one state to another while reading an input character, has the capacity to transition from one state to another without consuming any input at all. We denote such transitions, naturally, as $p \rightarrow q$.

In this chapter we wil do two things.

1. Show how *NFA*$_\lambda$*s* do not have any more acceprting power than ordinary *NFAs*, and
2. Motivate *NFA*$_\lambda$*s* by showing how they add convenience in building machines to do certain jobs.

First the formal definition.

9.1 Definition (*NFA*$_\lambda$)**.** *A* nondeterministic finite automaton with λ transitions *(NFA$_\lambda$) is a 5-tuple $\langle Q, \Sigma, \delta, S, F \rangle$, where $Q, \Sigma, s,$ and F are as defined for NFA, but where δ allows, in addition to transitions $p \xrightarrow{c} q$, transitions $p \rightarrow q$.*

*A **run** of an NFA$_\lambda$ on $x = a_1 a_2 \dots a_n$ is a sequence of states $\langle s_0, \dots, s_k \rangle$ such that*

- *$s_o \in S$;*
- *for each $i \geq 0$, either for some a, $s_i \xrightarrow{a} s_{i+1}$ is a transition in δ or $s_i \rightarrow s_{i+1}$ 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 $s_n \in F$, 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 $p \rightarrow q$ as "λ" transitions. 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 (which is a common confusion). Sometimes we will use the phrase "silent transition."

9.1 From NFA_λ to NFA

Obviously any ordinary *NFA* can be considered an NFA_λ. Our goal is to prove that in fact our new *FA* model doesn't actually add any recognizing power. 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.

Once we have done our work, this will show up as a Theorem.

Note that we can't just *delete* λ transitions of course: we must compensate for removing them.

9.2 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.

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.

9.3 Lemma. *Suppose NFA_λ M has the transition $p \rightarrow q$.*

If M has a transition $q \rightarrow r$ and we build a new NFA_λ by adding *the transition $p \rightarrow r$, 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. ///

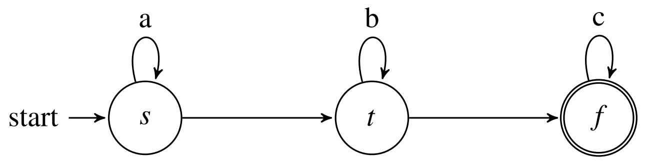
That lemma seems to take us in the wrong direction: it *adds* transitions rather than *removing* the ones we don't want. But the trick is that after all such transitions have been added, we can *then* just delete the bad transitions. Here is this idea expressed as an algorithm.

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'.

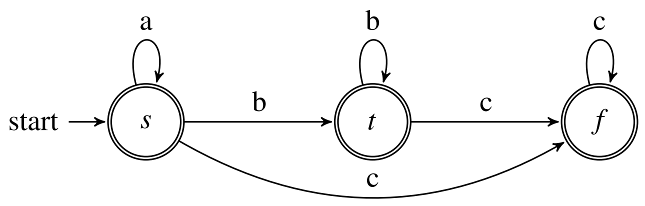
```
Algorithm 2: NFA_λ to NFA
Input: a NFA_λ $M = (\Sigma, Q, \delta, s, F)$
Output: a NFA $M'$ such that $L(M') = L(M)$
initialize: set $\delta^+$ to be $\delta$
repeat
    if $\delta^+$ has a transition $p \to q$ and a transition $q \to r$ then add $p \to r$ to $\delta^+$
    if $\delta^+$ has a transition $p \to q$ and a transition $q \xrightarrow{a} r$ then add $p \xrightarrow{a} r$ to
      $\delta^+$
until no change in $\delta^+$;
let $F'$ be $F \cup \{p \mid$ there is a λ transition $p \to f$ to a state $f$ in $F\}$ ;
let $M^+ = (\Sigma, Q, \delta^+, S, F')$;

define $\delta'$ to be $\delta^+$ with all λ transitions removed ;
return $M' = (\Sigma, Q, \delta', S, F')$
```

9.1.1 Example

9.4 Example. Let M be the *NFA*$_\lambda$



Eliminating λ transitions yields the *NFA*



9.5 Theorem. *Algorithm NFA_λ* to *NFA is correct:*

1. *the algorithm always terminates;*
2. *the output M' has no λ transitions, and*
3. *$L(M') = L(M)$.*

Proof.

1. Termination 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.

2. The fact that the output M' has no λ transitions is obvious from the algorithm.

3. 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 9.3.

   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 first λ transition is actually the last transition in the run:

$$s \xrightarrow{w}\!\!\!\!\rightarrow p \to f \quad \text{where } f \in F'$$

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

$$s \xrightarrow{w_1}\!\!\!\!\rightarrow p \quad \text{where } p \in F'$$

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 first λ transition has at least one transition:

$$s \xrightarrow{x}\!\!\!\!\rightarrow p \to q \xrightarrow{a} r \xrightarrow{y}\!\!\!\!\rightarrow f \quad \text{where } w = xay \text{ and } f \in F'$$

or

$$s \xrightarrow{x}\!\!\!\!\rightarrow p \to q \to r \xrightarrow{y}\!\!\!\!\rightarrow f \quad \text{where } w = xy \text{ and } f \in F'$$

In each of these cases the algorithm ensures that there is a transition $p \xrightarrow{a}\!\!\!\!\rightarrow r$, or $p \longrightarrow\!\!\!\!\rightarrow r$, as appropriate. So eliminating the transition from p to q yields a shorter accepting run, contradicting our assumption.

This completes the proof that *NFA* M' performs as advertised. ///

After seeing this algorithm and its correctness proof, we now have our result.

9.6 Theorem. *For every NFA$_\lambda$ 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.

Proof. This is a corollary of Theorem 9.5. ///

A variation on the construction?

In Algorithm 2 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 2 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 2 we *replaced* the line

> if δ^+ has a transition $p \to q$ 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. Exercise 90 asks you to find a counterexample.

9.2 *NFA*$_\lambda$ for Closure Properties

NFA$_\lambda$ are very convenient for building new automata from old ones, especially in light of the fact that the λ-transitions can be eliminated (if desired).

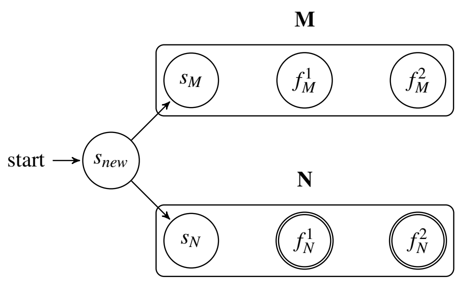
We will see how, given *DFAs* M and N, we can use λ-transition tricks to build *NFA*$_\lambda$*s* P accepting

- the union of the languages M and N accept;
- the concatenation of the languages M and N accept;
- the Kleene star of the language M accepts

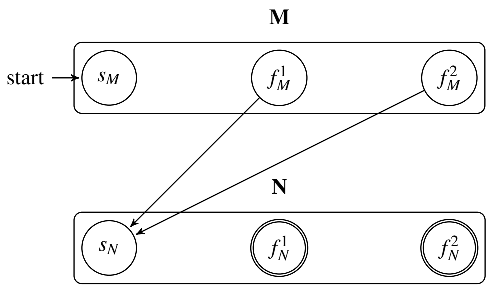
For each construction we first show a picture with the intuition for the construction, then give a precise algorithm, then state the precise correctness claim about the algorithm. The algorithms are sufficiently clear that we omit the correctness proofs, here, though.

9.3 Pictures

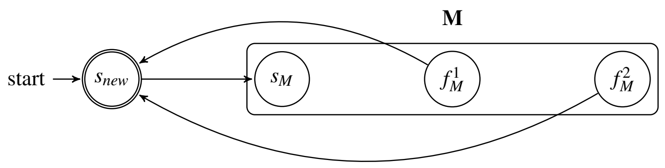
Union To build an NFA_λ recognizing the union of the languages of M and N, all we have to do is make a new start state, and let the machine "guess" which automaton to silently jump to.



Concatenation To build an NFA_λ recognizing the concatenation of the languages of M and N, we just let the accepting states of the first machine 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.



9.4 Algorithms and Correctness

```
Algorithm 3: NFA$_\lambda$-Union
Input: two NFA$_\lambda$s $M$ and $N$
Output: an NFA$_\lambda$ $P$ 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 $\delta_P$ of $P$ consists of the following transitions
        – Every transition of $\delta_M$ and of $\delta_N$ is a transition of $M$;
        – $\lambda$ transitions $s \rightarrow s_M$ and $s \rightarrow s_N$.
return $P = (\Sigma,\ Q_M \cup Q_N \cup \{s_{new}\},\ \delta_P,\ \{s_{new}\},\ F_M \cup F_N)$
```

The following theorem reflects the correctness of our Algorithms.

9.7 Theorem. *If M and N are NFA$_\lambda$s then*

1. *the NFA$_\lambda$ P returned by Algorithm 3 satisfies $L(P) = L(M) \cup L(N)$;*
2. *the NFA$_\lambda$ P returned by Algorithm 4 satisfies $L(P) = L(M)L(N)$;*
3. *the NFA$_\lambda$ P returned by Algorithm 5 satisfies $L(P) = L(M)^*$.*

Proof. These are all very easy to see; we omit a formal proof here. ///

Two notes.

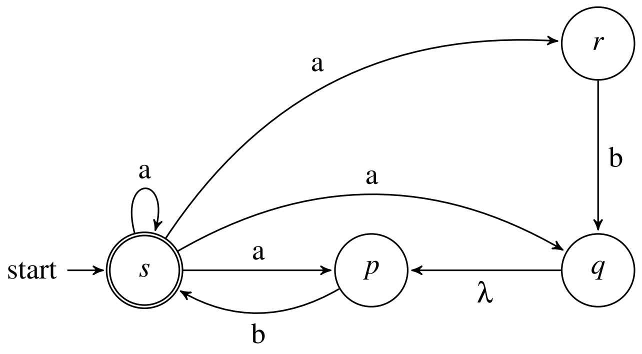
- See Exercise 94 for a subtlety concerning these constructions.
- Your first reaction to the construction in Algorithm 5 might well be: it doesn't have to be that complicated. Exercise 93 asks you to talk yourself out of that reaction.

```
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 $\delta_P$ of P consists of the following transitions
      – Every transition of $\delta_M$ and of $\delta_N$ is a transition of P;
      – For every state f in $F_M$, a λ transition $f \to s_M$
    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 $P = (\Sigma,\ Q_M \cup Q_N,\ \delta_P,\ s_M,\ F_N)$
```

9.5 Exercises

***Exercise* 88.** Make up some NFA_λ, then construct equivalent *NFA* for them.

***Exercise* 89.** Let N be the NFA$_\lambda$ pictured below. Build a DFA M such that $L(M) = L(N)$.



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 $f \rightarrow s_{new}$
  - A λ transition $s_{new} \rightarrow s_M$

  Caution: the old start and accepting states of M are no longer starting or accepting, as states of P.

***Exercise* 90.** Suppose suppose in Algorithm 2 we replaced the line

if δ^+ has transition $p \rightarrow q$ 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.

***Exercise* 91.** *Practice eliminating λ transitions*

Write down some simple *NFAs*. Combine some pairs of them using the technique in Algorithm 3, resulting in an NFA_λ. Now eliminate the λ transitions using Algorithm 2.

Do the same for concatenation and for Kleene-star.

***Exercise* 92.** 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 1 is: convert M and N separately to *DFAs* using the subset construction; then do the product construction on the results.
2. Algorithm 2 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

***Exercise* 93.** The following seems like a simpler way to do the construction that builds an *NFA* for A^* out of an *NFA* for A. Rather than add a new start state s_{new}, just (i) make the old start states 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.

***Exercise* 94.** 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 exercise applies to Algorithm 4 as well.

10 From Regular Expressions to Automata

As we noted earlier, we have two uses of the word “regular” in talking about languages: we speak of “regular expressions” and the languages they denote, and we defined “regular languages” as the languages accepted by *DFAs*. It would be dumb if these two phrases didn’t match up, right? By now we have done all the work we need to justify half of what we want to claim: in this section we collect this work and prove the following theorem.

10.1 Theorem. *If a language L is denoted by a regular expression then there is a DFA accepting L.*

Furthermore, there is an algorithm which constructs from a given regular expression E and DFA M such that $L(M) = L(E)$

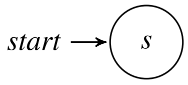
Proof. Algorithm 6 constructs, given E, an NFA_λ, let’s call it M_1 such that $L(M_1) = L(E)$. The correctness of Algorithm 6 follows from the correctness of the sub-algorithms it calls, which we have proven.

Now, we can use Algorithm 2 to convert M_1 to an equivalent ordinary *NFA* M_2. Finally we can use the subset construction on M_2 to arrive at our *DFA* M.

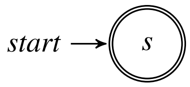
///

Algorithm 6: *RegExp* to NFA_λ

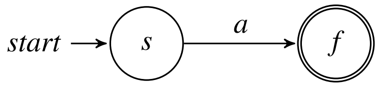
```
Input: a regular expression $E$
Output: an $NFA_\lambda$ $M$ such that $L(M) = L(E)$
The construction is recursive, and proceeds by cases on the form of $E$.
case $E$ is $\emptyset$: do
    return
```



```
case $E$ is $\lambda$: do
    return
```



```
case $E$ is $a$: do
    return
```



```
case $E$ is $E_1 \cup E_2$: do
    Let $M_1$ := RegExp-to-$NFA_\lambda$ $(E_1)$ ;
    Let $M_2$ := RegExp-to-$NFA_\lambda$ $(E_2)$ ;
    return the result of $NFA_\lambda$-Union (Algorithm 3) on $M_1$, $M_2$
case $E$ is $E_1E_2$: do
    Let $M_1$ := RegExp-to-$NFA_\lambda$ $(E_1)$;
    Let $M_2$ := RegExp-to-$NFA_\lambda$ $(E_2)$;
    return the result of $NFA_\lambda$-Concatenation (Algorithm 4) on $M_1$, $M_2$
case $E$ is $E_1^*$: do
    Let $M_1$ := RegExp-to-$NFA_\lambda$ $(E_1)$;
    return the result of $NFA_\lambda$-Kleene-Closure (Algorithm 5) on $M_1$
```

Perspective on Regular Closure Properties

We have proven the following facts. If A and B are regular, 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 9.6).

Now, it is not so clear that the converse of Theorem 10.1 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 these questions.

10.1 Exercises

***Exercise* 95.** [Koz97] Let K be the language denoted by

$$(01 \cup 011 \cup 0111)^*$$

Construct an *NFA* N with 4 states recognizing K. Convert N to a *DFA* M.

***Exercise* 96.** (from Kozen) *Conversion between machines and regular expressions* Give deterministic finite automata accepting the languages denoted by the following regular expressions. (First build NFA_λ, then convert to *NFA*, then to *DFA*.)

1. $(00 \cup 11)^*(01 \cup 10)(00 \cup 11)^*$
2. $(000)^*1 \cup (00)^*1$
3. $(0(01)^*(1 \cup 00) \cup 1(10)^*(0 \cup 11))^*$

11 From Automata to Regular Expressions

In this section we complete the story of the equivalence among *DFA*, *NFAs*, regular grammars, and regular expressions, by showing the one remaining simulation result. Our goal is to prove the following theorem.

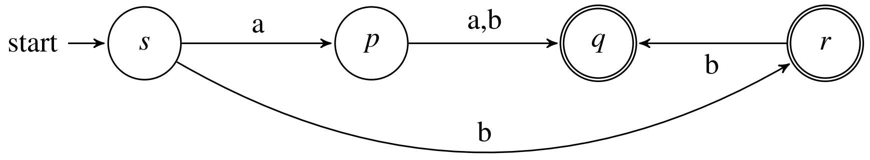
11.1 Theorem. *Given any NFA M we can construct a regular expression E such that $L(E) = L(M)$.*

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 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.

$$\begin{aligned} S &= aP + bR \\ P &= (a+b)Q \\ Q &= \lambda \\ R &= bQ \ \mid \ \lambda \end{aligned}$$

Also note that we did a little simplification, writing $P = aQ + bQ$ as $P = (a+b)Q$. This kind of trick can save writing.

We want to solve for S. We work back to it, using substitution. Substituting for Q:

$$\begin{aligned} S &= aP + bR \\ P &= (a+b)\lambda \\ R &= b\lambda \ \mid \ \lambda \end{aligned}$$

But since λ is the identity for concatenation, we get to

$$\begin{aligned} S &= aP + bR \\ P &= (a+b) \\ R &= b \ \mid \ \lambda \end{aligned}$$

Substituting for R:

$$\begin{aligned} S &= aP + b(b+\lambda) \\ P &= (a+b) \end{aligned}$$

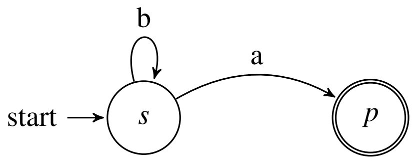
One more substitution and we get

$$S = a(a+b) + bb + b$$

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.

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 b^*a. But we can't find that answer using the simple substitution technique above. We start with the equations

$$S = bS + aP$$
$$P = \lambda$$

which simplify to

$$S = bS + a$$

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. Furthermore, if the empty string λ is not in A, this is the only solution.*

Proof.

> *A^*B is a solution:* This means verifying that $A^*B = A(A^*B) + B$. To see this, just note that the left-hand side consists of all strings consisting of zero or more concatenations from A concatenated with something from B; the right-hand side can be viewed as the set of strings which are either (i) one or more concatenations from A concatenated with something from B or (ii) zero concatenations from A concatenated with something from B. These are clearly the same.

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 entail 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 $AA^{k-1}B \subseteq C$.

$A * B$ is unique if $\lambda \notin A$:

For it suffices to show—in light of the minimality we just proved—that if C is any solution, then $C \subseteq A^*B$. We prove, by induction on string length, that if z is a string in C then $z \in A^*B$. So choose $z \in C$. Since $C = AC + B$ we have either (i) $z \in B$ or (ii) $z \in AC$. In case (i) if $z \in B$ then certainly $z \in A^*B$. In case (ii) if $z \in AC$ then z can be written as xz_1 where $x \in A$ and $z_1 \in C$. Here is where we use the assumption that $\lambda \notin A$: the string z_1 is shorter than z. So by induction hypothes, $z_1 \in A^*B$. Thus $xz_1 \in A(A^*B)$, so $xz_1 \in (A^*B)$.

///

That condition about $\lambda \notin A$ seems obscure, yes. But notice that for an equation $X = AX + B$ if we did have $\lambda \in A$ then taking X to be the set of all strings yields a solution in addition to $X = A^*B$. That is, $\Sigma^* = A\Sigma^* + B$. And there will be lots of other dumb solutions, too ...

11.3 Using Arden's Lemma: First Steps

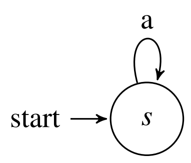
Next are some examples that do use Arden's Lemma but, since they have only one state, don't require the substitution technique.

11.5 Example. Return to the NFA from the previous example, where we arrived at the equation

$$S = bS + a$$

Arden's Lemma immediately yields $S = b^*a$, which matches the answer we get by just looking at the NFA. Good. (The little assumption about λ was satisfied easily, since the language denoted by expression b is just $\{b\}$, which does not contain λ)

11.6 Example. As another baby example, consider this NFA, which accepts no strings at all:

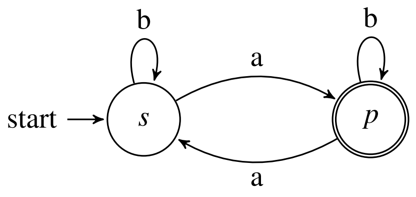


Here the equation we would write down is $S = aS$. (Note that s is not accepting!) That is, the (implicit) B in the framework of Arden's Lemma is $\emptyset$. So Arden's Lemma would yield the regular expression $a^*\emptyset$. And indeed $a^*\emptyset = \emptyset$.

11.7 Example. What happens if we insist on applying Arden's Lemma to an equation that is not really recursive, *i.e.*, one that looks like $X = B$? Well if we write this as $X = \emptyset X + B$, Arden's Lemma tells us that the answer is $\emptyset^* B$. But $\emptyset^*$ is just λ, so we get B after all. That's reassuring.

11.4 The General Technique: Arden's Lemma and Substitution

11.8 Example. 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

$$\begin{aligned} S &= aP + bS \\ P &= aS + bP + \lambda \end{aligned}$$

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

$$P = bP + (aS + \lambda) \tag{1}$$

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

$$P = b^*(aS + \lambda) \tag{2}$$

Simplifying just a little bit, we get

$$P = b^*aS + b^* \tag{3}$$

Now substitute this last expression for P back into that for S and combine terms:

$$\begin{aligned} S &= a(b^*aS + b^*) + bS & (4)\\ &= (ab^*a)S + ab^* + bS & (5)\\ &= (ab^*a + b)S + ab^* & (6) \end{aligned}$$

One more application of Arden's Lemma and we're done:

$$S = (ab^*a + b)^*ab^* \tag{7}$$

Convince yourself that this regular expression really does define the set of strings with an odd number of *a*s.

About λ-transitions Suppose the original NFA has λ-transitions? There's nothing special to do in this case.

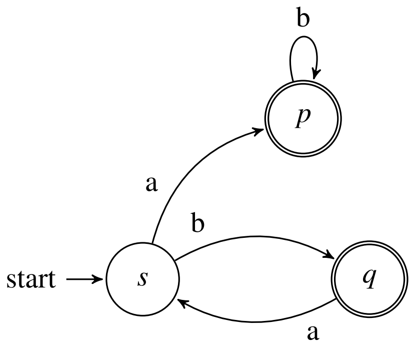
11.9 Example. Suppose we have an NFA with λ-transitions with the corresponding equation below (draw the NFA for yourself):

$$\begin{aligned} S &= P + aQ\\ P &= \lambda\\ Q &= R\\ R &= bS + aR \end{aligned}$$

So $R = a^*bS$. Thus $Q = a^*bS$. Substituting, we get $S = \lambda + aQ = \lambda + a(a^*bS)$. Rearranging, we get $S = (aa^*b)S \cup +\lambda$, which yields $S = (aa^*b)^*$

Several Accepting States There is nothing special needed to deal with *FAs* with more than one accepting state.

11.10 Example.



The equations are:

$$\begin{aligned} S &= aP + bQ \\ P &= bP + \lambda \\ Q &= aS + \lambda \end{aligned}$$

Note that the only effect of having more than one accepting state is that we have more than one equation that mentions λ.

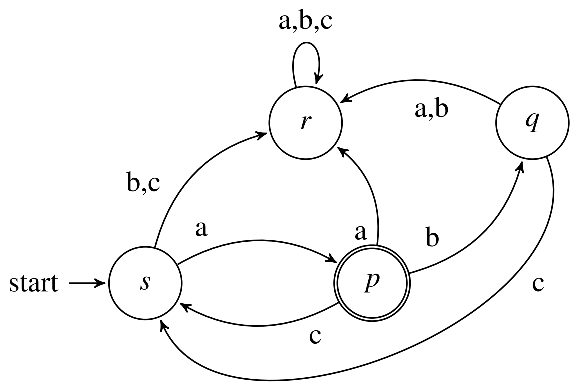
Applying Arden's Lemma to the equation for P we derive $P = b^*\lambda = b^*$. Substituting for P and for Q in the equation for S we get

$$S = ab^* + b(aS + \lambda) \quad = baS + (ab^* + b)$$

A final application of Arden's Lemma yields our answer:

$$S = (ba)^*(ab^* \cup b)$$

11.11 Example. This example is adapted from Robin Milner's beautiful little book on concurrency: *Communicating and Mobile Systems: the* π*-calculus*.



The equations are:

$$S = aP + (b+c)R \tag{8}$$
$$P = aR + bQ + cS + \lambda \tag{9}$$
$$Q = (a+b)R + cS \tag{10}$$
$$R = (a+b+c)R \tag{11}$$

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: $R = (a+b+c)^*\emptyset$, which is $\emptyset$.

Having noted that $R = \emptyset$ we can simplify the equations above.

$$S = aP \tag{12}$$
$$P = bQ + cS + \lambda \tag{13}$$
$$Q = cS \tag{14}$$

Now we can proceed as in the previous example. First substitute the 2nd equation into the first:

$$S = a(bQ + cS + \lambda) \tag{15}$$
$$= abQ + acS + a \tag{16}$$

Now substitute the 3nd equation into the first:

$$S = ab(cS) + acS + a \tag{17}$$
$$= (abc + ac)S + a \tag{18}$$

Now Arden gives us our answer:

$$S = (abc + ac)^* a \tag{19}$$

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.

11.12 Example. For this example we'll just start with the equations.

$$\begin{aligned} S &= aQ + aR + \lambda \\ P &= bS \\ Q &= aP + bQ \\ R &= bS + bP + \lambda \end{aligned}$$

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

$$S = ab^* abS + a(bS + bbS + \lambda) + \lambda$$

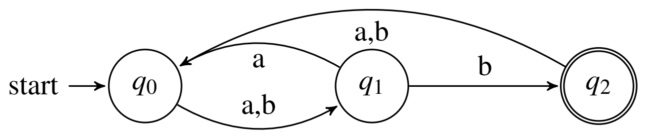
Collect terms, then use Arden

$$\begin{aligned} S &= (ab^* ab + ab + abb)S + a + \lambda \\ &= (ab^* ab + ab + abb)^* (a + \lambda) \end{aligned}$$

So the language accepted by our NFA is defined by the regular expression

$$(ab^* ab + ab + abb)^* (a + \lambda)$$

11.13 Example. Another example; this time we just give an answer, leaving it to you to work through the steps.



One answer is: $((a \cup b)(a \cup b(a \cup b)))^*(a \cup b)b$

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 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.

11.5.1 Equations about Machines

The equivalence between *RegExps* 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 *RegExps* 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.

11.5.2 Extended Regular Expressions

When regular expressions are used in applications, they are almost always given in a richer form, with other operators such as negation, intersection, character classes, back-references to subexpressions, etc. Sometimes these added features are macros added for convenience, in the sense that they could be unfolded into the "official" regular expression notation above. But sometimes these added notations really do have expressive power beyond that of what we defined above.

After our work above we are in a position to see that adding intersection and complement to regular expressions are each syntactic sugar, that is it does not change the expressive power of regular expressions.

11.14 Corollary. *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 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 Exercises

***Exercise* 97.** For each of the automata in Exercises 80 and 81, construct equivalent regular expressions.

***Exercise* 98.** Go back to Exercise 38. Do you see how, after the work in this section, each of the problems there now submit to a mechanical process for deriving the answer? Some of those problems were not easy to do just by intuition. Do some of them now, mechanically.

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 we will derive a test such that 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 what is known as a *characterization* of being regular.

If you have studied the Pumping Lemma before as a means of showing languages to be non-regular, you may know that 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.

12.1 The K-Distinguishability 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. What we will do in this section is something more subtle and useful. We will show how any given language induces a 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 "distinguishes" 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 distinguish 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* indistinguishable with respect to *K, or K*-indistinguishable, *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-distinguishable 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-distinguishable,* and we write $x \not\equiv_K y$.

Note that the relation $\equiv_K$ is defined without any reference to machines of grammars. It is a purely "combinatorial" idea!

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,

$$[x]_K = \{y \mid x \equiv_K y\}$$

12.2 Check Your Reading. *Before you go any further, do Exercise 99.*

12.1.1 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.

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 distinguishable from the other with respect to C. Since: for $i \neq j$ the strings a^i and a^j are distinguishable by the string $z = b^i$. (Note by the way that the string $z = b^j$ would also distinguish 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 distinguishable 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 distinguishable by the string $z = b^i$. ///

12.8 Example. Let $E = \{x \in \{a,b\}^* \mid x \neq \lambda$ and x begins and ends with the same symbol$\}$. There are 5 $\equiv_E$ classes, that is, the index of E is 5.

Proof. Here are the five $\equiv_E$ classes:

1. the set consisting only of the empty string
2. the set of strings starting with a and ending with a
3. the set of strings starting with a and ending with b
4. the set of strings starting with b and ending with a
5. the set of strings starting with b and ending with b

To see that this is correct, first observe that these 5 classes partition Σ^*, meaning that they are mutually disjoint, and their union equals Σ^*. So it suffices to show that (i) two strings in different classes are distinguishable, while (ii) within each class the strings are indistinguishable.

These details are left as an exercise. ///

12.9 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 distinguishable from the other with respect to E. Since: for $i \neq j$ the strings $a^i b$ and $a^j b$ are distinguishable by the string $z = a^i$. ///

12.10 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 distinguishes 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-distinguishable 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 $s \xrightarrow{x}\!\!\!\!\rightarrow p$ 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.

12.11 Lemma (Distinguishability 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 $\hat{\hat{\delta}}_M(s,x) = \hat{\hat{\delta}}_M(s,y)$. 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 $\hat{\hat{\delta}}_M(s,x) = \hat{\hat{\delta}}_M(s,y)$ 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

12.12 Corollary. *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 12.11. ///

Now we easily have

12.13 Corollary. *Let K be a language, and suppose there is an infinite collection $x_1, x_2, \dots$ of strings such that any two of them are K-distinguishable. 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 12.13.

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 14. But for now, Exercise 102 should be very helpful for your intuition.

12.4 Using Distinguishability to Prove Languages Not Regular

Let's see how to use Corollary 12.13 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-distinguishable.

12.14 Example. Let us prove that the following language is not regular.

$$K \overset{\text{def}}{=} \{a^n b^n \mid n \geq 0\}$$

We follow the script.

Define $X \overset{\text{def}}{=} \{a^n \mid n \geq 0\}$. 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.

When a language has infinitely many distinguishability 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!

I have written the following examples 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 distinguishable strings.

12.15 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 distinguishable from the others. $\lambda, a, aa, \ldots, a^i, \ldots$
3. To show that there are distinguishable, 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 distinguish them.

12.16 Example. Let $B = \{w \in \{a,b\}^* \mid w \text{ is an even-length palindrome}\}$

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 distinguishable from the others. $\lambda, ab, aab, \dots, a^i b, \dots$
3. To show that there are distinguishable, consider any $a^i b$ and $a^j b$.

   We have $a^i b \not\equiv_B a^j b$ because we can use the word $z = ba^i$ to distinguish them.

12.17 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 distinguishable from the others. $\lambda, (, ((, \dots (^i \dots$
3. To show that there are distinguishable, consider any $(^i$ and $(^j$.

   We have $(^i \not\equiv_C (^j$ because we can use the word $z =)^i$ to distinguish 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.

12.18 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 distinguishable 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 distinguishes them.

3. To show that there are distinguishable, 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.

12.19 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 12.18. 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.[6]

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.

[6]In fact, a famous conjecture, the "twin primes conjecture" asserts that there are infinitely many pairs of primes that differ by 2. This is unproven but number-theorists generally believe it to be true.

2. To show this, let $a^{p_1}, a^{p_2}, a^{p_3}, \dots$ be the sequence of words corresponding to the sequence of primes described above.

   We claim that these a^{p_i} are pairwise distinguishable.

3. To show that there are distinguishable, 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 distinguishes x and y.

12.5 Exercises

***Exercise* 99.** 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 $\equiv_A$-distinguishable, that is $ab \not\equiv_B ba$.
   Find a specific word z that witnesses that fact.

   (b) The words λ and abb are $\equiv_A$-distinguishable. Find a specific word z that witnesses that fact.

   (c) The words λ and ba are $\equiv_A$-distinguishable. 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 $\equiv_B$-distinguishable. 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 $\equiv_B$-distinguishable. 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 $\equiv_C$-distinguishable. Find a specific word z that witnesses that fact.

   (b) The words λ and abb are $\equiv_C$-distinguishable. Find a specific word z that witnesses that fact.

   (c) Explain why the words bba and ba are $\equiv_C$-equivalent.

***Exercise* 100.** 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.

***Exercise* 101.** 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 $\{u0v \mid u,v \in \{0,1\}^*, v \text{ has length } (n-1)\}$

1. For each n let L_n be the set of strings whose nth-to-last symbol is a 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-distinguishable
3. Conclude that the smallest *DFA* recognizing L_n has at least 2^n states.

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*.

***Exercise* 102.** For each of the examples in Section 12.1 for which 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.

***Exercise* 103.** Let Σ be the alphabet $\{0,1\}$

1. Let A be the language $\{w \mid \text{ length of } w \text{ is divisible by } 4\ \}$. Prove that any *DFA* recognizing A must have at least four states, by writing down four strings that are pairwise $\equiv_A$-distinguishable.

   Note for this and subsequent parts: To prove your 4 strings $\{w_1, w_2, w_3, w_4\}$ mutually indistinguishable 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_iz and w_jz is in A. Since you have 4 strings for this part, you will have $\binom{4}{2} = 6$ pairs to address.

2. Let B be the language $\{0^i1^j \mid i,j \geq 0\}$. Prove that any *DFA* recognizing B must have at least three states, by writing down three strings that are pairwise $\equiv_B$-distinguishable, and proving them to be $\equiv_B$-distinguishable, as outlined in the previous part.

3. Let C be the language $\{w \in \{0,1\}^* \mid\ w \text{ has at most three 1s }\}$. Prove that any *DFA* recognizing C must have at least five states, by writing down five strings that are pairwise $\equiv_C$-distinguishable, and proving them to be $\equiv_C$-distinguishable.

   (Rather than giving $\binom{5}{2} = 10$ individual arguments for indistinguishability you might prefer to give a generic argument!

***Exercise* 104.** Show the following languages to be non-regular.

These exercise 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 12.13, you should proceed as follows. You want to describe an infinite set of strings that are mutually distinguishable. 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 distinguishes them.

1. $\{a^n b^{2n} \mid n \geq 0\}$
2. $\{a^n b^m c^n \mid n, m \geq 0\}$
3. $\{a^n b^m \mid n \leq m\}$
4. $\{a^n b^m \mid n \geq m\}$
5. $\{a^i b^n c^n \mid i \geq 0, m \geq 0\}$.
6. $\{w \mid \exists x\, w = xx^R\}$, over the alphabet $\{a, b\}$.
7. $\{w \mid \exists x\, w = xx\}$, over the alphabet $\{a, b\}$.
8. the set of strings of as and bs whose length is a perfect square .
9. $\{a^n \mid n \text{ is a perfect cube}\}$
10. $\{a^n \mid n \text{ is a power of } 2\}$

***Exercise* 105.** *Bounded exponents.*

For each of the following languages, either prove that it is regular or prove that it is not regular.

1. $A = \{a^i b^j \mid i \geq j \text{ and } j \leq 100\}$
2. $B = \{a^i b^j \mid i \geq j \text{ and } j \geq 100\}$

***Exercise* 106.** *Regular subset*

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.

***Exercise* 107.** *Infinite subset of non-regular* Prove that no infinite subset of $\{a^n b^n \mid n \geq 0\}$ is regular.

***Exercise* 108.** *Applying closure properties* 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.

***Exercise* 109.** *Closure tricks* Assume that the following language is not regular. $A = \{a^n \mid n \text{ is a perfect square}\}$

Prove that the set D 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.

***Exercise* 110.** *Closure tricks again* 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.

1. $Neq = \{a^k b^l \mid k \neq l\}$. (Caution: this set is not the same set as $\overline{Eq}$)
2. For this problem let us introduce the temporary notation w^o to stand for the result of taking a string w and replacing as by bs and vice versa.

   Let $G = \{ww^o \mid w \in \{a,b\}^*\}$. Show G is not regular.
3. $K = \{w \mid w$ has an unequal number of as and bs$\}$ (Caution: this set is also not the same set as $\overline{Eq}$)

***Exercise* 111.** It is important that M be a *DFA*, not just an *NFA*, in Lemma 12.11. 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 12.11 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.

***Exercise* 112.** Lemma 12.11 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's a hint! A smart *DFA* won't work.)

13 *DFA* Minimization

Suppose A is a regular language. Can there be more than one *DFA* M with $L(M) = A$?

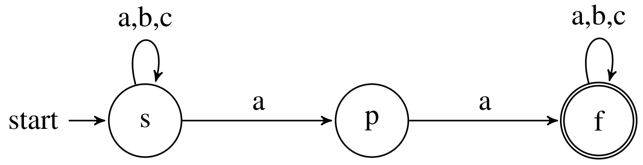
Let's first note that we need to be precise when we say "more than one *DFA*". Surely if we systematically rename the states of a *DFA* that shouldn't count as a different *DFA*, right? So whenever we speak *DFAs* being "the same" or not, we mean "up to renaming of states".

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 inaccesible states to M we'll get a different *DFA* but we will not have changed the language accepted.

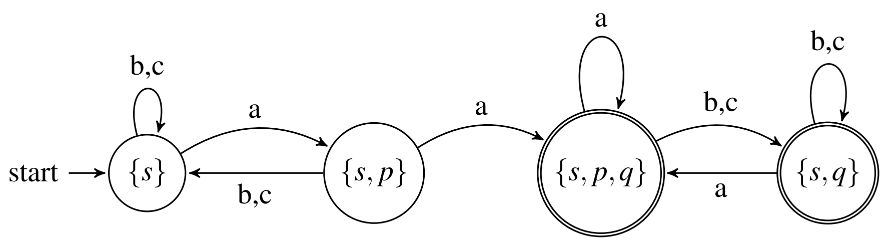
So the real question we want to ask is, *Suppose A is a regular language, can there be more than one DFA with no inaccesible states recognizing A, up to renaming of states?* The answer is yes; see several examples in this section.

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.

13.1 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.

13.1 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. **So henceforth we make that assumption: all *DFAs* we work with in this section have no inaccessible 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.

13.2 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-distinguishability 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-distinguishability compares strings, while $\approx$ compares machine states.

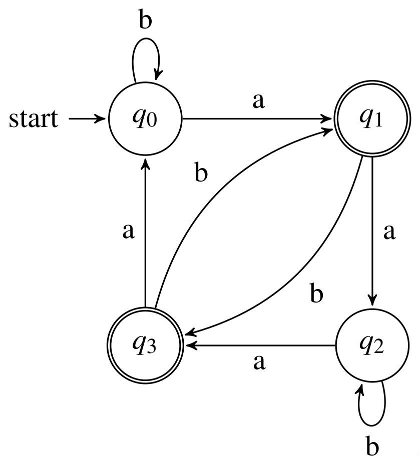
13.2 Definition. *Let* $M = (Q, \Sigma, \delta, q_0, F)$ *be a DFA; let p and q be states of* M. *We say that states p and q are* M-equivalent, *written if* $p \approx_M q$, *if*

$$\textit{for every } x \in \Sigma^* : \textit{ if } p \overset{x}{\twoheadrightarrow} p' \textit{ and } q \overset{x}{\twoheadrightarrow} q' \textit{ then } p' \in F \textit{ if and only if } q' \in F$$

When p and q are not M-equivalent we write $p \not\approx_M q$. Unfolding the definition, we have that $p \not\approx_M q$ if

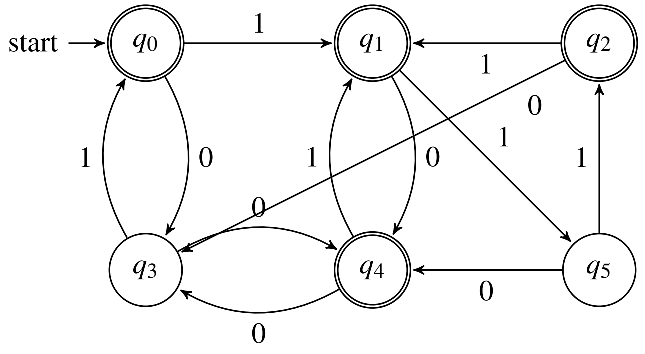
$$\exists x \in \Sigma^* : \text{ with } p \overset{x}{\twoheadrightarrow} p' \text{ and } q \overset{x}{\twoheadrightarrow} q' \text{ but exactly one of } p' \in F \text{ or } q' \in F$$

13.3 Example. Starting with



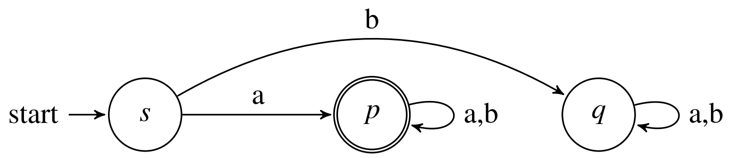
you can check that $q_0 \approx q_2$ and that $q_1 \approx q_3$.

13.4 Example. Let's start with this *DFA*.



Here it turns out that $q_0 \approx q_2$ and $q_0 \approx q_4$ and $q_3 \approx q_5$.

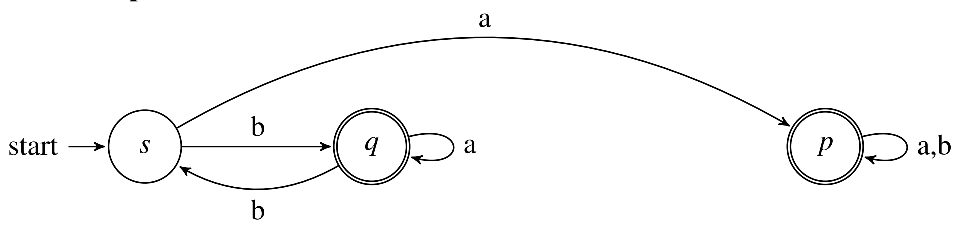
13.5 Example. In this *DFA*



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:

13.6 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.

13.7 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 13.1 , 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. We will do this in Section 13.

But first, some observations about $\approx$.

13.8 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 13.4

13.3 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 13.2) 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.

13.9 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 7. 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.

Algorithm 7: *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 $Q = \{q_0, q_1, \dots, q_n\}$.
We maintain a list $D$ of pairs of states $\{i, j\}$, with $i \neq j$.
initialize: $D := \{\{i, j\} \mid (q_i \in F$ and $q_j \notin F)$ or $(q_i \notin F$ and $q_j \in F)\}$.
The invariant is :
after k iterations of the repeat loop, D will contain pair $\{i, j\}$ iff there is a
  string x of length no greater than k distinguishing $q_i$ and $q_j$.
repeat
    Consider each $\{i, j\} \notin D$ in turn ;
    For each such $\{i, j\}$, consider each alphabet symbol $a \in \Sigma$ ;
    suppose $q_i \xrightarrow{a} q_m$ and $q_j \xrightarrow{a} q_n$ ;
    If $\{m, n\} \in D$, add $\{i, j\}$ to $D$.
until no change in $D$;
return the set of all those pairs $\{q_i, q_j\}$ of states such that $\{i, j\}$ is not in the
  set $D$.
```

13.10 Lemma. *Algorithm 7 correctly computes the $\approx$ relation for any DFA.*

Proof. The algorithm is guranteed to halt, since there are fewer than $(|Q|^2/2)$ distinct $\{i, j\}$ pairs considered at each stage, and each time through the loop we add at least one pair to the list D. And it is clear that the invariant we stated above holds for the set D.

It remains to prove that D contains precisely the pairs $\{i, j\}$ such that $q_i \not\approx q_j$.

First, suppose $\{i, j\} \in D$. Then certainly $q_i \not\approx q_j$; this is clear from our invariant.

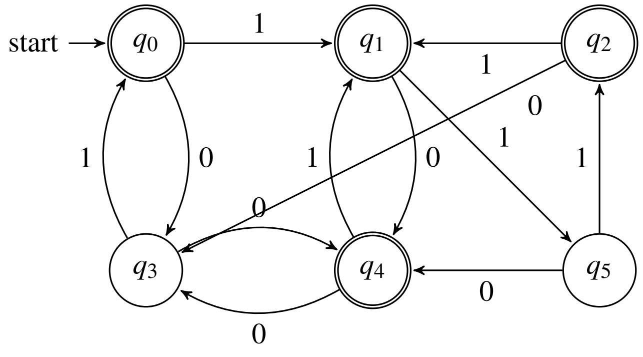
Next suppose that $q_i \not\approx q_j$; we want to show that $\{i, j\} \in D$.

We prove that $\{i, j\} \in D$ by induction on the length of a shortest such x. Indeed we prove that $\{i, j\} \in D$ by stage s for $s = |x|$. That is to say, what we do here is to prove that our invariant is actually an “if and only if.”

Let x be a string that distinguishes q_i and q_j. If $x = \lambda$ then $\{i, j\}$ will be put into D at stage 0, initialization. If $x = ay$ has length $s+1$ then let q_m satisfy $q_i \xrightarrow{a} q_m$ and $q_j \xrightarrow{a} q_n$.

Observe that $q_m \not\approx q_n$ and that the string y (of length s) distinguishes these states. By induction, then, $\{m,n\}$ is in D by stage s. Thus we put $\{i, j\}$ into D by stage $s+1$. ///

13.11 Example. Let’s look again at the *DFA* from Example 13.4.



We can work through Algorithm 7 by hand, if we make a little table for the D relation. Strictly speaking we are computing all the pairs $\{i, j\}$ of non-M-equivalent states but clearly we don’t need to explicitly mark both (i, j) and (j,i) in our table, so we just pay attention to the upper-right triangle.

Here is the end result of filling in the table. The “ _“ spaces are table slots that were not marked during the course of the algorithm.

$[q_0]$	1	_	λ	_	λ
	$[q_1]$	1	λ	1	λ
		$[q_2]$	λ	_	λ
			$[q_3]$	λ	_
				$[q_4]$	λ
					$[q_5]$

Once we have the table, slots (i, j) that have entires are the ones where $q_i \not\approx q_j$. So the empty (i, j) give us the $\approx$ relation. That is, if slot (i, j) is unmarked then we can collapse q_i and q_j.

So we verify what we did intuitively before, that $q_0 \approx q_2$ and $q_0 \approx q_4$ and $q_3 \approx q_5$.

The most important application of *DFA* state equivalence is in minimizing *DFAs*. We will explore this in depth in Section 13. But the following application is also useful, and will come in handy soon.

13.4 The Collapsing Quotient of a *DFA*

We have seen in Section 13.2 how to compute the $\approx$relation for a *DFA*. Now we'll use it to build a minimization of that *DFA*.

13.12 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.*

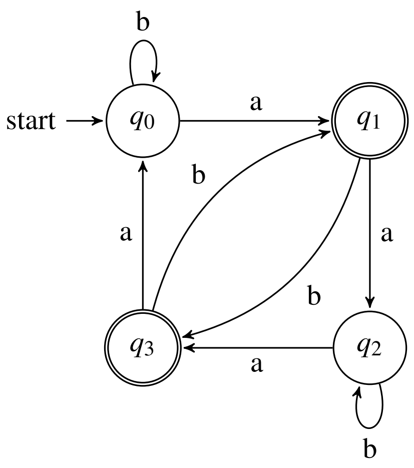
13.4.1 The problem of "well-definedness"

There is a technical matter to dispose of first. It is important to understand that the construction in Definition 13.12 makes sense only in light of the properties of $\approx$ in Lemma 13.8. 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 13.8 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.

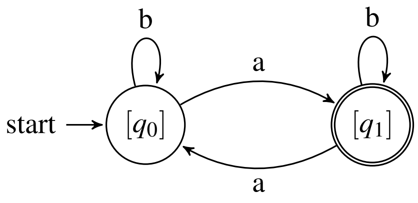
Then, in the collapse-*DFA* we are defining, $[p]$ and $[q]$ would be the same state (since $p \approx q$), but where would we transition to from this state on input a? One the one hand, the definition says we should go to $[p']$, because $p \xrightarrow{a} p'$, but the definition also says we should 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 13.8, 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.*

13.13 Example. Starting with

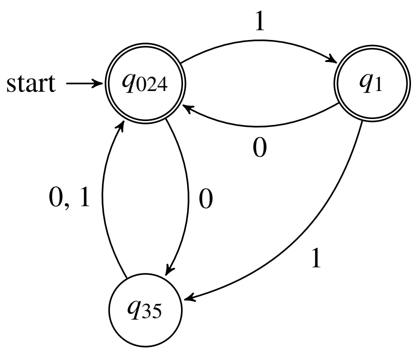


we observed in the last section that $q_0 \approx q_2$ and that $q_1 \approx q_3$. So when we collapse we get

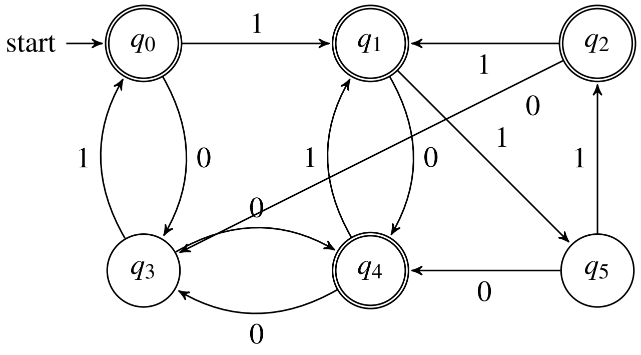


Important. In the picture we labelled the left state as "$[q_0]$" but we could just as easily have written "$[q_2]$"; the same goes for "$[q_1]$" versus "$[q_3]$". Make sure you understand this: $[q_0]$ and $[q_2]$ are different names for *exactly the same thing!*

13.14 Example. If we start with the *DFA* in Example 13.4, compute $\approx$as we did there, and then do the collapsing construction, here is the *DFA* we get.


$$\begin{tikzcd}
\text{start} \arrow[r] & q_{024} \arrow[rr, bend left, "1"] \arrow[d, bend left, "0"] & & q_1 \arrow[ll, bend left, "0"] \arrow[dll, bend left, "1"] \\
& q_{35} \arrow[u, bend left, "{0, 1}"] & &
\end{tikzcd}$$

13.15 Example. Let's look once again at the *DFA* from Example 13.4.


$$\begin{tikzcd}
\text{start} \arrow[r] & q_0 \arrow[r, "1"] \arrow[d, bend left, "0"] & q_1 \arrow[d, bend left, "0"] \arrow[dr, "1"] & q_2 \arrow[l, "1"] \arrow[dll, "0"] \\
& q_3 \arrow[u, bend left, "1"] \arrow[r, bend left, "0"] & q_4 \arrow[u, bend left, "1"] \arrow[l, bend left, "0"] & q_5 \arrow[u, "1"] \arrow[l, "0"]
\end{tikzcd}$$

We can work through Algorithm 7 by hand, if we make a little table for the D relation. Strictly speaking we are computing all the pairs $\{i, j\}$ of non-M-equivalent states but clearly we don't need to explicitly mark both (i, j) and (j, i) in our table, so we just pay attention to the upper-right triangle.

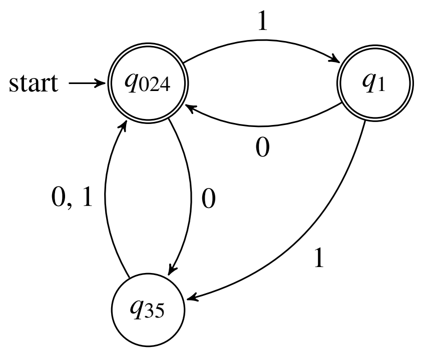
Here is the end result of filling in the table. The " _" spaces are table slots that were not marked during the course of the algorithm.

$[q_0]$	1	_	λ	_	λ
	$[q_1]$	1	λ	1	λ
		$[q_2]$	λ	_	λ
			$[q_3]$	λ	_
				$[q_4]$	λ
					$[q_5]$

Once we have the table, slots (i, j) that have entires are the ones where $q_i \not\approx q_j$. So the empty (i,j) give us the $\approx$ relation. That is, if slot (i,j) is unmarked then we can collapse q_i and q_j.

In this example we see that $q_0 \approx q_2$ and $q_0 \approx q_4$ and $q_3 \approx q_5$.

Here is the *DFA* we get.



13.16 Check Your Reading. *For each of the other examples in Section 13.2, build the collapsing quotient DFA.*

13.5 $M_\approx$ Does the Right Thing

The next Lemma says that when we collapse we don't change the ultimate behavior of a *DFA*.

13.17 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 $M_\approx$. ///

13.18 Check Your Reading. *Why did we choose to write x in the induction step as ya instead of ay?*

13.19 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 $L(M_{\approx}) = L(M)$.*

13.6 $M_{\approx}$ is special

Some natural questions come up now. Let M be any *DFA*, with no inaccesible 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 13.21.

The *DFA* $M_{\approx}$ has the following nice property, which can be stated as: indistinguishable strings end up in equivalent states. Note that this is a converse to the Distinguishability Lemma 12.11, but it only holds for collapsed *DFAs*!

13.20 Lemma. *Let $M_{\approx}$ be the output of Algorithm 7 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 $r' \in F$. ///

13.21 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 13.20 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 12.12 to conclude that the number states is actually equal to the number of classes.

///

It follows that $M_\approx$ cannot be collapsed.

13.22 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. ///

13.6.1 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 "isomomorphism" is in other contexts, that is what we will establish).

We will use the idea from Section 13.7. 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 state-equivalence relation on $Q_1 \cup Q_2$ as defined in Section 13.7. 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.

13.7 Application: Testing Equivalence of *DFAs*

Here is another application of $\approx$: testing 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 13.2 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 $s_1 \approx s_2$.

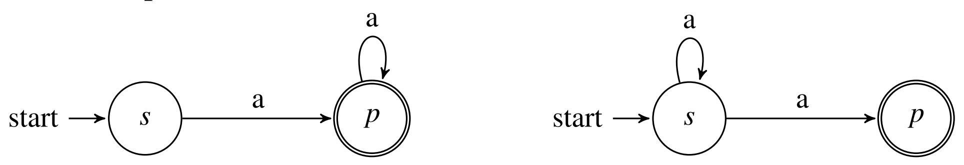
13.23 Lemma. *Let* $M_1 = (\Sigma, Q_1, \delta_1, s_1, F_1)$ $M_2 = (\Sigma, Q_2, \delta_2, s_2, F_2)$ *be as just described, and let* $\approx$*be the state-equivalence relation as just described. Then* $s_1 \approx s_2$ *if and only if* $L(M_1) = L(M_2)$.

13.8 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 state-equivalence might be in the presence of nondeterminism: think about it.

The next observation is the killer, though: it is not true that minimal 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.

13.24 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.

13.9 Exercises

***Exercise* 113.** (From [Koz97])

Consider the *DFAs* below, written in the table-notation introduced in Example 6.6

List the equivalence classes for the equivalence relation $\approx$.

1.

		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

2.

		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

3.

		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

4.

		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

5.

		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

6.

		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

***Exercise* 114.** Make an adjustment to Algorithm 7 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.

***Exercise* 115.** (From [Koz97])

Consider the *DFAs* from Exercise 113, 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.

***Exercise* 116.** For each language E, construct the minimal *DFA* M such that $L(M) = E$.

1. $E = (a \cup b)^*ab.$
2. $E = ((a \cup b)(a \cup b))^* \cup (a \cup b)^*b$
3. $E = (a+b)^*aba(a+b)^*$
4. $E = a^*b^* \cup b^*a^*$

Don't be clever and invent a minimal *DFA* from scratch - treat these as exercises 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.

***Exercise* 117.** This exercise is designed to help you understand the well-definedness requirement, used crucially in Definition 13.12.

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 $M_\approx$.

***Exercise* 118.** This exercise is designed to help you understand the well-definedness requirement, used crucially in Definition 13.12.

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 13.8. (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 $M_{\approx}$.

14 The Myhill-Nerode Theorem

Our work in Sections 13.2 and 13 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 section 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 Exercise 119 before reading further. After you do that you should be able to make a conjecture about what's coming next.

14.1 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".[7]

[7]It turns out that there are always countably many classes, but that will never matter to us.

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.

14.1 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 character 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 $L(D_K) = K$.

14.1.1 The problem of "well-definedness" again

Once again we have that issue of well-definedness, as introduced in Section 13.4.1. 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 $[x]_K$ is the same set as $[y]_K$.

14.2 Check Your Reading. *Go back to Exercise 119 above where you generated the equivalence classes for various languages. Start with language A there and write the classes as*

$$[x_1]_A,\ [x_2]_A,\ \dots$$

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.

14.3 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 14.3 we can be confident that Definition 14.1 really does define a *DFA*.

14.1.2 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.

14.4 Theorem. *Let K and D_K be as in Definition 14.1. 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

14.5 Theorem. *A language K is regular if and only if Index(K) is finite.*

Proof. One direction is just the contrapositive of Corollary 12.13. The other direction is the content of Definition 14.1 and Theorem 14.4. ///

14.6 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 12.12 ///

14.2 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 13 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.

14.7 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 14.1: 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.*

14.3 Exercises

***Exercise* 119.** Each language K below is regular. So each has finitely many $\equiv_K$-equivalence classes. List at least three strings in each class. Then construct a *DFA* for K. Finally—this is the most important part—compare the equivalence classes for the language with the states of your *DFA*.

1. Let $A = \{x \in \{a\}^* \mid x \text{ has length divsible by 3 }\}$.
2. Let $B = \{x \in \{a,b\}^* \mid x \text{ has length divsible by 3 }\}$.
3. Let $D = \{a^n b^m \mid n,m \geq 0\}$.
4. Let $G = \{a^n b^m \mid n,m \geq 0 \text{ and } n+m \text{ is divisible by 3 }\}$.
5. For fixed k let $H_k = \{w \in \{a,b\}^* \mid \text{the } k\text{th symbol from the end of } w \text{ is } a\}$.
6. Let $J = \{x \mid x \text{ has no consecutive repeated characters}\}$
7. Let $K = \{x \in \{a,b\}^* \mid x \neq \lambda \text{ and } x \text{ begins and ends with the same symbol}\}$.

***Exercise* 120.** Using Corollary 12.12 we can compute a bound on how many states are required for a DFA to accept certain regular languages. We use that here to prove that in the worst case there can be an exponential blowup in the number of states when converting *NFAs* to *DFAs*.

Let $\Sigma = \{a,b\}$. Let K be the language consisting of all strings w over Σ such that the nth letter from the end of w is a b.

$$K = (a \cup b)^* \, b \, (a \cup b) \dots (a \cup b)$$

where there are $(n-1)$ occurrences of $(a \cup b)$ after the b.

1. Show that there is an *NFA* recognizing H_k with $n+1$ states.
2. Describe the equivalence classes of the Myhill-Nerode relation $\equiv_R$. There are 2^n of them!
3. Conclude that the size of the smallest *DFA* for H_k is no less than 2^n.
4. You just proved:

   The conversion from NFAs to equivalent DFAs can induce an exponential blowup in the size of the state space.

***Exercise* 121.** For each of the languages L in Exercise 119, 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 14.1.
3. For each of the equivalence classes C, find a regular expression denoting C.

***Exercise* 122.** Generalizing from Exercise 121. 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 $\delta^*(x) = q$.

Hint. This problem could have gone in Section 11.

***Exercise* 123.** Let K be a language such that $\equiv_K$ has finite index. 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. Exercise 122 will help.

***Exercise* 124.** Following up on Exercise 123. Let K be a language with finite index and suppose you are given an arbitrary *DFA* recognizing K. Explain how to algorithmically compute regular expressions for each of the $\equiv_K$ equivalence classes.

15 Decision Problems about Regular Languages

Some decision problems concerning automata and regular expressions

Important. The input to the problems below is **not** a *language.* Indeed 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. So the inputs below are, for example, a DFA, or a regular expression, etc. Then the question that gets asked is about the language that the DFA (or whatever) represents.

15.1 DFA Membership

DFA Membership

INPUT: DFA M, word w

QUESTION: $w \in L(M)$?

Algorithm 8: DFA Membership

```
just simulate the machine.
```

Complexity: $O(|w|)$, since we take one step per character in the word w.

15.2 DFA Emptiness

DFA Emptiness

INPUT: DFA M

QUESTION: $L(M) = \emptyset$?

Algorithm 9: DFA Emptiness

```
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 $O(|Q| + k|Q|) = O(|Q|)$.

15.3 DFA Universality

DFA Universality

INPUT: DFA M

QUESTION: $L(M) = \Sigma^*$?

stress: Cannot do exhaustive search !!

Algorithm 10: DFA Universality

```
construct a DFA $M'$ with $L(M') = \overline{L(M)}$;
call the algorithm DFA Emptiness on $M'$;
if this returns YES then
    return YES
else
    return NO
```

Complexity: The same as for the *DFA Emptiness*, since the construction of M' can be done in constant time. So the complexity is $O(|Q|)$ where Q is the set of states of M.

15.4 DFA Subset

DFA Subset

INPUT: DFAs M_1, M_2

QUESTION: $L(M_1) \subseteq L(M_2)$?

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 11: DFA Subset

```
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 $N$;
if this returns YES then
    return YES
else
    return NO
```

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 $O(|Q_1| \times |Q_2|)$.

15.5 DFA Equality

DFA Equality

INPUT: DFAs M_1, M_2

QUESTION: $L(M_1) = L(M_2)$?

We saw one algorithm for this based on $\approx$. Here is another, that relies only on the product and complement constructions.

The idea: $X = Y$ iff $X \subseteq Y$ and $Y \subseteq X$.

Algorithm 12: DFA Equality

```
**call** *DFA Subset* on $(M_1, M_2)$ ;
**call** *DFA Subset* on $(M_2, M_1)$ ;
**if** *both of these return YES* **then**
    **return** YES
**else**
    **return** NO
```

Complexity: We make two calls to *DFA Subset*, which are each $O(|Q_1| \times |Q_2|)$, so the complexity of *DFA Equality* is $O(|Q_1| \times |Q_2|)$.

15.6 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 13: *DFA 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.

15.7 NFA and Regular Expression inputs

Even though DFAs, NFAs, Regular Expressions, etc are all equivalent from the perspective of language-recognition power, the complexity of algorithms can be very different if the input is in one form rather than another.

15.8 Complexity of Regular Conversion Algorithms

- Convert NFA to DFA: requires (worst-case) exponential time, since the number of states can blow up by an exponential.
- Convert a regular expression to an NFA with λ-transitions: linear time.
- Convert an NFA$_\lambda$ to an ordinary NFA: can take $O(|Q|^3)$ time.
- Convert a regular expression to an (ordinary) NFA: $O(n^3)$ time, by the previous remarks.
- Convert an NFA or DFA to a regular expression: can take exponential time.

15.9 NFA Membership

NFA Membership

INPUT: *NFA N*, word w

QUESTION: $w \in L(M)$?

An 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 Exercise 82.

15.10 Regular Expression Membership

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:

> *Regular Expression Membership*
>
> INPUT: a regular expression α, a word w
>
> QUESTION: is $w \in L(\alpha)$?

Here is a straightforward algorithm.

Algorithm 14: Regular Expression Membership

```
given $\alpha$, 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.

15.1 Check Your Reading. *Be sure you are able to describe in careful pseudocode algorithms for each of the preceding decision problems when the input is a DFA, an NFA, or a regular expression.*

15.11 Exercises

***Exercise* 125.** Give decision procedures for each of the following problems.

Each of your answers should be a script that does nothing besides

- call one or more algorithms we have developed in these notes for building new automata from old ones, and
- call one or more algorithms we have developed in this section for deciding properties of automata

Use the algorithms in this section as a guide to how formal to be.

To give you an idea of how things things go, the first one is done for you.

1. Given a *DFA M*: Does M accept any strings of even length?

   Solution.

   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....

   The algorithm

```
Algorithm 15: 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 $L(P)$ is empty then
    return NO
else
    return YES
```

2. Given a *DFA M*: Does M accept every even length string? (This is not the same as asking whether M accepts *precisely* the even-length strings.)

3. Given a *DFA M*: Does M reject infinitely many strings?

4. Given *DFAs* M and N: Do M and N differ on infinitely many inputs? (More formally: is the symmetric difference between $L(M)$ and $L(N)$ infinite?)
5. Given *RegExps* E and F: Do E and F define the same language?

Part III

Context-Free Languages

16 Context-Free Grammars

A context-free grammar (typically abbreviated CFG) is a formalism for generating strings. It does so using *rules*[8] 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.

16.1 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[9]
- 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*, of the form*

  $$X \to \beta$$

  where $X \in V$ *and* β *is an arbitrary string over* $V \cup \Sigma$.

Here is how grammars generate strings of mixed terminals and variables.

16.2 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.

[8]called *productions* by some authors

[9]Some authors call these *nonterminals*

16.3 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:*

$$L(G) \overset{def}{=} \{x \in \Sigma^* \mid S \Longrightarrow^* x\}$$

Note that although we defined derivations using strings that can be made up both variables and terminals, the language of grammar is by definition a set of *terminal* strings.

16.4 Example. A baby example:

$$\begin{aligned} S &\rightarrow aS \\ S &\rightarrow Sb \\ S &\rightarrow \lambda \end{aligned}$$

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 the previous grammar would be

$$S \rightarrow aS \mid Sb \mid \lambda$$

Here is a derivation, of the string *aab*.

$$S \Longrightarrow aS \Longrightarrow aaS \Longrightarrow aaSb \rightarrow aab$$

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 this grammar is the set $\{a^n b^m \mid n,m \geq 0\}$.

16.5 Check Your Reading. *The order of the symbols in a rule matters! Suppose we changed the previous example to be*

$$S \rightarrow aS \mid bS \mid \lambda$$

Convince yourself that this grammar generates all *strings over $\{a,b\}$.*

16.6 Example. Here is a context-free grammar G generating arithmetic expressions. For simplicity here 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

$$\begin{aligned} E &\to E+E \mid E*E \mid I \\ I &\to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{aligned}$$

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. 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.[10]

Here is a derivation in the grammar G.

$$\begin{aligned} E &\Longrightarrow E+E \Longrightarrow E+E*E \Longrightarrow I+E*E \Longrightarrow 2+E*E \\ &\Longrightarrow 2+E*I \Longrightarrow 2+E*3 \Longrightarrow 2+I*3 \Longrightarrow 2+5*3 \end{aligned}$$

This derivation derives the string $2+5*3$. So we say that $2+5*3 \in L(G)$.

It is worth being conscious of the completely different perspective afforded by grammars and automata. Regular Grammars *generate* strings, they are output-only "machines." Automata *accept* strings, they are input-only. When we develop an equivalence between these two formalisms we are doing something more subtle than mere simulation.

Suppose we didn't want to be restricted to expressions where the numbers were single digits? A great thing about grammars is that they compose easily.

16.7 Example (Decimal Numbers)**.** Here is a grammar to generate decimal representations of natural numbers. We don't want them to start with a 0 ...

$\Sigma = \{0, 1, \ldots, 9\}$

$V = \{N, M\}$

The start symbol is N.

$$\begin{aligned} N &\to 1M \mid 2M \mid \ldots \mid 9M \\ M &\to 0M \mid 1M \mid 2M \mid \ldots \mid 9M \mid \lambda \end{aligned}$$

[10]Though if the grammar writer uses "S" as one of the variables, they will typically not bother to say out loud that S is the start symbol.

Now here is a grammar for arithmetic expressions over (decimal representations of) natural numbers.

$$\begin{aligned} E &\to E+E \mid E*E \mid N \\ N &\to 1M \mid 2M \mid \dots \mid 9M \\ M &\to 0M \mid 1M \mid 2M \mid \dots \mid 9M \mid \lambda \end{aligned}$$

We will systematically explore this idea of building grammars in a modular way in Section 16.4.

16.1 Parse Trees

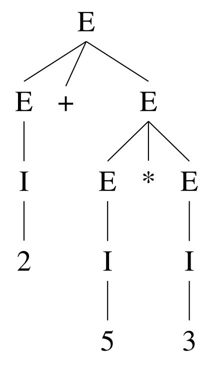
Parse trees, sometimes call derivation trees, are similar to derivations in that they witness the fact that a grammar can derive a word, but they are much more informative.

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 is totally clear once you see an example.

16.8 Example. Refer to the grammar in Example 16.6.

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 16.6 and there are eight interior nodes of this tree!

16.9 Check Your Reading. *Make sure you see how this parse tree is gotten from the derivation.*

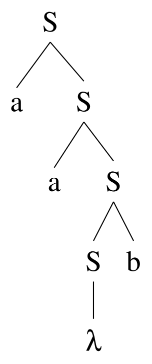
The reason that parse trees are important is that they impose a *structure* on a derived string. That is, when we have a parse tree we know more than just the fact that a string is derivable from a grammar: we know *how*.

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.

16.10 Example. Refer to the grammar in Example 16.4, 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 $S \rightarrow \lambda$ 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.

16.1.1 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.

16.11 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:

16.12 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 the following fact

> *For every parse tree T there is a unique leftmost derivation yielding T.*

16.13 Check Your Reading. *The derivation shown in Example 16.6 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.*

16.14 Check Your Reading. *Really, the only reason to even define the notion of "leftmost derivation" is so that we have some canonical particular derivation to pick out that corresponds to a parse tree.*

But there is nothing special about "leftmost". 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.

The extra information contained in a derivation is typically of no use to us... Some authors prefer to treat derivations as primary, though. So reading about context-free grammars you may hear a lot of talk about leftmost derivations. But the parse trees are where the action is.

16.2 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.

16.15 Definition. *A* regular grammar *is a context free grammar such that all of its productions are of the form*

$$X \to aY \quad or \quad X \to \lambda$$

where $X, Y \in V$ and $a \in \Sigma$.

Of course it is allowed that $X = Y$ in the definition above.

16.16 Examples.

1. The grammar in Example 16.7 is regular.
2. Here is a grammar that generates $\{a^n b^n \mid n \geq 0\}$ (not the same as $a^* b^*$, right?) that is, where the numbers of *a*s and *b*s 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).
3. 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.
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

   $$\begin{aligned} S &\to aS \mid B \\ B &\to Bb \mid \lambda \end{aligned}$$

   but it generates the language namely $a^* b^*$, which of course is a regular language. The following regular grammar generates the same language.

   $$\begin{aligned} S &\to aS \mid bB \mid \lambda \\ B &\to bB \mid \lambda \end{aligned}$$

It turns out that regular grammars can be viewed as really nothing more than a notational variant of *NFAs*.

Building an Automation 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.

16.17 Example. Let G be the following regular grammar (whose start symbol is Q_0).

$$\begin{aligned} Q_0 &\to aQ_0 \mid bQ_0 \mid bQ_1 \\ Q_1 &\to aQ_2 \mid bQ_2 \\ Q_2 &\to \lambda \end{aligned}$$

When we do the construction outlined above we get the automaton in Example 8.4.

Once we prove that the automaton thus constructed recognizes the same language that the grammar generates, we have the following.

16.18 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.

16.19 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
$$Q_i \to aQ_j$$
- for each accepting state $q_i \in F$ of M, P has a production
$$Q_i \to \lambda$$

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 $x \in L(M)$.

///

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 16.19 and 16.18.

16.20 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."

16.21 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.

16.3 More Examples and non-Examples

16.22 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:

$$S \rightarrow aSb \mid \lambda$$

16.23 Example (Palindromes)**.** A *palindrome* is a string x that reads the same backwards and forwards, in other words x is equal to its reversal: $x = x^R$. 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 $\Sigma = \{a, b, c, \dots, z\}$:

$$\begin{aligned} S &\rightarrow \lambda \mid aSa \mid bSb \mid cSc \mid \dots \mid zSz \\ S &\rightarrow a \mid b \mid c \mid \dots \mid z \mid \lambda \end{aligned}$$

A typical derivation:

$$S \Longrightarrow rSr \Longrightarrow raSar \Longrightarrow racScar \Longrightarrow racecar$$

16.24 Example (Doubles)**.** Another way to describe the even-length palindromes is as the set of all words of the form xx^R as x ranges over all words. It is easy to tweak the grammar above to get just the even-length palindromes. Thus $EPal = \{w \mid \text{ for some } x \in \Sigma,\ w = xx^R\ \}$ is a context-free language.

Interestingly, if we consider the set of "doubled words" $D = \{w \mid \text{ for some } x \in \Sigma,\ w = xx\ \}$ this is *not* a context-free language. It's tricky to prove, but it is a fact that there can be no CFG generating D.

On the other hand, 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 see ...)

16.25 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*. The proof is given in Section 22.1.

16.26 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.

The next example illustrates 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.

16.27 Example (Natural Language)**.** For example 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

$$\begin{aligned} S &\Longrightarrow NP \quad VP \\ &\Longrightarrow Det \quad NP \quad VP \\ &\Longrightarrow the \quad NP \quad VP \\ &\Longrightarrow the \quad boy \quad VP \\ &\Longrightarrow the \quad boy \quad V \quad NP \\ &\Longrightarrow the \quad boy \quad chased \quad NP \\ &\Longrightarrow the \quad boy \quad chased \quad the \quad N \\ &\Longrightarrow the \quad boy \quad chased \quad the \quad dog \end{aligned}$$

16.28 Check Your Reading. *Make a derivation of the sentence*

the dog ate a boy

Are there finitely many sentences derivable in this grammar, or infinitely many?

16.29 Check Your Reading. *Continuing this example: Add the symbol* and *to* Σ *and add this rule to the grammar:*

$$S \to S \, and \, S$$

What new sentences can you derive?

Are there finitely many sentences derivable in this grammar, or infinitely many?

16.30 Example (Arithmetic Expressions)**.** Here is a CFG generating the legal infix expressions over $+$ and $*$ and decimal numbers. The terminal alphabet is $\Sigma = \{+,-,*,/,0,1,\dots,9\}$ 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.

$$\begin{aligned} E &\to E+E \mid E-E \mid E*E \mid E/E \mid N \\ N &\to 1M \mid 2M \mid \dots \mid 9M \\ M &\to 0M \mid 1M \mid 2M \mid \dots \mid 9M \mid \lambda \end{aligned}$$

16.31 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

$$\Sigma = \{\texttt{while}, \texttt{do}, \texttt{if}, \texttt{then}, \texttt{else}, :=, ;, a, b, \dots z, 0, 1, \dots 9\}$$

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 Exercise 135.

$$\begin{aligned}
S &\to A \ | \ C \ | \ W \ | \ S;S \\
A &\to I \ := \ E \\
W &\to \texttt{while}\, E \; do\, S \\
C &\to \texttt{if}\, E \,\texttt{then}\, S \,\texttt{else}\, S \\
C &\to \texttt{if}\, E \,\texttt{then}\, S \\
E &\to \text{ [the rules for expressions Example 16.30]} \\
E &\to I \\
I &\to \text{ [the rules for identifiers from Exercise 135]}
\end{aligned}$$

16.32 Check Your Reading. *Make a derivation of the following program.*

$$\texttt{while}\, x \, do \, y \ := y+1;\ x := x-y$$

16.4 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 16.4.4, we can't do everything that we might like to do.

16.4.1 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 16: 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).

16.4.2 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 $S \to S_1S_2$

- 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 $S \to \lambda$

16.33 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

16.34 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. ///

16.4.3 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.

16.35 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 exercises you will apply these ideas to building context-free grammars corresponding to regular expressions.

16.4.4 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.

16.36 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 16.25; a proof that it is context-free is in Section 22.1.)

Then we can exhibit two context-free languages A_1 and A_2 such that $A_1 \cap A_2$ is not context-free. 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. (You are asked to do so in Exercise 129.) But $A_1 \cap A_2$ is $\{a^n b^n c^n \mid n \geq 0\}$ which we know to be non-context-free. ///

16.37 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.

16.38 Theorem. *Let A be context-free and let R be regular. Then $A \cap R$ is context-free.*

Proof. The easiest proof of this uses pushdown automata, covered later. So we leave this unproved for now. ///

16.5 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.

16.39 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.

16.40 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. ///

16.41 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 16.40 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 22

16.6 Exercises

***Exercise* 126.** Some easy stuff

1. What's the difference between a grammar and a language?
2. Can a grammar be infinite?
3. Does it make sense to talk about taking the complement of a grammar?
4. Does it make sense to talk about a grammar accepting a string?

***Exercise* 127.** (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. $abaaba$

***Exercise* 128.** 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.

***Exercise* 129.** Construct a context-free grammar generating the language

$$A_1 = \{a^n b^n c^m \mid n, m \geq 0\}$$

Construct a context-free grammar generating the language

$$A_2 = \{a^m b^n c^n \mid n, m \geq 0\}$$

***Exercise* 130.** For each of the following languages K, construct a context-free grammar G such that $L(G) = K$.

1. $\{a^i b^i c^k d^k \mid i,k \geq 0\}$
2. $\{a^i b^j c^k d^m \mid i = j \text{ or } k = m\}$
3. $\{a^{i_1} b^{i_1} a^{i_2} b^{i_2} \ldots a^{i_n} b^{i_n} \mid n \geq 0, \text{ each } i_j \geq 0\}$

***Exercise* 131.** For each of the following languages K, construct a context-free grammar G such that $L(G) = K$.

1. $\{a^i b^j \mid i = 2j\}$

   Hint. Start with a grammar for $\{a^i b^i \mid i \geq 0\}$, then make a little tweak.
2. $\{a^i b^j c^k \mid i + j = k\}$

***Exercise* 132.** Let $\Sigma = \{+, *, a, b\}$. Let K be the language of prefix arithmetic expressions over Σ. Write a context-free grammar to generate K. Show parse trees and leftmost derivations for several strings. Here are sample strings in K.

$+\,a\,a$	$+\,+\,a\,b\,*\,b\,a$
$+\,a\,*\,a\,b$	$+\,a\,*\,a\,+\,b\,b$

***Exercise* 133.** Prove that if A is context-free then the set A^R of reversals of strings in A is context-free.

***Exercise* 134.** Look back at examples of *DFAs* and *NFAs* from previous sections, and construct the corresponding regular grammars.

***Exercise* 135.** 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.

***Exercise* 136.** 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.

***Exercise* 137.** Inspired by Exercise 136, 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.

Unfortunately this is false.

Remember that the following language A is not regular: $A = \{a^n b^n \mid n \geq 0\}$. Show that this A can be generated by a left-right linear grammar.

This shows that we cannot allow ourselves both left-linearity and right-linearity in the same grammaar without moving beyond regularity.

17 Proving correctness of grammars

Here we explore how one proves that a certain CFG generates a certain language.

17.1 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.

17.2 Examples

17.1 Example. *Equal as and bs*

Let E be the set of strings over $\{a,b\}$ with an equal number of *a*s and *b*s.

Let G be the following CFG:

$$S \rightarrow \lambda \mid SS \mid aSb \mid bSa$$

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 characters 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 $L(G)$.

17.2 Example. The grammar in Example 17.1 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 $S \Rightarrow cSc \Rightarrow^* cw'c = w$.

17.3 Example. Even though the grammar in Example 17.2 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.

$$\begin{array}{lcl} S & \to & bS \mid aA \mid \lambda \\ A & \to & aS \mid bA \end{array}$$

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 . . .]

17.3 Exercises

***Exercise* 138.** *Prefixes: as versus bs* 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.

***Exercise* 139.** *Generating palindromes* 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.)

***Exercise* 140.** 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 ...

***Exercise* 141.** Compare these two grammars:

G_1 :	G_2 :
$S \rightarrow ASB$	$S \rightarrow AB$
$A \rightarrow a$	$A \rightarrow aA \mid a$
$B \rightarrow bb$	$B \rightarrow bbB \mid bb$

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 $S \Rightarrow^* \ldots$

***Exercise* 142.** *Balanced parentheses*

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.

$$S \rightarrow SS \mid aSb \mid \lambda$$

Prove that G generates the set *Bal* of "balanced parentheses".

***Exercise* 143.** Prove that the following grammar generates all strings over $\{a,b\}$.

$$S \to aS \mid Sb \mid bSa \mid \lambda$$

***Exercise* 144.** *Equal as and bs again* 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 142 consider the quantity $\#a(u) - \#b(u)$ as u ranges over prefixes of a string w. Characterize the strings in the language of equal numbers of as and bs in terms of this function. Then you can proceed as in problem 142.

***Exercise* 145.** Consider the language L over $\Sigma = \{a,b\}$: $L = \{x \mid x \text{ is } \textit{not} \text{ of the form } ww\}$. Show that L is context-free.

Hint. This is not easy. The following exercises gradually build up to a solution of that problem.

1. Write a CFG for $X = \{uav \mid u,v \in \{a,b\}^*, |u| = |v|\}$.
2. Observe that a CFG for $Y = \{ubv \mid u,v \in \{a,b\}^*, |u| = |v|\}$ is a trivial variation on the one for X.
3. Describe the languages XY and for YX. Observe that it is now easy to write a CFG for the concatenations XY and for YX.
4. Now 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
   $$y_1 \cdots y_{i-1} b y_{i+1} \cdots y_m z_1 \cdots z_{i-1} a z_{i+1} \cdots z_m$$
5. Put the pieces together.

18 Ambiguity

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.

$$E \rightarrow E+E \mid E*E \mid 0 \mid 1 \mid 2 \mid \dots \mid 9$$

Consider these three derivations

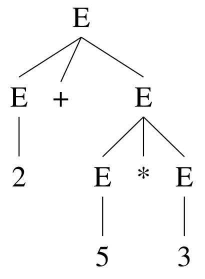
$$E \Longrightarrow E+E \Longrightarrow E+E*E \Longrightarrow 2+E*E \rightarrow 2+5*E \rightarrow 2+5*3$$

$$E \Longrightarrow E+E \Longrightarrow E+E*E \Longrightarrow E+5*E \rightarrow E+5*3 \rightarrow 2+5*3$$

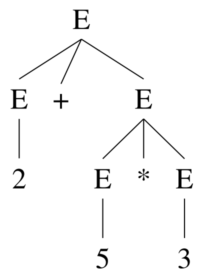
$$E \Longrightarrow E*E \Longrightarrow E+E*E \Longrightarrow 2+E*E \rightarrow 2+5*E \rightarrow 2+5*3$$

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.

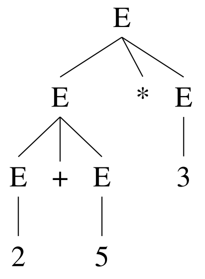
First derivation:



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.

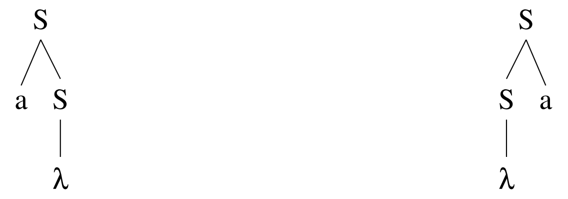
18.1 Definition. *A context-free grammar is* ambiguous *if there is at least one string w such that there is more than one parse tree yielding w in G.*

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 *ambiguous* if there is at least one string w such that there is more than leftmost derivation of w in G.

18.2 Example. This grammar is ambiguous:

$$S \to aS \mid Sa \mid \lambda$$

There are even two parse trees for deriving the single string a!



An unambiguous grammar deriving the same strings is:

$$S \to aS \mid \lambda$$

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.

18.1 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.

We won't be systematic here, but just focus on two typical and important kinds of ambiguity, *precedence* ambiguity and *grouping* ambiguity.

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.

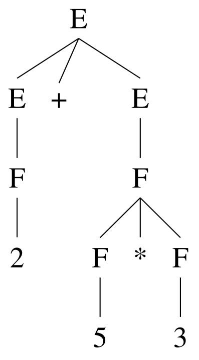
By the way, note that when people speak of "higher precedence" what they are saying is "lower in the parse tree." Don't get confused by that. (Lower in the parse tree means "happens earlier in the evaluation process", hence "precedes", hence "precedence.")

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").

$$\begin{aligned} E &\to E+E \mid F \\ F &\to F*F \mid 0 \mid 1 \mid 2 \mid \dots \mid 9 \end{aligned}$$

Now we can still derive 2+5 * 3, but in only one way:



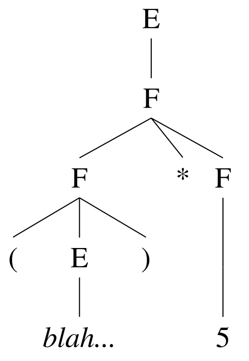
And the natural way to evaluate this will do the multiplication first, and get the answer 17.

Overriding Precedence

Of course we want to provide our users with the possibility of overriding the precedence our grammar has built in. For example she 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.**

$$\begin{aligned} E &\to E+E \mid F \\ F &\to F*F \mid (E) \mid 0 \mid 1 \mid 2 \mid \dots \mid 9 \end{aligned}$$

18.3 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.

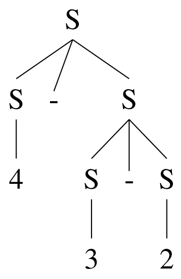
Grouping Ambiguity

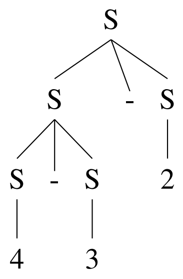
A different kind of ambiguity can arise with a single operator.

18.4 Example. Consider this little expression grammar, with just the single subtraction operator:

$$S \to S-S \mid 0 \mid \dots \mid 9$$

This is ambiguous, since 4 - 3 - 2 can be derived in two ways

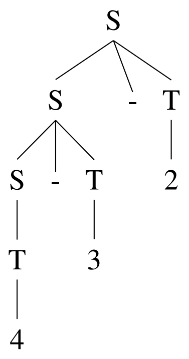




The natural way to evaluate these would yield 3 and -1 respectively. Assume we would like to make a grammar that enforced *left* associativity for -, that is, preferring the answer -1. How do we do it? The trick is to make sure that the symbol *S* can only be recursive "on the left." To do that we introduce another variable, *T*, to help out.

$$\begin{aligned} S &\to S - T \mid T \\ T &\to 0 \mid \ldots \mid 9 \end{aligned}$$

Now we can still derive 4-3-2, but in only one way:



18.5 Check Your Reading. *Make a grammar for exponentiation (use any symbol you like , maybe an up-arrow ↑) but enforce* right *associativity.*

18.6 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)

$$\begin{aligned} E &\to E+T \mid T \\ T &\to I*T \mid I \\ I &\to 0 \mid 1 \mid 2 \mid \dots \mid 9 \end{aligned}$$

18.2 Inherent Ambiguity

Suppose we are given a grammar G that we know to be ambiguous. Can we always construct an unambiguous equivalent G'? whether or not G is ambiguous?

The answer is no. There exist grammars G that are ambiguous but such that there are no unambiguous grammars generating the same language.

18.7 Example.

$$\begin{aligned} S &\to AB \mid C \\ A &\to aAb \mid ab \\ B &\to cBd \mid cd \\ C &\to aCd \mid aDd \\ D &\to bDc \mid bc \end{aligned}$$

The grammar generates

$$K = \{a^n b^n c^m d^m \mid m,n \geq 1\} \cup \{a^n b^m c^m d^n \mid m,n \geq 1\}$$

18.8 Check Your Reading. *Build two derivation trees for the string aabbccdd.*

It is a little 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...

18.9 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 18.9 and Definition 18.1. One gives a property of *grammars*, the other gives a property of *languages*.

18.3 Exercises

***Exercise* 146.** *Basic ambiguity* For each grammar, decide whether it is ambiguous or not. If it is, prove it (by exhibiting a string with two different parse trees). 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])

1.

$$S \to aS \mid Sb \mid ab$$

2.

$$\begin{aligned} S &\to aA \mid \lambda \\ A &\to aA \mid bB \\ B &\to bB \mid \lambda \end{aligned}$$

3.

$$\begin{aligned} S &\rightarrow AaSbB \mid \lambda \\ A &\rightarrow aA \mid a \\ B &\rightarrow bB \mid \lambda \end{aligned}$$

4.

$$S \rightarrow aSb \mid aSbb \mid b$$

5.

$$\begin{aligned} S &\rightarrow A \mid B \\ A &\rightarrow abA \mid \lambda \\ B &\rightarrow aBb \mid \lambda \end{aligned}$$

***Exercise* 147.** *Ambiguity and arithmetic I*

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.

$$\begin{aligned} E &\rightarrow E+T \mid T \\ T &\rightarrow T*I \mid I \\ I &\rightarrow a \mid b \mid c \end{aligned}$$

1. The grammar G is ambiguous not only because the precedence of the operators 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 left-associative operators.

   Change the grammar of Figure G^* so that it is still unambiguous and generates the same strings, but with + parsed as being right-associative and * as left-associative.

2. 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 $E_1 - E_2$.

   Enrich the grammar G^* 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 +.

3. 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".

   Enrich the grammar G^* 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 exponentiation is not associative. Repeated exponentiation conventionally groups to the right. For example, the conventional value of $2 \uparrow 3 \uparrow 2$ is $2 \uparrow 9$. Be sure your grammar induces the correct grouping!

4. Extend the grammar G^* 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 $+$.)

***Exercise* 148.** *Ambiguity and arithmetic II (from HMU)* The following grammar generates *prefix* arithmetic expressions over $+$, $*$, and $-$.

$$\begin{aligned} E &\to +EE \mid *EE \mid -EE \mid I \\ I &\to a \mid b \mid c \end{aligned}$$

Build a parse tree for the term $+*-abab$.

Is this grammar ambiguous? Explain why or why not.

***Exercise* 149.** *Ambiguity and Boolean logic* Here is a grammar G generating formulas of propositional logic over the alphabet $\Sigma = \{\wedge, \vee, \equiv, \neg, (,), p, q, r, s, \ldots\}$

$$\begin{aligned} B &\to B \wedge B \mid B \vee B \mid B \equiv B \mid \neg B \mid (B) \mid I \\ I &\to p \mid q \mid r \mid s \mid \ldots \end{aligned}$$

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.

***Exercise* 150.** *Ambiguity and programming languages*

Consider the following grammar over the alphabet

$$\Sigma = \{\texttt{if}, \texttt{then}, \texttt{else}, \texttt{statement}, \texttt{condition}\}$$

$$\begin{aligned} S &\to \texttt{if}\, C\, \texttt{then}\, S \\ S &\to \texttt{if}\, C\, \texttt{then}\, S\, \texttt{else}\, S \\ S &\to \texttt{statement} \\ C &\to \texttt{condition} \end{aligned}$$

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.
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.
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.

***Exercise* 151.**

1. Be able to explain the difference between *G is an ambiguous context-free grammar* and *L is an inherently ambiguous context-free language*.

2. Does it make sense to talk about a language being ambiguous?
3. Does it make sense to talk about a grammar being inherently ambiguous?
4. Does it make sense to talk about a parse tree being ambiguous?

19 Refactoring Context-Free Grammars

Sometimes we would like to change a grammar into another grammar that generates the same language but has nicer properties. The motivation is that we want to make equivalent "nice" forms of grammars to make them easier to reason about. Think of this as "refactoring" grammars.

The results are important, but equally important is the set of techniques used. Focus on the idea of transforming an object to obtain an end-result while maintaining an invariant, and reasoning about the construction using induction.

There is a theme running through this chapter, the notion of a *worklist algorithm.* These are simply algorithms in which a list of tasks is maintained, and at each stage we grab a task and complete it, perhaps adding new tasks to the worklist, and keep going until the worklist is empty. Such algorithms arise in lots of different scenarios.

19.1 Eliminating Useless Rules

The intuition here is that grammar variables that cannot be reached from the start symbol are "dead code", and variables that cannot lead to terminal strings may as well never be generated.

19.1 Definition.

- *A variable A in a grammar G is* reachable *if there is some G-derivation $S \Rightarrow^* \alpha A \beta$.*
- *A variable A in a grammar G is* generating *if there is some G-derivation $A \Rightarrow^* w$ with $w \in \Sigma^*$.*
- *A variable A in a grammar is* useful *if it is both generating and reachable, otherwise we say that it is* useless.
- *A rule is a grammar is* useless *if it involves any useless variables.*

To get rid of useless rules in a grammar G, we must first compute which variables of G are useful, that is, which variables are reachable and generating.

19.1.1 Computing Reachable Variables

Computing the reachable variables is very easy.

```
Algorithm 17: 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 $\alpha$ 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. ///

We can eliminate rules involving non-reachable variables without changing the language.

```
Algorithm 18: 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. ///

19.1.2 Computing Generating Variables

Computing the generating variables is also easy; we just have to reason backwards.

```
Algorithm 19: 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 $\alpha$ either in $\Sigma$ or in $V'$ 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. ///

We can eliminate rules involving non-generating variables without changing the language.

```
Algorithm 20: 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 Correctness. similar to the argument for *Eliminate Unreachable* ///

19.1.3 Putting Things Together

If we want to eliminate all useless variables from a grammar, we first eliminate the non-generating variables, and then eliminate the non-reachable variables.

```
Algorithm 21: 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 *EliminateNonGenerating* 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)*). ///

In case the above argument seems fussier than it needs to be, have a look at Exercise 153 to see see that things can be more subtle than they appear at first glance.

19.2 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

$$\begin{aligned} S &\to aA \\ A &\to bA \;\mid\; c \\ C &\to cB \mid c \end{aligned}$$

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.

19.2 Eliminating Chain Rules

19.3 Definition. *A* chain rule *in a grammar is one of the form $A \to B$, where B is a variable.*

Our goal is to prove the following theorem.

19.4 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.

19.5 Lemma. *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.*

Proof. This is easy: the added rule can be simulated by the original rules. ///

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 22: 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 19.5.

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 $S \overset{*}{\Longrightarrow} w$ 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

$$S \overset{*}{\Longrightarrow} xB\alpha \Rightarrow xC\alpha \overset{*}{\Longrightarrow} w$$

we must have next rewritten C via some rule $C \to \beta$

$$S \overset{*}{\Longrightarrow} xB\alpha \Rightarrow xC\alpha \Rightarrow x\beta\alpha \overset{*}{\Longrightarrow} w$$

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:

$$S \overset{*}{\Longrightarrow} xB\alpha \Rightarrow x\beta\alpha \overset{*}{\Longrightarrow} w.$$

This contradicts our assumption that the derivation we started with was shortest. So this completes the proof that grammar G' performs as advertised. ///

19.3 Eliminating Erasing Rules

19.6 Definition. *An* erasing rule *in a grammar is one of the form $A \to \lambda$.*

Some authors call erasing rule "λ-rules." But this is potentially confusing because it suggests that such rules have something to do with λ-transitions in automata, which is not the case. [11]

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.

Our goal is to prove the following theorem.

19.7 Theorem. *For every CFG G there exists a CFG G' such that*

1. $L(G') = L(G) - \{\lambda\}$

[11] In fact, it is *chain rules* that are closely related to λ-transitions in automata! See Section 19.6

2. *G' has no erasing rules.*

Furthermore, there exists an algorithm to compute G' from G.

That "$L(G') = L(G) - \{\lambda\}$" business seems weird. Why can't we just say "$L(G') = L(G)$"? The reason is that a grammar G' with no erasing rules can never generate λ; so if the original grammar G did generate λ, we can't ask for $L(G') = L(G)$. The best we can hope for is what we wrote, namely $L(G') = L(G) - \{\lambda\}$.

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?).

19.8 Example.

$$\begin{aligned} S &\to AB \\ A &\to a \\ B &\to \lambda \end{aligned}$$

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. The key idea in getting rid of individual rules that generate λ is to enlarge our perspective and think about *sequences* of rule that generate λ. We do that now.

19.9 Definition (Nullable Variables)**.** *A variable A is* nullable *in a grammar G if there is a derivation $A \Longrightarrow^* \lambda$ in G.*

Note that this captures a more general concept than erasing rules. 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.

The strategy for eliminating erasing rules is :

1. Build an auxiliary grammar with extra rules designed to compensate for not having erasing rules. This will involve working with nullable variables.

2. *Then* delete erasing rules from this augmented grammar.

For instance, in the little grammar of Example 19.8, 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, but this is the basic idea.

Here is an algorithm to compute the set of all nullable variables in a grammar.

```
Algorithm 23: Compute Nullable Variables
Input: a CFG $G$
Output: the set of nullable variables of $G$
initialize $Nullable$ to be $\{A \mid A \to \lambda$ is a rule $\}$;
repeat
    if there is a rule $A \to \alpha$ such that each element of $\alpha$ 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 19 Left as an exercise. ///

Now, back to eliminating erasing rules. The key idea of the algorithm is contained in the following lemma. It expresses the fact that certain transformations of a grammar leave the language of the grammar unchanged.

19.10 Lemma. *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.*

Proof. This is easy: the added rule can be simulated by the original rules. ///

It will useful to isolate the key step in the algorithm as a definition all its own:

19.11 Definition. *Let $G = (\Sigma, V, P, S)$. Let P^* be the smallest set of rules such that*

1. *every rule of P is in P^*, and*
2. *if $A \to \alpha B \beta$ is in P^* (with $\alpha, \beta \in (V \cup \Sigma)^*$) and B is nullable in G, then $A \to \alpha\beta$ is in P^*.*

Be careful to note that Definition 19.11 can require adding several "variations" for a given rule, as the next example shows.

19.12 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^*:

$$A \to acCB \qquad A \to aBC$$
$$A \to aBcB \qquad A \to acC$$
$$A \to aBcC \qquad A \to ac$$
$$A \to acB$$

one new rule for each non-empty subset of the occurrences of nullable variables.

Here at last is the algorithm for eliminating erasing rules.

Algorithm 24: 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 $\lambda$-rules
compute the nullable variables of $G$;
define $G^* = (\Sigma, V, P^*, S)$ where $P^*$ is defined as in Definition 19.11
define $G' = (\Sigma, V, P', S)$ where $P'$ is $P^*$ with all erasing rules removed
return $G'$
```

To prove that Algorithm 24 is correct we identify a few lemmas.

19.13 Lemma. *Let $G = (\Sigma, V, P, S)$. Let P^* be defined as in Definition 19.11. Then if we set $G^* = (\Sigma, V, P^*, S)$, we have $L(G^*) = L(G)$.*

Proof. This is an easy consequence of Lemma 19.10. ///

19.14 Lemma. *Let $G = (V, \Sigma, P, S)$. Let P^* be defined as in Definition 19.11, and set $G^* = (\Sigma, V, P^*, S)$.*

Then for $w \neq \lambda$, if $S \overset{*}{\Longrightarrow} w$ then there is a derivation in which no λ-rule of G^* 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 $S \overset{*}{\Longrightarrow} x\delta$ is the shortest leftmost derivation of $x\delta$ in G^*, 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

$$S \overset{*}{\Longrightarrow} xB\delta \Rightarrow x\delta$$

This occurrence of B originated by a derivation step using a rule $A \to \alpha B\beta$

$$S \overset{*}{\Longrightarrow} x'A\theta \Rightarrow x'\alpha B\beta\theta \equiv x'\alpha B\delta \overset{*}{\Longrightarrow} xB\delta \Rightarrow x\delta$$

But then $A \to \alpha\beta$ is in P^* by the way we built P^*. So here is a shorter derivation of $x\delta$:

$$S \overset{*}{\Longrightarrow} x'A\theta \Rightarrow x'\alpha\beta\theta \equiv x'\alpha\delta \overset{*}{\Longrightarrow} x\delta$$

This contradicts our assumption that the derivation we started with was shortest.

///

19.15 Theorem (Correctness of Eliminate Erasing)**.** *Algorithm 24 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 it is possible to 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 other than $S \to \lambda$ 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 19.13.

Now to prove $L(G') = L(G^*)$: Since the rules of G' are a subset of those of G^*, it is obvious that $L(G') - \{\lambda\} \subseteq L(G^*)$. So it remains to prove that $L(G^*) \subseteq L(G') - \{\lambda\}$. For this it suffices to prove that for any non-λ string $w \in L(G^*)$ there is a derivation of w in G^* that does not use λ-rules. This follows from Lemma 19.14

This completes the proof that grammar G' performs as advertised. ///

19.3.1 Putting Things Together

If we want to eliminate all erasing and chain rules from a grammar, we first eliminate the erasing rules, and then eliminate the chain rules.

```
Algorithm 25: Eliminate Chain And Erasing Rules
Input: a CFG $G = (\Sigma, V, P, S)$
Output: a CFG $G'$ such that $L(G') = L(G) - \{\lambda\}$ and $G'$ has no $\lambda$-rules and
        no chain rules
let $G'$ be the result of Eliminate Erasing on $G$;
let $G''$ be the result of Eliminate Chain on $G'$;
return $G''$
```

19.16 Theorem (Correctness of Eliminate Chain And Erasing Rules)**.** *Algorithm 25 computes, from grammar G, a grammar G' such that $L(G') = L(G) - \{\lambda\}$, G' has no chain rules, and G' has no erasing rules.*

Once again, we have to be careful what order to do things in. See Exercise 154.

Summarizing, we have proved the following.

19.17 Theorem. *Let G be a context-free grammar. There is a context-free grammar G' such that*

- *$L(G') = L(G)$, and*
- *G has no chain rules or λ-rules, with the possible exception of a rule $S \to \lambda$ where S is the start symbol of G'.*

Furthermore, there is an algorithm which given G will return the corresponding G'.

19.18 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.

19.19 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

$$\begin{aligned}
S &\to aBb \mid ab \mid aES \mid aS\\
B &\to PE \mid P \mid E \mid aBb \mid BB \mid B \mid ab\\
P &\to c \mid E\\
E &\to B \mid e \mid E
\end{aligned}$$

It should be noted that above we have not bothered to remove rules such as $B \to B$ which our algorithm adds but which clearly are not necessary; these will go away when we remove chain rules next.

Now we want eliminate chain rules. When add all the rules first required by our Algorithm, we get to:

$$\begin{aligned}
S &\to aBb \mid ab \mid aES \mid aS\\
B &\to PE \mid P \mid E \mid aBb \mid BB \mid B \mid ab \mid c \mid E\\
P &\to c \mid e \mid B \mid E\\
E &\to PE \mid P \mid B \mid e \mid P \mid E \mid aBb \mid BB \mid ab \mid c
\end{aligned}$$

If we then remove chain rules we arrive at

$$\begin{aligned}
S &\to aBb \mid ab \mid aES \mid aS\\
B &\to PE \mid aBb \mid BB \mid ab \mid c \mid e\\
P &\to c \mid e\\
E &\to PE \mid e \mid aBb \mid BB \mid ab \mid c
\end{aligned}$$

19.4 An Upper Bound on Derivation Lengths

There is a significant virtue in having no chain or erasing rules: we get an easy upper bound on the length of derivations in the grammar. This means that we have a very simple *algorithm* for the problem of deciding whether a given word is derivable. The simple algorithm isn't efficient enough to be useful in practice, but it is the precursor of tractable algorithms.

19.20 Theorem. *Suppose G is a grammar with no chain rules nor erasing rules. Let x be in $L(G)$. Then every G-derivation of w has no more than $2|x|-1$ steps.*

Proof. If $S \overset{*}{\Longrightarrow} \alpha_1 \overset{*}{\Longrightarrow} \dots \overset{*}{\Longrightarrow} \alpha_k = x$ 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. ///

19.5 Chomsky Normal Form

Chomsky Normal Form is useful theoretically, and as a prelude to the Cocke-Kasami-Younger algorithm we will see in Section 21.

19.21 Definition. *A CFG G is in* Chomsky Normal Form *if every rule 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.)*

19.22 Theorem. *For every CFG G there exists a CFG G_C such that*

1. $L(G_C) = L(G) - \{\lambda\}$
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 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 19.7 we may build a G' such that $L(G') = L(G) - \{\lambda\}$ and G' has no chain rules or λ-rules. 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. ///

19.23 Example. This little grammar obviously generates $\{a^i b^i \mid i \geq 1\}$ Note that we've left out λ here.

$$S \rightarrow ASb \mid ab$$

An equivalent Chomsky Normal Form grammar is

$$\begin{aligned}
S &\rightarrow AT|AB \\
T &\rightarrow XB \\
X &\rightarrow AT \mid AB \\
A &\rightarrow a \\
B &\rightarrow b
\end{aligned}$$

19.6 $NFA_\lambda s$ Revisited

As a final remark, we connect the notion 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 character, has the capacity to transition from one state to another without consuming any input at all. We denote such transitions, naturally, as $p \rightarrow q$, 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 are *exactly the same thing as 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 16.2. Then eliminating λ-transitions in M and eliminating chain rules from G_M are *exactly the same algorithm* just expressed in different notation. The context-free grammar setting is more general, since not every CFG corresponds to an NFA_λ, but algorithm for removing λ-transitions from an NFA_λ is just the special case of Algorithm 22 when the input grammar is regular.

We just observe that whenever we add productions 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.

19.24 Check Your Reading. *Make sure this last remark is clear to you, by drawing some pictures of $NFA_\lambda s$ and then writing down the corresponding regular grammars with chain rules; then do some examples the other way around.*

By the way, it is superficially tempting to think that erasing rules in a grammar should have something to do with λ-transitions in an automaton. Resist that temptation! It is chain rules, not erasing rules, that are related to λ-transitions.

19.7 Exercises

***Exercise* 152.** Find a grammar generating the same language as the one below, with no useless rules.

$$\begin{aligned} S &\to dS \mid A \mid C \\ A &\to aA \mid a \\ B &\to bB \mid b \\ C &\to cC \end{aligned}$$

***Exercise* 153.** Suppose we first eliminate non-reachable variables from a grammar and then eliminate non-generating variables, Show by means of an example that this does not always yield a grammar with no useless rules.

***Exercise* 154.** Suppose we first eliminated the chain rules from a grammar and then eliminated the erasing rules. Show by means of an example that this does not always yield a grammar with no chain or erasing rules.

***Exercise* 155.** For each grammar give an equivalent grammar with no λ- or chain rules.

G_1 :

$$\begin{aligned} S &\to aSbb \mid ST \mid c \\ T &\to bTaa \mid S \mid \lambda \end{aligned}$$

G_2 :

$$\begin{aligned} S &\to aA \mid bB \mid A \\ A &\to aaA \mid B \\ B &\to cc \mid A \end{aligned}$$

G_3 :

$$\begin{aligned} S &\to aBbB \\ B &\to P \mid bBb \\ P &\to c \mid E \\ E &\to e \mid \lambda \end{aligned}$$

***Exercise* 156.** For each grammar G give a grammar generating $L(G)$ with no useless rules. Then eliminate chain and erasing rules.

G_1 :

$$\begin{aligned} S &\to AB \mid CA \\ A &\to a \\ B &\to BC \mid AB \\ C &\to aB \mid b \end{aligned}$$

G_2 :

$$\begin{aligned} S &\to ASB \mid \lambda \\ A &\to aAS \mid a \\ B &\to SbS \mid A \mid bb \end{aligned}$$

***Exercise* 157.** 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? (Big-Oh notation is fine.)

***Exercise* 158.** Build Chomsky normal form grammars equivalent to the grammars below.

G_1 :

$$S \to aSa \mid bSb \mid a \mid b$$

G_2 :

$$\begin{aligned} S &\to XY \\ X &\to aXbb \mid abb \\ Y &\to cY \mid c \end{aligned}$$

G_3 :

$$\begin{aligned} S &\to aSa \mid aBa \\ B &\to Bb \mid b \end{aligned}$$

G_4 :

$$\begin{aligned} S &\to ASc \mid AScc \mid ABc \mid ABcc \\ A &\to Aa \mid a \\ B &\to bB \mid b \end{aligned}$$

G_5 :

$$\begin{aligned} S &\to A \mid aAbB \mid ABC \mid a \\ A &\to aA \mid a \\ B &\to bBcC \mid b \\ C &\to abc \end{aligned}$$

***Exercise* 159.** *Building Chomsky Normal Form (from [Koz97])* 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. $\{a^k b^m c^n \mid k, m, n \geq 1, 2k \geq n\}$

***Exercise* 160.** Suppose G is a grammar in Chomsky Normal Form. Let x be in $L(G)$. Then every G-derivation of w has exactly $2|x| - 1$ steps.

20 Pushdown Automata and Parsing

We have a “machine model” for regular languages, namely finite automata. That is, when M is a *DFA* accepting regular language L, M accepts strings x as input and computes a decision as to whether $x \in L$. In this section 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 $x \in L(G)$?

Now, this membership problem 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, namely: given a string x, decide whether $x \in L$ and if so, compute a parse tree for x, together with some semantic actions (such as code generation in a compiler), and if $x \notin L$ return some useful error message to the user about why $x \notin L$.

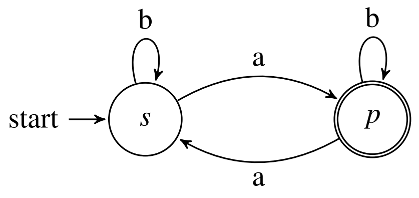
It is not so straightforward to add (certain) semantic actions, or to generate good error messages; that is more advanced work. But what we do here is the essential first step.

20.1 Warm Up: *DFAs* Are Parsers for Regular Grammars

Suppose the *CFG* G we start with is regular (Section 16.2). For instance, take G to be

$$\begin{aligned} S &\to aP \mid bS \\ P &\to aS \mid bP \mid \lambda \end{aligned}$$

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 character; if the char is an a calls procedure P, else if the character is b is calls itself; else if the char is end-of-file the program rejects.
- procedure S reads a character, and if the char is an a calls procedure S, else if the character is b is 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 characters, 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.[12]

Even a very simple example shows us what the issue is in the non-regular case. Look at this grammar

$$\begin{aligned} S &\to aAa \mid \lambda \\ A &\to bSb \end{aligned}$$

If we think of S as being a procedure in a standard program, it does this (the story for procedure A is similar).

[12]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...

1. reads a character (or returns successfully on end-of-file)
2. makes a call to procedure A
3. when that call returns, reads another character 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 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.

20.2 Pushdown Automata

We start with an 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.[13] Careful details are below.

Moves

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.

[13] Actually, there are two notions of acceptance typically considered for *PDAs*: acceptance *by final state* and acceptance *by empty stack*. They 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 wil only consider *PDAs* that accept by empty stack.

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

$$(\,(p,B),\ (q,\beta)\,)$$

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 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.

To complete the formal definition of a *PDA* we must designate a start state s of the machine and some accepting states. It is also convenient to postulate a special initial stack symbol $\perp$. Putting this all together we have the following

20.1 Definition. *A pushdown automaton M is a tuple* $(\Sigma, Q, s, \Gamma, \perp, \delta)$, *where*

- Σ *is a finite* input alphabet,
- Q *is a finite set of* states,
- $s \in Q$ *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*

$$(\,(p,a,B),\ (q,\beta)\,) \qquad \textit{or}$$
$$(\,(p,B),\ (q,\beta)\,)$$

  where $p,q \in Q,\ Z \in \Gamma,\ \beta \in \Gamma^*$

We allow that the terminal alphabet Σ can 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

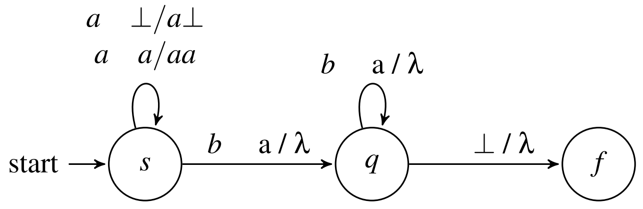
Just as with *DFAs* and *NFAs*, it can be helpful to draw pictures of *PDAs*. We just label the arcs with (i) the input symbol being read, if any, and (ii) the action on the stack. See the example, next.

20.2 Example. Here is the raw definition of a *PDA*, *P*, designed to accept the language $\{a^n b^n \mid n \geq 1\}$. The intuitive idea is that *P* reads *a*s, storing them on the stack until it starts to see *b*s; while it reads *b*s it pops *a*s off the stack; if the input string is exhausted precisely when the stack has no more *a*s, 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

$$\begin{gathered}
((s,a,\perp),(s,a\perp))\\
((s,a,a),(s,aa))\\
((s,b,a),(q,\lambda))\\
((q,b,a),(q,\lambda))\\
((q,\perp),(f,\lambda))
\end{gathered}$$

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.

Such an example is helpful in building intuition, but *PDAs* are tricky enough that we need to be perfectly rigorous in defining what are computation is and what it means to accept a string. We do that now.

Computation

The action of M is described in terms of transitions from one such machine configuration to another.

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:

20.3 Definition (*PDA* Configuration)**.** *Let M be a PDA. A* configuration *of M is a triple*

$$[\, p,\ w,\ \gamma\,] \quad \textit{with} \quad p \in Q, \quad w \in \Sigma^*, \quad \gamma \in \Gamma^*.$$

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.

Now, a *computation* is precisely this: a sequence of configurations generated according to the δ relation.

20.4 Definition (*PDA* Computation)**.** *The one-step transition relation $\vdash$ is defined as follows. If*

$$((p,a,B),\ (q,\beta))$$

is a move in δ then for any x and any γ

$$[\,p,\ ax,\ B\gamma\,] \vdash [\,q,\ x,\ \beta\gamma\,].$$

If

$$((p,B),\ (q,\beta))$$

is a move in δ then for any x and any γ

$$[\,p,\ x,\ B\gamma\,] \vdash [\,q,\ x,\ \beta\gamma\,].$$

We deine the relation $\overset{*}{\vdash}$ to be the reflexive transitive closure of $\vdash$. This means that $C \overset{*}{\vdash} D$ if D follows from C by a finite number (zero or more) steps of $\vdash$.

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 $C \vdash C'$. By the same token there may no transitions available, that is no C' with $C \vdash C'$. 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 $\vdash$ 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 $\overset{*}{\vdash}$ is the set of *computations.*

Acceptance

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. Formally we have the following definition.

20.5 Definition (*PDA* Acceptance)**.** *Let M be a PDA with start state s, initial stack symbol $\perp$, and transition relation $\overset{*}{\vdash}$. A string x is* accepted *by M if for some state f*

$$[\,s,\ x,\ \perp\,] \overset{*}{\vdash} [\,f,\ \lambda,\ \lambda\,].$$

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.

20.6 Example (continued)**.** We continue Example 20.2 by showing some computations.

On input *aab*:

$$\begin{aligned}[\,s,\ aab,\ \perp\,] &\vdash [\,s,\ ab,\ a\perp\,]\\ &\vdash [\,s,\ b,\ aa\perp\,]\\ &\vdash [\,q,\ \lambda,\ a\perp\,]\end{aligned}$$

and now the machine blocks, since there are no δ-moves defined out of q when the stack top is a. 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*:

$$\begin{aligned}[\,s,\ abb,\ \perp\,] &\vdash [\,s,\ bb,\ a\perp\,]\\ &\vdash [\,q,\ b,\ \perp\,]\\ &\vdash [\,f,\ b,\ \lambda\,]\end{aligned}$$

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 *aab*:

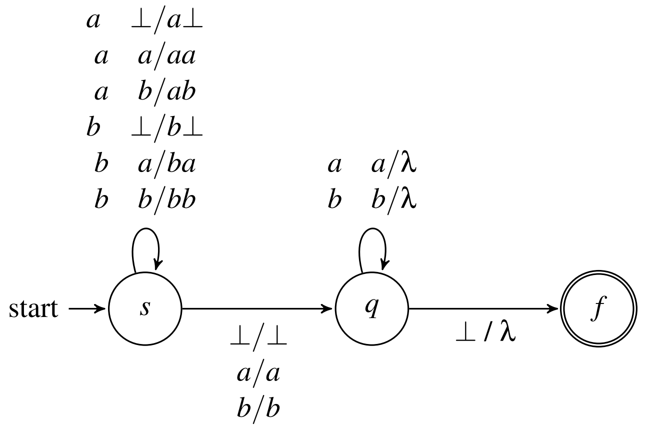
$$\begin{aligned}[\,s,\ aabb,\ \perp\,] &\vdash [\,s,\ abb,\ a\perp\,]\\ &\vdash [\,s,\ bb,\ aa\perp\,]\\ &\vdash [\,q,\ b,\ a\perp\,]\\ &\vdash [\,q,\ \lambda,\ \perp\,]\\ &\vdash [\,f,\ \lambda,\ \lambda\,]\end{aligned}$$

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.

20.7 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 $\{a^n b^n \mid n \geq 0\}$?

20.8 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 20.2 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 *a*s or *b*s 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*

$$
\begin{aligned}
[\,s,\ abba,\ \perp\,] &\vdash [\,s,\ bba,\ a\perp\,] \\
&\vdash [\,s,\ ba,\ ba\perp\,] && \text{the guess happens next} \\
&\vdash [\,q,\ ba,\ ba\perp\,] \\
&\vdash [\,q,\ a,\ a\perp\,] \\
&\vdash [\,q,\ \lambda,\ \perp\,] \\
&\vdash [\,f,\ \lambda,\ \lambda\,]
\end{aligned}
$$

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.

$$\begin{aligned}
[\,s,\ abba,\ \perp\,] &\vdash [\,s,\ bba,\ a\perp\,] \\
&\vdash [\,s,\ ba,\ ba\perp\,] \\
&\vdash [\,s,\ a,\ bba\perp\,] \qquad \text{ops, waited too long to guess} \\
&\vdash [\,q,\ a,\ bba\perp\,]
\end{aligned}$$

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.

20.9 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.*

Exercise 161 asks you to modify this *PDA* so that it accepts even-length palindromes *and* odd-length palindromes.

20.3 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 q, input symbol c and stack symbol Z, one of the following holds, but not both:

- there is exactly one (p,β) such that $((p,c,X),(p,\beta)) \in \delta)$
- there is exactly one (p,β) such that $((p,X),(p,\beta)) \in \delta)$

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 have already seen an example: the *PDA* in Example 20.8 cannot be converted to a deterministic *PDA*. In other words, there is no deterministic *PDA* D such that $L(D) = \{ww^R \mid w \in \{a,b\}^*\}$.

20.4 *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.

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.

20.10 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.*

Here is the construction that is the basis for the first part of the theorem. Amazingly, we can always build the *PDA* we want with only one state.

20.11 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 s, which is of course the start state*
- *input alphabet Σ*
- *stack alphabet $N \cup \Sigma$,*
- *the initial stack symbol is the start symbol S from G*
- *the move relation δ is as follows:*

1. *for each rule $A \to \alpha$ from P, $(\ (s,A),\ (s,\alpha)\ )$ is a move in δ.*
2. *for each symbol $a \in \Sigma$, $(\ (s,a,a),\ (s,\lambda)\ )$ 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 c, that's good, we scan past c in the input and pop c off the stack.
- If the top of the stack is a terminal symbol different from the current input symbol c, 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.

Let's see how this works in an example.

20.12 Example. Let G be the following grammar (for the language $\{a^n b^n \mid n \geq 1\}$):

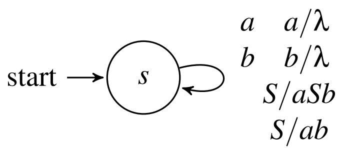
$$S \to aSb \ \mid \ ab$$

The *PDA M* we build has these four δ moves

$$(\ (s,a,a),\ (s,\lambda)\ ) \qquad (\ (s,S),\ (s,aSb)\ )$$

$$(\ (s,b,b),\ (s,\lambda)\ ) \qquad (\ (s,S),\ (s,ab)\ )$$

A picture:



Here is a run of M on input $aabb$.

$$\begin{aligned}
[\,s,\ aabb,\ S\,] &\vdash [\,s,\ aabb,\ aSb\,] && (1)\\
&\vdash [\,s,\ abb,\ Sb\,] && (2)\\
&\vdash [\,s,\ abb,\ abb\,] && (3)\\
&\vdash [\,s,\ bb,\ bb\,] && (4)\\
&\vdash [\,s,\ b,\ b\,] && (5)\\
&\vdash [\,s,\ \lambda,\ \lambda\,] && (6)
\end{aligned}$$

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:

$$S \Longrightarrow aSb \Longrightarrow aabb$$

20.13 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 labelling each of the transitions above with the number of a move taken by the PDA.*

20.4.1 Perspective

Recall Example 20.2. 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 20.11.

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 languges. In our current setting, *PDAs* generated by Definition 20.11 will not take advantage of human insights about specific grammars, which might, for example, eliminate non-determinism.

The key virtue of Definition 20.11 is that it shows how to build a *PDA* for *any CFG*, even if it is too complex for a human to understand it intuitively, with a guarantee that the *PDA* will be correct.

20.14 Example. Suppose we start with the following grammar

$$E \rightarrow E+E \mid E*E \mid 0 \mid 1$$

We get the *PDA* $(\Sigma, Q, s, \Gamma, E, \delta,)$, where δ consists of the following moves

- $((s,E),\ (s,E+E))$
- $((s,E),\ (s,E*E))$
- $((s,E),\ (s,0))$
- $((s,E),\ (s,1))$
- $((s,+,+),\ (s,\lambda))$
- $((s,*,*),\ (s,\lambda))$
- $((s,0,0),\ (s,\lambda))$
- $((s,1,1),\ (s,\lambda))$

Here is the trace of a computation accepting the input string $w = 1+0*1$.

$$\begin{aligned}
[\,s,\ 1+0*1,\ E\,] &\vdash [\,s,\ 1+0*1,\ E+E\,] && (1)\\
&\vdash [\,s,\ 1+0*1,\ 1+E\,] && (2)\\
&\vdash [\,s,\ +0*1,\ +E\,] && (3)\\
&\vdash [\,s,\ 0*1,\ E\,] && (4)\\
&\vdash [\,s,\ 0*1,\ E*E\,] && (5)\\
&\vdash [\,s,\ 0*1,\ 0*E\,] && (6)\\
&\vdash [\,s,\ *1,\ *E\,] && (7)\\
&\vdash [\,s,\ 1,\ E\,] && (8)\\
&\vdash [\,s,\ 1,\ 1\,] && (9)\\
&\vdash [\,s,\ \lambda,\ \lambda\,] && (10)
\end{aligned}$$

This *PDA* computation corresponds naturally to the following derivation

$$E \Longrightarrow E+E \Longrightarrow 1+E \Longrightarrow 1+E*E \Longrightarrow 1+0*E \Longrightarrow 1+0*1$$

20.15 Check Your Reading. *Label each of the transitions above with the number of a move taken by the PDA.*

Returning to our example, here is the trace of another computation on the same input string $w = 1+0*1$, but this time the *PDA* computation corresponds to the following derivation

$$E \Longrightarrow E*E \Longrightarrow E+E*E \Longrightarrow 1+E*E \Longrightarrow 1+0*E \Longrightarrow 1+0*1$$

$$
\begin{aligned}
[\,s,\ 1+0*1,\ E\,] &\vdash [\,s,\ 1+0*1,\ E*E\,] && (1)\\
&\vdash [\,s,\ 1+0*1,\ E+E*E\,] && (2)\\
&\vdash [\,s,\ 1+0*1,\ 1+E*E\,] && (3)\\
&\vdash [\,s,\ +0*1,\ +E*E\,] && (4)\\
&\vdash [\,s,\ 0*1,\ E*E\,] && (5)\\
&\vdash [\,s,\ 0*1,\ 0*E\,] && (6)\\
&\vdash [\,s,\ *1,\ *E\,] && (7)\\
&\vdash [\,s,\ 1,\ E\,] && (8)\\
&\vdash [\,s,\ 1,\ 1\,] && (9)\\
&\vdash [\,s,\ \lambda,\ \lambda\,] && (10)
\end{aligned}
$$

20.16 Check Your Reading. *Label each of the transitions above with the number of a move taken by the PDA.*

20.5 *PDAs* and Parsing Algorithms

If G is a *CFG* and M is a *PDA* accepting $L(G)$ then, as we described at the beginning of this section, we can view M as a solution to the membership problem for G. And we can see M as an abstract representation of a parsing algorithm for G, if we tweak it just a bit to construct a parse tree during its computation. (This isn't hard.)

But to be fair, in the absolutely 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.

Now, by our work on refactoring grammars we can massage our grammar to have no λ- or chain-rules, which will in turn give an upper bound on the number of *PDA* steps we need to simulate (check this for yourself). But let's try to do better.

Before reading this section I suggest doing Exercise 162, parts 3 and 4.

20.17 Theorem (Greibach Normal Form)**.** *Let G be a context-free grammar. There is a context-free grammar G' such that*

- *$L(G') = L(G)$, and*
- *each rule of G' has the form*

  $$A \rightarrow aB_1 \dots B_k \quad k \geq 0$$

  for some terminal a and variables $B_1, \dots B_k$, with the possible exception of a rule $S \rightarrow \lambda$ where S is the start symbol of G'.

Furthermore, there is an algorithm which given G will return the corresponding G'.

Proof. Omitted here. ///

Now suppose G is a grammar in Greibach Normal Form and let M be a *PDA* constructed from G as in the proof of Theorem 20.10. Recall the moves of the *PDA*:

1. For each rule $A \rightarrow \alpha$ from P, $(\,(s,\lambda,A),\ (s,\alpha)\,)$ is a move in δ.
2. For each input symbol $a \in \Sigma$, $(\,(s,a,a),\ (s,\lambda)\,)$ 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 $A \rightarrow aB_1 \dots B_k$.

At this point it is clear that, 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. So provided we use the current input symbol as our guide for which rules to use in the next type-(i) rule we might as well just push the $B_1 \dots B_k$ part of the right-hand side $aB_1 \dots B_k$ and scan past the input symbol a in one step. Since each *PDA* move will scan past an input symbol, the machine will take a number of steps equal to the length of the input string.

Note that such a *PDA* will not be one as constructed in the proof of Theorem 20.10, since those *PDAs* push the entire right-hand-sides of rules onto the stack, whereas the *PDAs* we are talking about now will never have terminal symbols on the stack.

Deterministic Grammars

Now finally suppose that we are lucky enough that our grammar satisfies the following property:

> *Property Q:* For each variable A there are no two rules $A \rightarrow aB_1 \dots B_k$ and $A \rightarrow aC_1 \dots C_p$ whose right-hand sides start with the same terminal.

In this case the choice of type-(i) move is completely determined. 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.

20.6 Exercises

***Exercise* 161.** 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 20.11, modify the *PDA* in Example 20.8.
- Do some sample computations on each of your answers, over some strings in the language and not in the language, to gain confidence in your construction.

Hint. The even-length palindromes were written as $\{ww^R \mid w \in \{a,b\}^*\}$ in Example 20.8. 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 20.8. Just modify the *PDA* in Example 20.8. You don't even have to add any states.

***Exercise* 162.** For each of the following *CFGs G*,

- Construct a *PDA* M such that $L(M) = L(G)$, using the construction in Definition 20.11.
- Do some sample computations on each of your answers, over some strings in the language and not in the language, to gain confidence in your construction.
- For each accepting *PDA* computation, give a corresponding grammar derivation

1. G_1 is

   $$S \to aSb \mid \lambda$$

   Hint. Just modify the *PDA* in Example 20.12
2. G_2 is

   $$E \to E + E \mid F$$
   $$F \to F * F \mid 0 \mid 1$$

   Does your *PDA* have more than one accepting computation on 1+0*1? Compare to Example 20.14

3. G_3 is

$$\begin{aligned} S &\to aS \mid bP \\ P &\to aP \mid bS \mid a \end{aligned}$$

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 20.11 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.

4. G_4 is

$$E \to +EE \mid *EE \mid 0 \mid 1$$

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 G_4 is unambiguous.

***Exercise* 163.** Let L be

$$\{a^i b^i c^i \mid i \geq 0\}$$

Show that $\overline{L}$ is context-free.

***Exercise* 164.** The language $D = \{w \mid w$ can be written as xx for some string $x\}$ is not context-free (over the alphabet $\Sigma = \{a,b\}$).

Show that $\overline{D}$, the complement of D, is context-free.

As a hint, note that $\overline{D}$ can be written as

$$\begin{aligned} &\{w \mid \text{ the length of } w \text{ is odd}\} \\ \cup &\{xayx'by' \mid x,y,x',y' \in \Sigma^*,\ |x| = |x'|,\ |y| = |y'|\} \\ \cup &\{xbyx'ay' \mid x,y,x',y' \in \Sigma^*,\ |x| = |x'|,\ |y| = |y'|\} \end{aligned}$$

Try building a *PDA* accepting this language.

***Exercise* 165.** *CFL and regular closure* 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. $L_1 - L_2$

Hint. Remember the intersection of a regular language and a context-free language is guaranteed to be context-free.

***Exercise* 166.** 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 $R \cap N = \emptyset$ then X is not context-free.

21 Context-Free Membership and the CKY Algorithm

The problem of *parsing* is: given a context-free grammar G and a string x, decide whether or not x is in $L(G)$ and if so, construct a parse tree for x. For simplicity we will concentrate on the pure decision problem:

> *CFG Membership*
>
> INPUT: a context-free grammar G and a string x of terminals.
>
> QUESTION: is $x \in L(G)$?

The method we will ultimately settle on for deciding this problem can be refined to produce a parse tree in case of a positive answer.

21.1 Example.

$$\begin{aligned} S &\to aS \mid bA \mid bS \\ A &\to aB \mid bB \\ B &\to a \mid b \end{aligned}$$

Is *abb* derivable? How about *bba*? do string abb (no), then bba (yes)

Is there a natural relation between length of string and length of a derivation?

21.2 Example. Consider:

$$\begin{aligned} S &\to AA \\ A &\to BB \\ B &\to b \mid \lambda \end{aligned}$$

Then in fact, λ is generated, but takes 7 steps! (Try it)

21.3 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.

$$\begin{aligned} S &\to AB \mid AC \mid SS \\ C &\to SB \\ A &\to l \\ B &\to r \end{aligned}$$

Is this string derivable?

$$lrlrllrlrrllrr$$

21.1 A Bounded Exhaustive Search

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.

$$S \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow ??? \Rightarrow x\ ??$$

But this is not yet a correct solution. It would be fine if we had a computable upper bound on the number of steps in the derivations we had to search through, since there are only finitely many derivation sequences of a given length, and we can effectively generate them. But for an arbitrary CFG there is no nice relationship between the number of steps in a derivation and the length of the string derived. The problem is λ-rules and chain-rules.

21.4 Theorem. *The membership problem for CFGs is decidable.*

Proof. Theorem 19.20 gives an upper bound on how long a derivation could be in a grammar with no chain or erasing rules. So, let G be a grammar and x a string fo which we want to decide "$x \in L(G)$?".

In the special cas where x is λ, use Algorithm 23 to determine whether the start symbol of G is nullable. This tells us whether $nil \in L(G)$

When $x \neq \lambda$, convert G to a grammar G' with no chain rules or erasing rules, and $L(G') = L(G) - \{\lambda\}$. So $x \in L(G)$ if and only if $x \in L(G')$.

We can decide whether $x \in L(G')$ by examining all of the finitely many G'-derivations of length no more than $2|x| - 1$. ///

This is fine as a decidability result, but it obviously does not give an algorithm that one wants to use in practice. We can do better: there is a famous algorithm based on dynamic programming called the *Cocke-Kasami-Younger* algorithm, which solves the membership problem in time $O(|x|^3)$. This is still not fast enough for most applications.

The parsing algorithms that are used in practice can be usefully viewed as refinements of a *machine model* for context-free languages, pushdown automata, to which we now turn.

21.5 Lemma. Suppose *our grammar does NOT have any erasing-rules or chain-rule. Then whenever* $S \overset{*}{\Longrightarrow} w$, *for any string* $w \in \Sigma^*$, *the number of steps in the derivation is at most* $2|w| - 1$.

Proof. By hypothesis, every rule in the grammar looks like (i) $A \to \gamma$ with γ of length at least 2, or (ii) looks like $A \to c$ for some terminal symbol c. So if $\alpha \Rightarrow \alpha'$ in a derivation, either (i) the length of α is greater than the length of α', or (ii) α has one more terminal symbol than does α' (or both things can be true).

Now suppose $S \overset{*}{\Longrightarrow} w$. How many derivation steps of type (i) can there be? At most $|w|-1$, since the total length from S to w increases by $|w|-1$. How many derivation steps of type (ii) can there be? At most $|w|$, since there are $|w|$ terminals to be generated. So the maximum number of steps is at most $2|w|-1$. ///

For such a grammar, an exhaustive search can solve the membership problem.

Without that assumption it is unclear how to put a bound on derivation length (in the presence of chain rules we might actually have *loops* in our derivations.)

So we'd like to eliminate λ- and chain-rules if we can.

Algorithm 26: A Naive Algorithm for *CFG* Membership

```
Input: a CFG $G = (\Sigma, V, P, S)$ and a string $w \in \Sigma^*$
Decides whether $w \in L(G)$;
if $w = \lambda$ then
    use Algorithm 23 to see the start symbol is nullable
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
```

21.1.1 Can We Do Better?

It's easy to see that an exhaustive search through the space of derivations 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 polynomial-time (cubic time in fact) algorithm that works for an arbitrary context-free grammar. This is presented in Section 21.2.

2. For most grammars that arise in practice 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 . . . we refer you to any good book about compilers.

21.2 The Cocke-Younger-Kasami algorithm

A dynamic programming algorithm for CFG membership, running in $O(n^3)$ time.

Details to be written. . .

21.3 Exercises

***Exercise* 167.** *Using the CYK algorithm I* Consider lots of CNF grammars G and strings x and run through the Cocke-Younger-Kasami algorithm to determine whether the string x is in $L(G)$.

Example. Let G be the following grammar.

$$\begin{aligned} S &\to AB \\ A &\to a \\ B &\to AB \mid b \end{aligned}$$

Use the Cocke-Younger-Kasami algorithm to determine whether the string aab is in $L(G)$. (It is.) Then make a parse tree deriving aab. Is there more than one? Then make a leftmost derivation corresponding to your parse tree(s).

***Exercise* 168.** Here is the CNF grammar generating $\{a^ib^i \mid i \geq 1\}$ from Section 19.23.

$$\begin{aligned} S &\to AT|AB \\ T &\to XB \\ X &\to AT \mid AB \\ A &\to a \\ B &\to b \end{aligned}$$

Run the CYK algorithm for this grammar on input string $aaabbb$. Then make a parse tree deriving $aaabbb$. Is there more than one? Then make a leftmost derivation corresponding to your parse tree(s).

***Exercise* 169.** *Extending the CYK algorithm I* Modify the Cocke-Younger-Kasami algorithm to count the number of parse trees of a given string.

***Exercise* 170.** *Extending the CYK algorithm II* Suppose that to each rule in a Chomsky Normal Form grammar G we associate a cost, and say that the cost of a derivation of a string w is the sum of the costs of the individual rule steps.

Give an algorithm for finding a minimum-cost derivation of a string w in such a grammar. What is the complexity of your algorithm?

Hint. Modify the CYK algorithm.

22 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.

First we do a concrete example, then show the general result.

22.1 A Language Which is Not Context-Free

One can prove a very general Pumping Lemma —similar to the one for regular languages— for context-free languages, which can be applied to show lots of languages not to be context-free. In this note we will be content to show non-context-freeness of a particular language, namely $L = \{a^n b^n c^n \mid n > 0\}$. The work we do for this particular result is almost all we need to prove the general Pumping Lemma, so once you understand this note you should be able to proceed to the general result if you care to.

First, a purely combinatorial lemma.

22.1 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. ///

22.2 Proposition. *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 22.1 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. ///

22.3 Proposition. *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 Proposition 22.2 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

$$\begin{aligned} S &\overset{*}{\Longrightarrow} u\,A\,\sigma \\ &\overset{*}{\Longrightarrow} uz_1 \\ &\equiv z \end{aligned}$$

Here σ is a mix of terminals and variables, and $A\sigma \overset{*}{\Longrightarrow} z_1$.

Since that A is repeated on the path we are thinking about, our leftmost derivation must look like

$$\begin{aligned}
S &\overset{*}{\Longrightarrow} u\,A\,\sigma \\
&\overset{*}{\Longrightarrow} u\,v\,A\,\alpha\,\sigma && \text{via } A \overset{*}{\Longrightarrow} v\,A\alpha \\
&\overset{*}{\Longrightarrow} u\,v\,w\,\alpha\,\sigma && \text{via } A \overset{*}{\Longrightarrow} w \\
&\overset{*}{\Longrightarrow} u\,v\,w\,x\,\sigma && \text{via } \alpha \overset{*}{\Longrightarrow} x \\
&\overset{*}{\Longrightarrow} u\,v\,w\,x\,y && \text{via } \sigma \overset{*}{\Longrightarrow} y \\
&\equiv z
\end{aligned}$$

Now observe that the following, completely different, derivation is also a legal derivation in G

$$\begin{aligned}
S &\overset{*}{\Longrightarrow} u\,A\,\sigma \\
&\overset{*}{\Longrightarrow} u\,v\,A\,\alpha\,\sigma && \text{via } A \overset{*}{\Longrightarrow} v\,A\,\alpha \\
&\overset{*}{\Longrightarrow} u\,v\,v\,A\,\alpha\,\alpha\,\sigma && \text{via } A \overset{*}{\Longrightarrow} v\,A\,\alpha \\
&\overset{*}{\Longrightarrow} u\,v\,v\,w\,\alpha\,\alpha\,\sigma && \text{via } A \overset{*}{\Longrightarrow} w \\
&\overset{*}{\Longrightarrow} u\,v\,v\,w\,x\,x\,\sigma && \text{via } \alpha \overset{*}{\Longrightarrow} x \\
&\overset{*}{\Longrightarrow} u\,v\,v\,w\,x\,x\,y && \text{via } \sigma \overset{*}{\Longrightarrow} y
\end{aligned}$$

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 $A \overset{*}{\Longrightarrow} vA\alpha$ and $\alpha \overset{*}{\Longrightarrow} x$ above, we know that $A \overset{*}{\Longrightarrow} vAx$ 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 Proposition. ///

Next we prove a general Pumping Lemma for context-free languages.

22.2 A Pumping Lemma for Context-Free Grammars

22.4 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):*

$$\begin{aligned} S &\overset{*}{\Longrightarrow} uAy \\ &\overset{*}{\Longrightarrow} uvAxy && \text{// via} \quad A \overset{*}{\Longrightarrow} vAx \\ &\overset{*}{\Longrightarrow} uvwxy && \text{// via} \quad A \overset{*}{\Longrightarrow} w \\ &= z \end{aligned}$$

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

$$\begin{aligned} S &\overset{*}{\Longrightarrow} u\,A\,\sigma \\ &\overset{*}{\Longrightarrow} uz_1 \\ &\equiv z \end{aligned}$$

Here σ is a mix of terminals and variables, and $A\sigma \overset{*}{\Longrightarrow} z_1$.

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

$$\begin{aligned}
S &\overset{*}{\Longrightarrow} u\,A\,\sigma && \\
&\overset{*}{\Longrightarrow} u\,v\,A\,\alpha\,\sigma && \text{via } A \overset{*}{\Longrightarrow} v\,A\alpha \\
&\overset{*}{\Longrightarrow} u\,v\,w\,\alpha\,\sigma && \text{via } A \overset{*}{\Longrightarrow} w \\
&\overset{*}{\Longrightarrow} u\,v\,w\,x\,\sigma && \text{via } \alpha \overset{*}{\Longrightarrow} x \\
&\overset{*}{\Longrightarrow} u\,v\,w\,x\,y && \text{via } \sigma \overset{*}{\Longrightarrow} y \\
&\equiv z &&
\end{aligned}$$

Now observe that the following, completely different, derivation is also a legal derivation in G

$$\begin{aligned}
S &\overset{*}{\Longrightarrow} u\,A\,\sigma && \\
&\overset{*}{\Longrightarrow} u\,v\,A\,\alpha\,\sigma && \text{via } A \overset{*}{\Longrightarrow} v\,A\,\alpha \\
&\overset{*}{\Longrightarrow} u\,v\,v\,A\,\alpha\,\alpha\,\sigma && \text{via } A \overset{*}{\Longrightarrow} v\,A\,\alpha \\
&\overset{*}{\Longrightarrow} u\,v\,v\,w\,\alpha\,\alpha\,\sigma && \text{via } A \overset{*}{\Longrightarrow} w \\
&\overset{*}{\Longrightarrow} u\,v\,v\,w\,x\,x\,\sigma && \text{via } \alpha \overset{*}{\Longrightarrow} x \\
&\overset{*}{\Longrightarrow} u\,v\,v\,w\,x\,x\,y && \text{via } \sigma \overset{*}{\Longrightarrow} y
\end{aligned}$$

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 $A \overset{*}{\Longrightarrow} vA\alpha$ and $\alpha \overset{*}{\Longrightarrow} x$ above, we know that $A \overset{*}{\Longrightarrow} vAx$ 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 Proposition. ///

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 22.1. 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.

22.5 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 refcfg-pumping to show languages not context-free.

[@@ to be continued]

22.3 Exercises

***Exercise* 171.** *CFL intersection* Show that the context-free languages are *not* closed under intersection.

***Exercise* 172.** *CFL complement* Show that the context-free languages are *not* closed under complement.

***Exercise* 173.** *CFL subset*

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.

***Exercise* 174.** *Using the pumping lemma.*

Prove that

$$L = \{a^n b^n c^i \mid i \leq n\}$$

is not context-free.

***Exercise* 175.** *More pumping* 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. $\{a^n b^m \mid n \leq m^2\}$

***Exercise* 176.** *Taxonomy* 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. $\{a^n b^m c^k d^l \mid n = m \text{ or } k = l\}$
8. $\{a^n b^m c^k d^l \mid n = m \text{ and } k = l\}$
9. $\{a^n b^m c^k d^l \mid n = k \text{ or } m = l\}$
10. $\{a^n b^m c^k d^l \mid n = k \text{ and } m = l\}$
11. $\{a^n b^m c^k d^l \mid n > k \text{ or } m > l\}$
12. $\{a^n b^m c^k d^l \mid n > k \text{ and } m > l\}$
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.

***Exercise* 177.** Prove that

$$L = \{a^i j c^k \mid i < j < k\}$$

is not context-free.

***Exercise* 178.** 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: $\{a^n b^n c^n \mid n > 999\}$

***Exercise* 179.** Assume the following fact: Over the alphabet $\Sigma = \{a,b\}$, the language $D = \{w \mid w$ can be written as xx 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.

***Exercise* 180.**

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...]

23 Decision Problems About CFGs

Some decision problems concerning context-free grammars.

Some results here are sketched, not fully developed.

Important. The input to the problems below is **not** a *language*. Indeed 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. So the inputs below are, for example, a context-free *grammar*. Then the question that gets asked is (typically) about the language that the CFG generates.

23.1 CFG Membership

CFG Membership

INPUT: CFG G, string w

QUESTION: $w \in L(G)$?

There are polynomial-time algorithms for this problem, such as the Cocke-Kasami-Younger algortihm. Here we just present an exhaustive search, based on bounding the lengths of derivations. One annoyance is that we have to treat the empty string as a special case because of the way we eliminate λ-transitions from a gramma.

Algorithm 27: CFG Membership

```
if $w = \lambda$ then
    return YES if the start symbol is nullable
else
    Build grammar $G'$ with no λ-transitions ;
    Generate all derivation of $G'$ with no more than $2|w| - 1$ steps;
    if if w is derived by one of these then
        return YES
    else
        return NO
```

23.2 CFG Emptiness

CFG Emptiness

INPUT: CFG M

QUESTION: $L(M) = \emptyset$?

Algorithm 28: *CFG* Emptiness
If the start symbol is generating return NO else return YES

23.3 *CFG* Infinite Language

> *CFG infiniteness*
>
> INPUT: a context-free grammar G
>
> QUESTION: is $L(G)$ infinite?

First, a preliminary claim:

if G is a CFG with no chain or λ rules, with n variables, and whose rules all have right-hand sides of length at most k then, letting $p = k^n$, $L(G)$ *is infinite if and only if there is a string* $z \in L(G)$ *with* $p \leq |z| < 2p$.

To verify the claim: first note that this p is precisely the p given by the proof of the Pumping Lemma for CFLs. Now, if there is a such a string in $L(G)$ then $L(G)$ is certainly infinite, since the Pumping Lemma explicitly builds infinitely many strings in $L(G)$. On the other hand, suppose $L(G)$ is infinite, we exhibit a string $z \in L(G)$ with $p \leq |z| < 2p$. Indeed, let z be the shortest string in $L(G)$ whose length is at least p, and suppose for sake of contradiction that the length of this z were not less than $2p$. By the Pumping Lemma we can write z as $uvwxy$ with $|vwx| \leq p$, at least one of v and x non-empty, and uv^iwx^iy in $L(G)$ for all i. In particular uwy is in $L(G)$. But we supposed that the length of z was at least $2p$, so (by the fact that $|vwx| \leq p$) the length of uwy must be at least p. This contradicts the choice of z as the shortest string in $L(G)$ of length at least p.

Now it is easy to describe the algorithm for testing infiniteness of $L(G)$:

```
Convert $G$ to an equivalent grammar $G'$ with no chain or $\lambda$ rules;
Let $n$ be the number of variables in $G'$;
let $k$ be the maximum size of a right-hand-side;
let $p$ be $k^n$.

For each string $z$ with $p \leq |z| < 2p$,
    run the membership test asking whether $z \in L(G')$.
    If the answer is ever "yes", return "yes".

// If we get here no "yes" was found in the loop
answer "no".
```

Note that the fact that G' may differ from G as to whether the empty string is generated makes no difference to the question of whether $L(G)$ is infinite.

Observe that the input to the algorithm is the grammar G — calculating the G' and an appropriate p, namely $k^{\text{number of variables}}$, in the first line, is a job for the algorithm.

The correctness of this algorithm is an immediate consequence of the claim we proved above.

23.4 *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.

23.5 *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.

23.6 Exercises

***Exercise* 181.** Show that the following problem is undecidable, *assuming* that the *CFG* Universality problem is undecidable.

> *CFG Subset*
>
> INPUT: *CFGs* M_1, M_2
>
> QUESTION: $L(M_1) \subseteq L(M_2)$?

***Exercise* 182.** Show that the following problem is undecidable, *assuming* that the *CFG* Universality problem is undecidable.

> *CFG Equality*
>
> INPUT: *CFGs* M_1, M_2
>
> QUESTION: $L(M_1) = L(M_2)$?

Part IV

Computability

24 Computability: Introduction

Our goal is to study the most fundamental question about computing:

> *what problems can be solved by algorithms?*

First we need to decide exactly what we mean by "problem" and exactly what we mean by "algorithm."

The short answers are

- We will mainly focus on *decision problems.* These are problems with yes/no answers. We model these formally as *languages* over an alphabet.
- We simply take "algorithm" to mean, "program" in any standard programming language. It is an well-know fact that all general-purpose programming languages have exactly the same expressive power.

24.1 The Halting Problem

We start by presenting the cornerstone result of the whole subject: the undecidability of the decision problem known as the Halting Problem. It doesn't look so interesting at first glance. But it is the most important problem, because—as we will see—it is the key to deriving essentially all other undecidability results.

> *The Halting Problem*
>
> INPUT: a program p and a string x
>
> QUESTION: does p halt on x?

We ask the question: Can there exist a "halt-test" program that will answer this question? That is, for any pair of inputs p and x, halt-test should return 1 if program p would halt.

The key to understanding the proof of the next result is to remember that a program is just a text file. So it makes perfect sense to run a program on another program; in particular it makes perfect sense to run a program on itself.

24.1 Theorem. *There is no program that solves the Halting Problem.*

Proof. We assume that there is such a program and obtain a contradiction. So suppose that `haltTest` is a program taking two inputs p and x with

$$\texttt{haltTest}[p,x] = \begin{cases} \text{returns 1} & \text{if } p \text{ halts on } x \\ \text{returns 0} & \text{if } p \text{ fails to halt on } x \end{cases}$$

We can then write—in C, just to be concrete—the following code.

```
int main (p) {
       if (haltTest(p,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 `haltTest` does what we claim it does, we always get a contradiction. Thus there can be no procedure `haltTest` that performs as we claimed originally. That's the end of the proof. ///

24.2 The Halting Problem Revisited

Here is the same proof, presented visually. This presentation shows a subtle relationship with the diagonalization argument used to prove certain sets uncountable. 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 do draw pictures.

Remember that a program is just a text file, and so can be viewed as a string. We know how to enumerate all Σ_2 strings, as $w_0, w_1, \ldots$, each string occurring as one of the w_i. And since we know how to consider each string in turn and determine whether it is a legal program, it follows that we can generate a list $p_0, p_1, \ldots$, of all programs.

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 p_i (as an input string) is accepted by program p_r.

	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 happens to halt on precisely the strings with even indices,
- program number 5000001 halts on no inputs, ...

That picture, then, is precisely the picture of the behavior of our infamous `haltTest` program. 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: ...” 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 gave legal code for `main`!

So where is the contradiction? It lies in the fact that the matrix of $\downarrow$ and $\uparrow$ symbols cannot be computed by a *program* `haltTest`. 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 undecidabibility 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. That is the business of the next section.

24.3 Exercises

***Exercise* 183.** Consider the following 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 (haltTest(p,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 `haltTest` behaves as assumed, and then reasoning about this `main1` program in exactly the same way as we did for `main`, namely by examining `main1(main1)`?

25 Decision Problems, Languages, and Encoding

We are interested in understanding which problems can be solved by computer. The first step is to be clear about what kinds of problems can even be *given as input* to a computer. The main point is that computers can only process finite objects: integers, rational numbers, graphs and trees, automata, files, etc. Computers cannot directly process mathematical functions, or real numbers, for example.

25.1 Things Are Strings

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.

In other words, without loss of generality,

> *When we manipulate finite objects, we are manipulating strings.*

25.2 Decision Problems Are Languages

Let's define precisely what we mean by **decision problem**. Informally, a decision problem is "a family of yes/no questions." But not arbitrary yes/no questions of course: we mean questions that have well-defined mathematical answers.

There is a close connection between the kind of "yes/no" questions that we often want to solve in real life, such as testing integers for primality or testing graphs for being planar or testing programs for correctness, and questions about *membership in languages.* The connection is simply this: once we settle on an alphabet Σ over which we will represent instances of our problem, the instances for which the answer to our question is "yes" define a language over Σ, that is, a subset of Σ^*. That is:

> *We identify a given* ***problem*** *(in the sense of "a bunch of yes/no questions") with the* ***language*** *consisting of all the strings getting a "yes" answer to the given problem.*

This change in perspective and terminology is useful, for example when you are explaining to your roommate what you've been up to in this class, who may not want to sit still for a discussion of alphabets and languages.

25.1 Example. As described above, the following *language*

$$\{w \in \{0,1\}^* \mid \text{ } w \text{ codes an even number.}\}$$

can be identified with the *problem*

> *Evenness*
>
> INPUT: An integer n in binary
>
> QUESTION: Is n even?

25.2 Example. The following *problem*

> *Primality*
>
> INPUT: An integer n in binary
>
> QUESTION: Is n prime?

can be associated with the *language*

$$\{w \mid w \text{ is a binary code for a prime number}\}$$

The second question is "harder" then the first, intuitively. As we go on we'll see various ways of making that precise. But the point to be made here is that these are the same *kind* of question.

25.3 Example. The following *problem*

> *Termination-on-17*
>
> INPUT: A C program p
>
> QUESTION: On input 17, does p terminate normally?

can be associated with the *language*

$$\{p \mid p \text{ is the source of a C program which terminates normally on input 17}\}$$

In this way, the issue of whether a given *problem* is solvable by machine is transformed into a question about whether membership in the associated *language* is solvable by machine.

25.2.1 Problem instances with more than one input

We have been talking about problems as languages, that is, subsets of Σ^* for whatever Σ is at hand. But many problems have—intuitively— more than one input.

For example, when we formalize the "Graph Connectivity" carefully:

> *Graph Connectivity*
>
> INPUT: A graph G and two nodes n_1 and n_2
>
> QUESTION: Is there a path from n_1 to n_2?

...conceptually this is a completely straightforward extension of what we have done before but we need to be a little fussy when we formalize it. It turns out that it is most convenient to still formalize this as a language, that is, as defining a single set of strings over an alphabet. Here is how we represent instances of a problem that intuitively has several parts to each input (such as the example immediately above).

- Isolate an alphabet Σ in which we can represent the data for the problem instances. (In the Graph Connectivity example we might let Σ be the a set of symbols to represent nodes and edges.)
- Specify a symbol \$ that will never occur in the data for the problem instances.
- Represent the problem instance involving Σ^*-strings $\langle a_1, a_2, \dots, a_k \rangle$ as the single string $a_1\$a_2\$\dots\$a_k$ This is a string over the alphabet $\Sigma \cup \{\$\}$. (In the Graph Connectivity example, problem instances would look like $g\$x\$y$ where g is some string encoding the graph and x and y are strings encoding nodes.)

It should be clear that from the perspective of answering questions about whether there are or are not algorithms to solve problems, the details of such encoding are not important. So, rather than include such details when we describe problems we will just use the notation $\langle a_1, a_2, \dots, a_k \rangle$ to stand for a problem instance determined by $a_1, a_2, \dots, a_k$

The crucial takeaway from this observation is that even decision problems that seems at first glance to be about more than one input can be phrased as decision problems about a single (complex) input. And so *they are also language-membership problems.*

25.3 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 are making a *DFA* as part of designing a traffic light, Σ may include "red", "yellow" and "green" as alphabet symbols. And so on. 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 encode each a_i as a length-k bit string. And then any word 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 word to the original word.[14]

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

25.4 Enumerating the Bit Strings

It is often useful to have a nice enumeration of the finite bitstrings; we develop that here. 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 this is easily fixed, as follows.

Suppose we list the bitstrings in lexicographic order (ie ordering first by length, then by comparing earliest difference).

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

[14]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.

We can then associate these strings to naturals numbers in the obvious way:

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

Now suppose you were asked for the encoding of 173? Do you have to walk through the list of strings till you get to 173? Similarly, what if you had to decode 001101001? No; it's a nice and somewhat amusing fact that coding and decoding under the above scheme can be done "arithmetically" as follows.

Given a string x, add a "1" to the beginning of x, calculate the number that $1x$ encodes in binary, then subtract 1 from the answer. You can check that that gives the association above.

25.5 Summary

1. We can naturally encode finite objects as strings over an alphabet Σ
2. Having chosen such a Σ, we can easily encode tuples of objects as strings over an enriched alphabet $\Sigma \cup \{\$\}$.
3. Decision problems are naturally represented as languages, perhaps using the $\Sigma \cup \{\$\}$ trick.
4. We can encode strings over any alphabet, such as some $\Sigma \cup \{\$\}$, as strings over Σ_2.
5. Putting the above remarks together, we can naturally decision problems, even those with several inputs, as languages over Σ_2.
6. There is a natural way to put the stings over Σ_2 into one-to-one corresponds with the natural numbers.

25.6 Exercises

***Exercise* 184.** *Languages and decision problems* For each of the following languages write the corresponding decision problem (in Input/Question style).

1. The set of strings of even length: $\{x \mid \text{length}(x) \text{ is even}\}$
2. The set of primes in binary.
3. The set $\{a^n b^n \mid n \geq 0\}$.

4. (Assume that we have agreed upon some convention about how to encode graphs as strings.) The set of strings which code graphs that are directed and acyclic.

5. The set of terminating programs: $\{p \mid \forall x . p[x] \downarrow\}$

***Exercise* 185.** *Decision problems and languages* For each of the following problems write down the corresponding language in "set comprehension" style

1.
   INPUT: An integer m
   QUESTION: Is m a perfect square?

2.
   INPUT: A graph G
   QUESTION: Is G connected?

3.
   INPUT: A program m
   QUESTION: Does m accept every string?

4.
   INPUT: A program m
   QUESTION: Is $L(m)$ regular?

5.
   INPUT: A program m
   QUESTION: Is there some program n with fewer lines than m such that $L(n) = L(m)$?

***Exercise* 186.** Using the enumeration of Σ_2^* described in Section 25.4:

- Give the number corresponding to 101. Give the number corresponding to 0101. Give the number corresponding to 0000000000 (there are ten 0s in that last string).

- Give the string corresponding to 10. Give the string corresponding to 99. Give the string corresponding to 1024.

***Exercise* 187.** Here we define a "pairing function" from $\mathbb{N}^2$ to $\mathbb{N}$. Define:

$$\pi(x,y) = \frac{1}{2}(x+y)(x+y+1)+y$$

Draw a picture of $\mathbb{N}^2$ as a grid, that is, with (0,0), (0, 1), (0,2) ... as one row, then (1,0), (1, 1), (1,2) ... as the next row, etc.

Now apply π to these pairs, and see the pattern.

Using this intuition, argue that π is a bijection from $\mathbb{N}^2$ to $\mathbb{N}$. (It is actually somewhat tricky to prove this rigorously).

***Exercise* 188.** For each $k \geq 2$, define the function $\pi^k : \mathbb{N}^k \to \mathbb{N}$ as follows. Take π^2 to be π from Exercise 187. For $k > 2$, define

$$\pi^k(n_1, n_2, \ldots, n_k) = \pi(\pi^{k-1}(n_1, n_2, \ldots, n_{k-1}), n_k)$$

Show that π^k is a bijection, assuming that π^2 is a bijection.

***Exercise* 189.** It is often useful to have a nice (meaning computable) enumeration of the set $(\Sigma_2^*)^2$. Invent one. That is, define a computable bijection $f : (\Sigma_2^*)^2 \to \mathbb{N}$.

There is no one right answer here! All that matters is that your function f computable by a program.

Now generalize your construction, to define, for each k, a bijection $f^k : (\Sigma_2^*)^k \to \mathbb{N}$.

Hint. You might use the results of Exercise 188.

26 Functions, Programs, and Decidability

The very first thing we have to do is settle on what mathematical model to use to capture general computation.

26.1 Turing Machines and Programs

You may be familiar with two other models of computation:

- finite automata, which accept the *regular* sets, and
- pushdown automata, which accept the *context-free* sets.

Pushdown automata are more powerful than finite automata: they can accept 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 accept 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. In fact this model will be powerful enough to capture everyone's intuitive notion of "computing machine."

This next kind of machine in the hierarchy of computing devices is the *Turing machine* [Alan Turing, 1937]. Roughly speaking a Turing machine is a finite automaton with an unbounded read/write memory. A Turing machine can be used as a language acceptr just as finite automata and pushdown automata can, but since they can write as well as read symbols they can also be viewed as producing output based on their input.

Why are Turing machines are studied?

- Turing machines are exactly as powerful as programs in any standard high-level programming language such as C/C++, Java, or Scheme.
- They are much simpler than programs in a high-level program. This simplicity is very useful in advanced work, for example in showing that certain "combinatorial" problems are not decidable, or in the study of complexity theory.

It is remarkable that a single modification of finite automata (adding read/write memory) buys us universal computing power. But the fact that Turing machines are so low level makes them awkward to work with in most situations: constructing a Turing machine to do even a simple job like testing whether a string is a palindrome is tedious and unenlightening.

So we are going to proceed in a simpler way. We will use *programs themselves* rather than Turing machines, as much as we can, to study computability. Since our goal is to study pure computability as a concept, It does not matter whether we take Turing machines or programs as our "official" formalism. And our investigations will stay closer to our intuitions if we stick with familiar programming idioms.

By the way, it does not matter *which* standard programming language we use to study computability. We can make this remark more precise after we do a little work; see Section 26.4.1.

26.2 Partial Functions

The definitions in this section make perfect sense for an arbitrary alphabet, but remember that we agreed to work over $\Sigma_2 = \{0,1\}$ for simplicity.

26.1 Definition (Partial Functions and Functions)**.**

- *A* partial function $f : \Sigma_2^* \to \Sigma_2^*$ *is a set of ordered pairs* (x,y) *of strings such that for every* $x \in \Sigma_2^*$ *there is* ***at most one*** $y \in \Sigma_2^*$ *with* $(x,y) \in f$.
- *A* function $f : \Sigma_2^* \to \Sigma_2^*$ *is a set of ordered pairs* (x,y) *of strings such that for every* $x \in \Sigma_2^*$ *there is* ***exactly one*** $y \in \Sigma_2^*$ *with* $(x,y) \in f$.
- *The* domain *of a partial function* f *is* $\{x \mid \exists y,\ (x,y) \in f\}$.

Mathematical convention leads us to usually write $f(x) = y$ instead of $(x,y) \in f$.

It is rare in ordinary mathematics to work with partial functions. But in computability theory, partial functions are the norm. This may seem like a weird design choice for our investigation. You'll see whay we *have* to make that choice, later (Section 32).

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_2^*: we are just leaving open that possibility. That is to say, a "function" is *also* a "partial function."

26.3 Programs Compute Partial Functions

Recall a few conventions about programs in the C programming environment:

- A text file is conventionally modeled as a sequence of characters (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 take input from the standard input file `stdin`, and it can return values, which we will treat as strings.
- 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.

26.2 Notation. Suppose p is a program, and $x = x_1, \dots x_k$ is a sequence of string over Σ_2. Suppose the program p is executed with standard input consisting of the bitstring x. Then we write

- $p[x] \downarrow y$ if the execution of p halts, and y is the initial sequence of 0s and 1s on standard output (before the first non 0 or 1 character if any);
- $p[x] \uparrow$ if the execution of p does not halt.

What About Running out of Memory (etc)? 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. The point we are making now is that we are not going to put any arbitrary bounds on the time or space as part of the formalism.

26.3 Definition (The Partial Functions Computed by a Program)**.** *Let p be a program. The* partial function computed by *p, denoted* $\mathsf{pfn}(p)$*, is defined on strings x as follows*

$$\mathsf{pfn}(p)(x) = \begin{cases} y & \text{if } p[x] \downarrow y \\ \text{undefined} & \text{if } p[x] \uparrow \end{cases}$$

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* $\mathsf{pfn}(p)$.

26.4 Programs Are Not The Same As 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.

26.4.1 Programming Languages are Equally Expressive

The definitions we have made above are—apparently— only rigorous definitions relative to a particular programming language. That is, if we had decided to use Jana in our discussion rather than C, we might have arrived a different set of mathematical functions on "computable by a program.", it seems. But a crucially important fact is that this is not the case.

Fact 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^*.

Given any two specific languages it is a perfectly precise statement to say that they compute the same partial functions. But we don't state the above Fact as a theorem, just because we don't want to carefully define what we mean by "standard high-level programming language."

26.5 Programs Accept Languages

Once we understand that a program computes a function over strings, we can then consider a program p to define a set of strings over Σ_2: those strings which cause the program to return 1. That is, any program defines a *language*.

26.4 Definition (Language of a Program)**.** *Suppose p is a program. The* language accepted by *p, denoted $L(p)$, is the set of all strings for which p returns 1:*

$$L(p) \overset{def}{=} \{x \mid \ p[x] \downarrow 1\}$$

Note carefully that there are two distinct ways that a program p might fail to accept a string x: either p terminates on x with a return value other than 1, or p does not terminate at all when x is the input. The only way $x \in L(p)$ holds is for p to terminate on x, returning 1.

26.5 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) but rather the fact that they will never run forever on any input.

26.6 Decidable Languages

Finally we can make our central definition.

26.6 Definition. *A language $L \subseteq \Sigma_2^*$ is* decidable *if it is $L(p)$ for some decision procedure p, that is, if there is a program p with*

$$\mathsf{pfn}(p)(x) = \begin{cases} 1 & \textit{if } x \in L \\ 0 & \textit{if } x not \in L \end{cases}$$

We have already seen lots of examples of decidable problems.

- Given a context-free grammar G and a string w, is $w \in L(G)$?
- Given a *DFA* M, is $L(M) = \emptyset$?
- Given two *DFAs* M and N, is $L(M) = L(N)$?

26.7 Two Examples

26.7 Example. Let *p* be a program whose pseudocode is:

```
// accepts $0^n 1^n$
read input into an array;
oksofar = true;
scan array, counting 0's, hold count in n1;
scan array, counting 1's, hold count in n2;

    if see an 0, return (0);

if n1==n2 return (1) else return (0)
```

$L(p)$ is $\{0^n 1^n \mid n \geq 0\}$. Here, more is true than the simple fact that the program *p* there accepts this language. Since *p* always halts, we can treat return (1) as saying "yes, the input string looked like $a^n b^n c^n$ for some *n*," and we can treat return (0) as saying "no, the input string did not look like $0^n 1^n$ for any *n*." That is, *p* is a *decision procedure* for testing membership in the language $\{0^n 1^n \mid n \geq 0\}$.

Compare this situation to the following one.

26.8 Example. Let *q* be a program whose pseudocode is:

```
// Checks a CFG for ambiguity
read 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 *q* is a pretty dumb program, but you should be able to see that if an ambiguous grammar is presented as input to *q* then *q* is guaranteed to eventually realize that *q* is ambiguous and halt with return (1); while if *q* is given a non-ambiguous grammar it will never do an return (1). Thus $L(q)$ is $\{G \mid \ G \text{ is an ambiguous CFG}\}$.

The important thing to note about the two programs above is that *p* is a decision procedure and *q* is not. In fact if *G* is a CFG which is not ambiguous and which generates infinitely many strings, *q* will fail to halt on that grammar. So *q* is *not* what we would call a decision procedure for testing grammar ambiguity.

Each program defines a language. But only the former is a decision procedure. This distinction is crucial, in fact the whole study of decidability hinges on it.

26.8 Undecidable Problems: a Cardinality Argument

It is easy to see that not *every* problem is decidable, just by cardinality.

Notice that a C program is just a text file, which means that it is (just) a finite string over Σ_2. Thus: the set of all C programs is countable.

26.9 Theorem. *There exist languages that are not decidable.*

Proof. There are uncountably many languages over Σ_2. But since there are only countably many C programs there are only countably many decidable languages. ///

One way to read this result is that *most* languages are undecidable. But that is not so satisfying, and it doesn't give any insight about what kinds of things are undecidable. Read on.

26.10 Check Your Reading. *Explain why the argument above goes through even if we had not committed to the binary alphabet Σ_2. That is explain why for any non-empty alphabet Σ, there are uncountably many languages over Σ.*

On terminology

Sometimes different words are used for these all notions. (Skip the following discussion if you like, it's here to reconcile the notation here with what you might see in other texts)

- A synonym for *decidable* used by some authors is *recursive*.

  The term "recursive" used in this context has nothing to do with the idea of a program calling itself. It just happens to be the term used by the original researchers exploring questions of decidability (at a time before there were any high-level programming languages supporting what we now call "recursion!") So do not read anything into the term — it means nothing more nor less than what the definition of decidable says.

- The terms *recursively enumerable* and *semidecidable* are synonyms. Sometimes the term *Turing-recognizable* is used.

  The phrase "recursively enumerable" is also a bit strange at this point. A justification for this phrase is in Section 31 below. A convenient abbreviation for "recursively enumerable" is "RE." Indeed this is such an easy-to-pronounce shorthand that most people prefer to use "RE" to refer to a semi-decidable language.

- There are synonymous terms in use for the notion of computable function as well: some authors call computable functions "recursive" functions (which is really confusing: we won't do this).

26.9 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. To see the point consider the following two problems, taken from [RR67]

> *Exactly n 5s*
>
> INPUT: 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. (It is clearly semi-decidable.) But constrast this with the following problem.

> *At Least n 5s*
>
> INPUT: 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.

To see this, 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` $n \leq k$`, return 1, else return 0.`

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 that doesn’t matter. The point is that regardless of our human state of knowledge at the moment, we have shown that **there exists** an algorithm for the *At Least n 5s* problem. 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.

What you should take away from this example is a deeper understanding of what undecidability means. It is not about how hard certain 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

26.11 Check Your Reading. *True or false: the number*

$$282468234668236482342851872043845045 6$$

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.

26.10 Section Summary

- A **partial function** $f : \Sigma_2^* \to \Sigma_2^*$ is a set of pairs of strings such that for every $x \in \Sigma_2^*$ there is at most one $y \in \Sigma_2^*$ such that $f(x) = y$.

  If for every x there is *exactly* one y such that $f(x) = y$, we imply say that f is a *function.*

  Sometimes, if we want to emphasize the fact a certain f is really a function, that is, defined on all inputs, we will say **total function.** But that is just a reminder to the reader, strictly speaking, the word “total” is redundant.

- When p is a **program,** $\mathsf{pfn}(p)$ is the **partial function** defined by

$$\mathsf{pfn}(p)(x) = \begin{cases} y & \text{if } p[x] \downarrow y \\ \text{undefined} & \text{if } p[x] \uparrow \end{cases}$$

- A partial function $f : \Sigma_2^* \to \Sigma_2^*$ is **computable** if there is some program p such that f is $\mathsf{pfn}(p)$.

- When p is a program, the **language accepted by** p is the set of all strings for which p returns 1, and is denoted $L(p)$:

$$L(p) \overset{\text{def}}{=} \{x \mid \ p[x] \downarrow 1\}$$

- Program p is a **decision procedure** if for every input x,

$$\text{either } p[x] \downarrow 0 \quad \text{or} \quad p[x] \downarrow 1.$$

- A language $L \subseteq \Sigma_2^*$ is **decidable** if it is $L(p)$ for some decision procedure p.
- When a language fails to be decidable we may say that it is **undecidable.** That is, “not decidable” and “undecidable” are synonymous.

26.11 Exercises

***Exercise* 190.** Give a concrete example showing that we can have $L(p) = L(q)$ yet $\mathsf{pfn}(p) \neq \mathsf{pfn}(q)$ Even more, find *three* programs p, q, and r that compute three different functions, yet they all accept the same language.

***Exercise* 191** (A useful construction)**.** Suppose p is a program, which returns the value 1 for certain inputs (and on other inputs either returns something else or maybe loops forever). Then we can construct another program p' that behaves as follows. On any input x:

- if $p[x] \downarrow 1$ then $p'[x] \downarrow 1$,
- if $p[x] \downarrow y$, for any $y \neq 1$, then $p'[x] \uparrow$
- if $p[x] \uparrow$ then $p'[x] \uparrow$

Thus $L(p') = L(p)$ but the domain of the function computed by p' is precisely $L(p')$.

Show how to construct p' from p (cf. Section 191). Notice that what you are asked for here is a *construction* that takes as input some program p and returns as output an appropriate program p'.

27 Some Decidable Languages

Which languages are decidable? There are lots of easy examples:

- the set of prime numbers
- the set connected graphs
- the set of circuits computing a given boolean function
- the set of strings that make a syntactically legal Java program

When you start tackling more sophisticated questions, you can encounter less-obviously-decidable questions. Most especially, questions that involve reasoning about systems can start to edge towards undecidability (the prime example is of course the Halting Problem). In this section we will look at languages which embody reasoning about systems modeled as finite automata and context-free grammars. The moral of the story is that these relatively simple systems do mostly lend themselve to decidable reasoning. When—in later chapters—we explore reasoning about fully-expressive systems, *i.e.* full-fledged program, the news is more discouraging.

In the second part of this chapter we collect some useful results about closure of the decidable languages under operations such as intersection, concatenation, etc.

27.1 Decision Problems about Finite Automata

Refer to Section 15, where we provided a selection of questions concerning *DFAs*, *NFAs*, and regular expressions, each of which was decidable. They were presented there in decision-problem formalism; we describe them more succinctly here as languages.

1. DFA Membership
$$\{\langle M, w\rangle \mid M \text{ is a } DFA,\ w \text{ is a string, } w \in L(M)\}$$
2. DFA Emptiness
$$\{M \mid M \text{ is a } DFA,\ L(M) = \emptyset\ \}$$
3. DFA Universality
$$\{M \mid M \text{ is a } DFA \text{ over an alphabet } \Sigma, \text{ and } L(M) = \Sigma^*\ \}$$

4. DFA Subset

$$\{\langle M_1, M_2\rangle \mid M_1, M_2 \text{ are } DFAs, \text{ and } L(M_1) \subseteq L(M_2)\}$$

5. DFA Equality

$$\{\langle M_1, M_2\rangle \mid M_1, M_2 \text{ are } DFAs, \text{ and } L(M_1) = L(M_2)\}$$

6. DFA Infinite Language

$$\{M \mid M \text{ is a } DFA, \text{ and } L(M) \text{ is infinite }\}$$

7. NFA Membership

$$\{\langle M, w\rangle \mid M \text{ is an } NFA, w \text{ is a string, } w \in L(M)\}$$

8. Regular Expression Membership

$$\{\langle E, w\rangle \mid E \text{ is a regular expression, } w \text{ is a string, } w \in L(E)\}$$

27.2 Decision Problems About CFGs

In Section 23 we looked at a selection of decision languages concerning *CFGs* and *PDAs*. Not all of these were decidable. The following ones were.

1. *CFG* Membership

$$\{\langle G, w\rangle \mid G \text{ is a } CFG, w \text{ is a string, } w \in L(G)\}$$

2. *CFG* Emptiness

$$\{G \mid G \text{ is a } CFG, \text{ and } L(G) = \emptyset\}$$

We said in Section 23 that the *CFG* Universality and *CFG* Ambiguity problems were not decidable; those proofs can be found in Section 35.

27.3 Closure Properties of the Decidable Languages

Let's collect some easy abstract properties of decidable languages.

27.1 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 such that $L(p_A)$ is A, and let p_B be a decision procedure such that $L(p_B)$ is B. Our goal is to show that there exists a decision procedure 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 are accepting, 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 $L(r)$ is $A \cap B$. 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 $L(r)$ is $A \cap B$ 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 such that $L(p_A)$ is A, and let p_B be a decision procedure such that $L(p_B)$ is B. Our goal is to show that there exists a a decision procedure program r such that $L(r)$ is $A \cup B$. 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 are accepting, 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 $L(r)$ is $A \cup B$. 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 $L(r)$ is $A \cup B$ 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 such that $L(p_A)$ is A, and let p_B be a decision procedure such that $L(p_B)$ is B. Our goal is to show that there exists a a decision procedure r such that $L(r)$ is 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 are accepting, 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 $L(r)$ is 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 $L(r)$ 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 $w = w_1 w_2$.

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 such that $L(p_A)$ is A. Our goal is to show that there exists a a decision procedure program r such that $L(r)$ is A^*. We exhibit pseudocode for such a program as follows:

```
on input $w$;
consider in turn each sequence of strings $w_1, w_2, \dots w_n$ such that
$w_1 w_2 \dots w_n = w$:
    run $p_A$ on $w_1$;
    run $p_A$ on $w_2$;
    ...
```

```
        run $p_A$ on $w_n$;
        if each of these runs are accepting, 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 $L(r)$ is 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, $w_2, \dots, w_n$ such that $w_1 w_2 \dots, w_n = w$. 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 $L(r)$ is A^* is straight from the definition of asterate: a string w is in A^* if and only if there are strings $w_1, w_2, \dots w_n$ such that each w_i is in A and $w_1 w_2 \dots w_n = w$.

///

27.4 Exercises

***Exercise* 192.** Give decision procedures for each of the following problems.

Each of your answers should be a script that does nothing besides

- call one or more algorithms we have developed in these notes for building new automata from old ones, and
- call one or more algorithms we have developed in this section for deciding properties of automata

Use the algorithms in this section as a guide to how formal to be.

To give you an idea of how things things go, the first one is done for you.

1. Given a *DFA M*: Does M accept any strings of even length?

   Solution.

   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....

The algorithm

```
Algorithm 29: 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 $L(P)$ is empty then
    return NO
else
    return YES
```

2. Given a *DFA M*: Does M accept every even length string? (This is not the same as asking whether M accepts *precisely* the even-length strings.)
3. Given a *DFA M*: Does M reject infinitely many strings?
4. Given *DFAs M* and N: Do M and N differ on infinitely many inputs? (More formally: is the symmetric difference between $L(M)$ and $L(N)$ infinite?)
5. Given *RegExps E* and F: Do E and F define the same language?

28 Some Undecidable Languages

So far we have only one example of an undecidable language: the language associated with the Halting Problem:

$$\mathsf{Halt} \overset{\text{def}}{=} \{\langle p,x\rangle \mid \ p \text{ halts on } x\ \}$$

In this section we generate lots more examples, focusing on properties of programs. In later chapters we look at problems in other domains, such as grammars and arithmetic.

28.1 Standalone Arguments

The hard way to show languages undecidable is to give an argument directly, without piggy-backing on any other known undecidability results. Here are two examples (just variations on what we did with Halt).

28.1.1 The Self-Halting Problem

Define

$$\mathsf{SelfHalt} \overset{\text{def}}{=} \{p \mid \ p \text{ halts on } p\ \}$$

This is the language of programs that halt on themselves. It is a useful one-argument version of the Halting Problem. It is undecidable, and indeed we already did the work in showing it undecidable, in Section 24. That is, what the proof of the Halting Problem *really* showed is that SelfHalt is undecidable. To make the point, here is a full proof that SelfHalt is undecidable; please note that almost all of it is verbatim from the proof we did in Section 24.

Proof. We assume that there is a program deciding SelfHalt 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—in C, just to be concrete—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 `haltTest` that performs as we claimed originally. That's the end of the proof. ///

28.1.2 The Acceptance Problem

The *Acceptance Problem* is a mild variation on the Halting Problem. Suppose we don't ask whether a given program *P* halts on a given input *x*, but rather ask whether *P accepts x*. We use the term "accepts" to evoke acceptance in the sense of finite automata, but we need to define what we mean exactly when we say that a *program* accepts an input.

Recall the definition of acceptance (not the same as halting, since a program could halt on an input without accepting it).

28.1 Definition (Program Acceptance)**.** *Let p be a program and let x be an input string. Say that p* accepts *x if, when p is executed with x as input, p halts and returns 1.*

The decision problem is

Program Acceptance

INPUT: A program *p* and an input *x*

QUESTION: Does *p* accept *x*?

and the language corresponding to this decision problem is

$$\mathsf{Acc} \overset{\text{def}}{=} \{\langle p,y\rangle \mid \text{program } p \text{ accepts input } y.\}$$

The Acceptance Problem is not decidable. The proof is almost exactly like the proof for the Halting Problem (which isn't surprising since the the two problems are pretty similar). The fact that the proofs are so similar makes it valuable to study them both. By seeing where they are the same and where they are different you will get insight into the fundamental idea.

28.2 Theorem. *The Acceptance Problem is not decidable.*

Proof. For sake of contradiction, Suppose that `accTest` is a program taking two inputs p and x with

$$\texttt{accTest}(p,x) = \begin{cases} \text{returns 1} & \text{if } p \text{ accepts } x \\ \text{returns 0} & \text{if } p \text{ fails to accept } x \end{cases}$$

Now we build the following program. ...

```
int accSwap (p)
{
  if (accTest(p,p) == 1)    /* if p accepts p         */
     return (0);             /* then reject             */
  else                       /* if p doesn't accept p  */
     return (1);             /* then return 1           */
}
```

So

$$\texttt{accSwap}(p) = \begin{cases} \text{returns 0} & \text{if } p \text{ accepts } p \\ \text{returns 1} & \text{if } p \text{ fails to accept } p \end{cases}$$

Now we play the same trick as we did with the Halting Problem: we analyze `accSwap[accSwap]`. We have

$$\texttt{accSwap [accSwap] }) = \begin{cases} \text{returns 0} & \text{if accSwap accepts accSwap} \\ \text{returns 1} & \text{if accSwap fails to accept accSwap} \end{cases}$$

This is absurd. So our assumption about `accTest` is inconsistent, and no such `accTest` program can exist.

///

28.2 Undecidability via Reduction

The most powerful technique to show languages undecidable is to leverage previously-derived undecidability results. To show a language X to be undecidable we can proceed this way:

1. Cleverly choose some language U that has previously been proven to be undecidable
2. Make an argument by contradiction, namely, given an argument that shows that *if* X were decidable, *then* U would be undecidable

Typically, the way we establish (2) is to give pseudocode for a program which which calls as a subroutine a procedure D_X that asks questions about membership in X, and which would correctly compute membership in U under the assumption that D_X correctly computes membership in X. Since we already know that there cannot be a correct decision procedure for U, we can conclude that there cannot be a correct decision procedure for X.

28.2.1 The Hello World Problem

Consider the decision problem of determining whether a given program computes the constant function that always returns "Hello World."

28.3 Proposition. *The following language is undecidable.*

$$\mathsf{HelloWorld} \overset{def}{=} \{p \mid \textit{ p is a program that always prints "Hello World".}\}$$

Proof. We show that if HelloWorld were decidable, then SelfHalt would be decidable. So let us suppose, for the sake of contradiction that we had a decision procedure $D_{\mathsf{HelloWorld}}$ for membership in HelloWorld. With the help of $D_{\mathsf{HelloWorld}}$, we can build a decision procedure D for membership in SelfHalt. Here is D.

```
on input w,
  1. build the following program p
       On input x:
       simulate w on w;
       print ''Hello World''
  2. return the result of D_HelloWorld on p.
```

So, to be clear, D is a decision procedure that *first* constructs a certain program p, and *then* calls $D_{\mathsf{HelloWorld}}$ on that program.

We want now to show that D returns 1 on an input w if and only if $w \in \mathsf{SelfHalt}$.

- Suppose w is in $\mathsf{SelfHalt}$. Then the new program prints "Hello World" on all inputs x (after the intial simulation of w on w). So $D_{\mathsf{HelloWorld}}$ will return 1 on this p. So D will return 1 on w.
- Suppose w is not in $\mathsf{SelfHalt}$. Then on any input x, the new program will loop forever. So the new program certainly doesn't print "Hello World" on all inputs. So $D_{\mathsf{HelloWorld}}$ will return 0 on this p. So D will return 0 on w.

We have just argued that D is a decision procedure for $\mathsf{SelfHalt}$, provided that $D_{\mathsf{HelloWorld}}$ behaves as we assumed. Since we already know that there cannot be a decision procedure for $\mathsf{SelfHalt}$, we conclude that $D_{\mathsf{HelloWorld}}$ cannot behave as we assumed. ///

This is a hard proof to understand. But it is worth the effort, since it actually proves lots of seemingly different results with no extra effort. Read on.

28.2.2 The Identity Problem

Consider the decision problem of determining whether a given program computes the identity function.

28.4 Proposition. *The following language is undecidable.*

$$\mathsf{Id} \overset{def}{=} \{p \mid \ p \text{ is a program computing the identity function}\}$$

Proof. This proof will be almost word-for-word the same as the proof of Proposition 28.3. Read carefully, and notice the tiny places where it differs.

We show that if Id were decidable, then $\mathsf{SelfHalt}$ would be decidable. So let us suppose, for the sake of contradiction that we had a decision procedure D_{Id} for membership in L_{id}. With the help of D_{Id}, we can build a decision procedure D for membership in $\mathsf{SelfHalt}$. Here is D.

on input w,

1. *build the following program p*

```
    On input x:
    simulate w on w;
    return x
```

2. *return the result of* D_{Id} *on* p.

We want now to show that D returns 1 on an input w if and only if $w \in \mathsf{SelfHalt}$.

- Suppose w is in SelfHalt. Then the new program computes the identity function (in a perverse way, to be sure). So $D_{\mathsf{HelloWorld}}$ will return 1 on this p. So D will return 1 on w.
- Suppose w is not in SelfHalt. Then on any input x, the new program will loop forever. So the new program certainly doesn't compute the identity function. So $D_{\mathsf{HelloWorld}}$ will return 0 on this p. So D will return 0 on w.

We have just argued that D is a decision procedure for SelfHalt, provided that D_{Id} behaves as we assumed. Since we already know that there cannot be a decision procedure for SelfHalt, we conclude that D_{Id} cannot behave as we assumed.

///

28.2.3 The Always-Halting Problem

Consider the decision problem of determining whether a given program halts on *all* of its inputs.

28.5 Proposition. *Let* AlwaysHalts $\overset{def}{=} \{p \mid p$ *is a program that always halts.*$\}$ *Then* AlwaysHalts *is undecidable.*

Proof. We show that if AlwaysHalts were decidable, then SelfHalt would be decidable. So let us suppose, for the sake of contradiction that we had a decision procedure $D_{\mathsf{AlwaysHalts}}$ for membership in AlwaysHalts. With the help of $D_{\mathsf{AlwaysHalts}}$, we can build a decision procedure D for membership in SelfHalt.

And now we use **exactly the same** D as we used in the proof of Proposition 28.3, with the one exception that in the final step where we used D_{Id} in that proof, we use $D_{\mathsf{AlwaysHalts}}$ here. And we use **exactly the same** argument to argue that D returns 1 on an input w if and only if $w \in \mathsf{SelfHalt}$.

This works because (i) when the inner program computes the identity function (as we discussed in the first proof) that inner program is a program that always halts, and so is in $\mathsf{AlwaysHalts}$, and (ii) when the inner program loops on all inputs, that inner program is a program that doesn't always halt, and so is not in $\mathsf{AlwaysHalts}$. ///

28.2.4 The Emptiness and Non-Emptiness Problems

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 no inputs is the same as saying the $\mathsf{pfn}(p)$ is the empty function.

28.6 Proposition. *The following language is undecidable.*

$$\mathsf{NonEmp} \overset{def}{=} \{p \mid \mathsf{pfn}(p) \neq \emptyset\}$$

Proof. This is another result that—magically—submits to the same proof as the two previous ones. We assume we have a decision procedure D_{NonEmp} for NonEmp, then on input w we build the same new program p, and observe that determining whether p is in NonEmp is equivalent to determining whether $w \in \mathsf{SelfHalt}$. ///

By the way, since the decidable languages are closed under complement, we have the following corollary.

28.7 Corollary. *The following language is undecidable.*

$$\mathsf{Emp} \overset{def}{=} \{p \mid \mathsf{pfn}(p) = \emptyset\}$$

Proof. If Emp were decidable, then its complement NonEmp would be decidable, contradicting Proposition 28.6. ///

28.3 Reduction by Specialization

One of the easiest ways to do a reduction 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.

28.3.1 The Program Equivalence Problem

We will now prove that the following decision problem is undecidable.

> *Program Equivalence*
>
> INPUT: Two programs p_1 and p_2.
>
> QUESTION: Does $\mathsf{pfn}(p_1) = \mathsf{pfn}(p_2)$?

To be consistent with the other results in the chapter we will phrase the result in terms of a language.

28.8 Proposition. *The following language is undecidable.*

$$\mathsf{Equiv} \overset{def}{=} \{\langle p_1, p_2\rangle \mid \mathsf{pfn}(p_1) = \mathsf{pfn}(p_2)\}$$

Proof. Recall that we proved that the following language is undecidable.

$$\mathsf{Id} \overset{\text{def}}{=} \{p \mid p \text{ is a program computing the identity function}\}$$

We now show that if Equiv were decidable, then Id would be decidable. So let us suppose, for the sake of contradiction, that we had a decision procedure D_{Equiv} for membership in Equiv. With the help of D_{Equiv}, we can build a decision procedure D for membership in Id.

Let p_{id} be the following program: `on input` x`, return x .`

With the help of p_{id} we can build our D:

> *On input p, run D_{Equiv} with $p_1 = p$ and $p_2 = p_{id}$, and return that answer.*

This is a decision procedure for Id, since to say that a program p computes the identity function is precisely to say that it computes the same function as p_{id}. Since the existence of a decision procedure for Id is a contradiction, there cannot be a program deciding Equiv. ///

28.4 Exercises

***Exercise* 193.** Show that SelfAcc is undecidable, where

$$\begin{aligned}\mathsf{SelfAcc} &= \{w \mid \langle w, w\rangle \in \mathsf{Acc}\}\\ &= \{w \mid w \in L(w)\}\\ &= \{w \mid w[w] \downarrow 1\}\\ &= \{w \mid \mathsf{pfn}(w)(w) = 1\}\end{aligned}$$

Hint. This is just a simplification of the proof about the Acc, there are no new ideas. Your job here is to write a *careful* proof: it will be a good exercise in keeping all the self-referential things straight in your head.

***Exercise* 194.** Consider a variation on the proof of the undecidability of the Acceptance Problem, in which we keep everything the same except that we use the following program to try to get our contradiction.

```
int main2 (p)
{
L:  if (accTest(p,p))       /* if p accepts p         */
     while 1 do ;           /* then loop forever      */
else                        /* if p doesn't accept p  */
     return(1);             /* then return 1          */
}
```

Will this support a proof-by-contradiction as we did earlier? That is, if we assume that `accTest` behaves as assumed, can we get a contradiction by reasoning about this `main2` program in exactly the same way as we did before, namely by examining `main2(main2)`?

***Exercise* 195.** Prove that the following problem is undecidable.

Program Inclusion

INPUT: Two programs p_1 and p_2.

QUESTION: Is $L(p_1) \subseteq L(p_2)$?

Hint. You might proceed in a similar way as we did in Proposition 28.8, using a different language as the "specializer."

Alternatively, you could simply use the result of Proposition 28.8, and show that if ProgramInclusion were decidable, then ProgramEquivalence would be as well.

***Exercise* 196.** Show that the following problem is undecidable, *assuming* that the *CFG* Universality problem is undecidable.

> *CFG Equality*
>
> INPUT: *CFGs* M_1, M_2
>
> QUESTION: $L(M_1) = L(M_2)$?

Hint. Use the "specialization" trick.

***Exercise* 197.** Show that the following problem is undecidable, *assuming* that the *CFG* Universality problem is undecidable.

> *CFG Subset*
>
> INPUT: *CFGs* M_1, M_2
>
> QUESTION: $L(M_1) \subseteq L(M_2)$?

Give two proofs, one using the specialization trick, and the other using Exercise 196

***Exercise* 198.** *Applying closure properties* 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.

29 Rice's Theorem

Problems like testing whether a program halts on an input may 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.

In this section we prove a theorem about program properties, Rice's Theorem. This is a monster theorem which expresses the idea that *any non-trivial functional property of programs is undecidable*.

First we have to say carefully what those words mean.

29.1 Functional Properties of Programs

Our goal is to show that any non-trivial functional property of programs is undecidable. As noted earlier, we need to make sure this has a precise meaning.

- The way we capture the notion of *property* of programs mathematically is to consider *sets* of programs. This is standard: the "property" of being red can be identified with the set of red things in the world; the property of being round can be identified with the set of round things in the world; the "property" of being even can be identified with the set of even numbers.
- *Non-trivial* means that we have to be talking about a property that is exhibited by some programs but not by others.
- 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 (which are strings) themselves.

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 property of programs.

29.2 Functional Sets

29.1 Definition. *Let L be a set of strings, considered as programs. We say that L is an* functional set of programs *if whenever two programs p and q compute the same partial function, then $p \in L$ if and only if $q \in L$.*

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 $q \notin L$.

Another way to define the notion of functional set of programs is as follows. Let $\mathcal{F}$ be a set of partial computable *functions.* Let L be the set of programs that compute some function in $\mathcal{F}$. Then L is a functional set of programs. And every functional set of programs arises in this way.

We will allow ourselves to simply use the phrase "functional set" in this section.

For the sake of gaining intuition, let's be informal for a minute and 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.

29.2 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 that correctly perform prime factorization.
4. The set of all programs that return an answer on all inputs.

5. The set of all programs that return an answer on input "010100".
6. The set of programs computing increasing functions (with respect to lexicographic order on strings).

In the first three examples above our functional set L was the set of all programs computing a certain single function. In the latter three examples the functional set L was the set of programs for a non-singleton *set* of functions.

29.2.1 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 accepts its own code as input, and there could easily be some other program that accepts the things as q accepts, but doesn't happen to accept itself.

For a concrete example, suppose q is a program that accepts 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 character at the end of the program). Now let q' be a program also that accepts 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.

29.3 What do we mean by "non-trivial"?

Here are two rather dumb properties. But they *are* functional properties.

- $L = \emptyset$, 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.

29.3 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.

29.4 The Theorem

We are now ready to state Rice's Theorem, which addresses the question: *when are decision problems about 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.

29.4 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. First, let p_0 be the following program: `''while true do ;''`. Certainly p_0 computes the empty function.

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 the proof below on the language $\overline{L}$ and show *it* to be undecidable). Note that it is just here that we have used the fact that it is not the set of all programs.

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.

Summarizing the setup:

- p_0 computes the empty function and is not in L
- p_1 computes some non-empty function and is in L

Now, in the usual way, let us suppose, for the sake of contradiction that we had a decision procedure D_L for membership in L. With the help of D_L, we can build a decision procedure D for membership in SelfHalt. Here it is.

```
on input w,
    1. build the following program p
```

```
        On input x:
        simulate w on w;
        simulate $p_1$ on x
```

2. *return* $D_L[p]$.

We want now to show that D returns 1 on an input w if and only if $w \in \mathsf{SelfHalt}$.

- Suppose w is in SelfHalt. Then the inner program p does the same things as p_1 does (after the initial simulation of w on w, which takes time, but will halt, so this does not change the fact that the subsequent computation is just that of p_1). So the new program computes the same function as p_1. This means that the inner program is in L, since $p_1 \in L$. And so $D_L[p]$ will return 1, which is to say that D returns 1.
- Suppose w is not in SelfHalt. Then on any input the new program will loop forever. So the new program computes the empty function, that is, the same function as p_0. This means that the new program is not in L, since $p_0 \notin L$. And so $D_L[p]$ will return 0, which is to say that D returns 0.

So there cannot be such a decision procedure D_L for L, otherwise there would be a decision procedure for SelfHalt. ///

Rice's Theorem is an all-purpose tool for showing properties 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: 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.

29.5 Exercises

***Exercise* 199.** For each language L below:

- say whether or not L is a functional set of programs
- say whether or not L is decidable

1. $\{p \mid 010 \in L(p)\}$.
2. $\{p \mid p \text{ has even length}\}$

3. $\{p \mid L(p)$ has at most 10 strings$\}$.
4. $\{p \mid L(p)$ has at least 10 strings$\}$.
5. $\{p \mid L(p) = \Sigma_2^*\}$.
6. $\{p \mid L(p) \neq \emptyset\}$.
7. $\{p \mid L(p)$ is regular$\}$.
8. $\{p \mid p$ accepts $p.\}$
9. $\{p \mid \mathsf{pfn}(p)$ is the identity function$\}$.
10. $\{p \mid \mathsf{pfn}(p)$ is a constant function$\}$.
11. $\{p \mid p$ is the shortest program computing $\mathsf{pfn}(p)\}$.
12. $\{p \mid L(p) = L(q)\}$, where q is a fixed program that returns 0 on odd length strings and loops forever on even-length strings
13. $\emptyset$.
14. $\{p \mid L(p) = \emptyset\}$
15. $\{p \mid p$ halts on all inputs. $\}$
16. $\{p \mid L(p)$ is semidecidable. $\}$

    (Sort of a trick question, mostly here to provide context for the next question...)
17. $\{p \mid L(p)$ is decidable. $\}$
18. $\{p \mid$ there is some shorter program q computing the same function as $p.\}$

***Exercise* 200.** 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).

30 Semidecidable Languages

We can get a lot of insight into decidability, and especially a lot of insight into certain undecidable problems, by widening our focus and looking into a generalization of decidability.

Remember that a language K is *decidable* if it is $L(p)$ for some decision procedure p. Let's relax that condition a bit.

30.1 Definition. *A language L is* semidecidable *if it is $L(p)$ for some program p.*

The difference between decidable and semidecidable languages is the difference between decision procedures and general programs. To be decidable is to to be the language of a decision procedure. To be semidecidable is to to be the language of any program at all.

To be more explicit:

L is semidecidable there is a program p such that, for all strings x,

$$\begin{aligned} &x \in L \textit{ implies } p[x] \downarrow 1 \\ &x \notin L \textit{ implies either } p[x] \downarrow y, \textit{ with } y \neq 1, \textit{ or } p[x] \uparrow \end{aligned}$$

The following observation is trivial but it is important enough that we call it a theorem.

30.2 Theorem. *If L is decidable then L is semidecidable.*

Proof. Follows right from the definitions: just observe that every decision procedure is a program. ///

The converse of this statement is not true: there are semidecidable languages that are not decidable. We already know an example: the Halting Problem is semidecidable. In fact it is in some sense the canonical example of a semidecidable language.

30.3 Theorem. *The Program Halting Problem is semidecidable.*

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). ///

So now we know that semi decidability differs from decidability.

30.4 Corollary. *Semidecidability does* not *imply decidability.*

Proof. We showed earlier that the Halting Problem is not decidable. ///

Important!

If we ever say that a language A is semidecidable , this should not be taken to imply that A is not decidable. If all one says is "A is semidecidable " then this means "A is semi-decidable; it may or may not also be decidable."

Semidecidable Decision Problems

Semidecidability is a property of a *language.* But in the usual way, since decision problems and languages are different ways to describe the same underlying idea, we will often refer to a decision problem as being semidecidable, when what we mean is that the associated language is a semidecidable language.

For example:

30.5 Example. The following problem is semidecidable.[15]

Halt On λ

INPUT: A C program p

QUESTION: Does p terminate normally on input λ?

[15]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.

What we mean here is that the language

$$\{p \mid p \text{ is a program that terminates normally on } \lambda\}$$

is a semidecidable 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 $\lambda$;
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.

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 (Section 29) 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.

30.6 Example. The language

$$\{G \mid G \text{ is an ambigous context-free grammar}\}$$

is a semidecidable language.

This is what Example 26.8 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 26.8 shows the weaker fact that it is semidecidable, but 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.

The Dovetailing Trick

Before giving some 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 $p_1, p_2, \dots p_k$ and a sequence inputs $x_1, x_2, \dots$, 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.[16]

When we refer to a "dovetailing" computation of $p_1, p_2, \dots p_k$ on inputs $x_1, x_2, \dots$ we mean the following. It is a (potentially infinite) computation that that runs in stages. Specifically:

```
At each stage $s$,
run $p_1[x_1], p_1[x_2], \dots, p_k[x_s]$ each for $s$ steps,
or until it halts, whichever comes first.
```

[16]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.

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.

30.7 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 $\mathsf{pfn}\, p \neq \emptyset$?

Written as a language:

$$\mathsf{NonEmp} \overset{\text{def}}{=} \{p \mid p \text{ is a program and } \mathsf{pfn}\, p \neq \emptyset\}$$

Proposition 28.6 we showed this to be to be undecidable. Here we show it to be semidecidable. 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. ///

30.1 Decidable versus Semidecidable

What's the difference between semidecidable and decidable? We already observed that if a language A is decidable then A is semidecidable.

The previous section showed that the converse is not true. But there *is* in fact an interesting relationship between decidable and semidecidable. This is the content of the next section.

30.1.1 Semidecidable plus Co-Semidecidable Implies Decidable

Complementation is at the heart of the difference between decidable and semidecidable . First, an easy result.

30.8 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 $L(p') = \overline{A}$. 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!). That much shows that $\overline{A}$ is semidecidable ; but we have furthermore that p' halts on all inputs, just because p did. 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 accepted 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*).

Now we can state the most important fact relating decidable and semidecidable .

30.9 Theorem. *A is decidable if and only if both A and $\overline{A}$ are semidecidable .*

Proof. Suppose A is decidable. Then certainly A is semidecidable . Furthermore we have that $\overline{A}$ is decidable as well, by Theorem 30.8. This implies that $\overline{A}$ is semidecidable . This proves the left-to-right direction of the theorem.

Now suppose that A is semidecidable and $\overline{A}$ is semidecidable . In order to show that A is decidable we will build a program p which halts on all inputs and accepts 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 $w$;
dovetail $p_1[w]$ and $p_2[w]$
If $p_1$ accepts $w$ halt and return (1).
If $p_2$ accepts $w$ halt and return (0).
```

To see that p performs as required first note that it accepts the same strings as p_1 and so $L(p)$ is indeed A. We need to see that it returns 0 or 1 on all inputs: but this follows from the fact that for any w, w is either in A or in $\overline{A}$ and so either p_1 or p_2 will return 1 on w, which in turn will cause p to halt with the right answer on w. ///

Some Languages Are Not Even Semidecidable

30.10 Example (Program Emptiness Reconsidered)**.** Recall that $\mathsf{Emp} = \{p \mid p \text{ is a program with } L(p) = \emptyset\}$ We showed that this language in not decidable, in Section 28.2.4. (One could also use Rice's Theorem to show this.)

But the complement of Emp is NonEmp, which we just showed to be semidecidable.

This implies that Emp is not only not decidable, it is not even semidecidable. Since: if it were then, give that its complement NonEmp *is* semidecidable, 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 semidecidable, is to show that the complement, $\overline{X}$ is semidecidable.

Finally, you might be wondering if things can be even worse than that, namely, whether there can be languages X that are not semidecidable, yet whose complements are not semidecidable either. The answer is yes, and in fact there is a natural example: the language consisting of the always-halting programs. But we don't have the tools to prove that fact here.

30.2 Closure Properties of the Semidecidable Languages

The semidecidable languages are closed under many set-theoretic operations, but since semidecidable languages are accepted by programs which are not guaranteed to halt, the proofs are more delicate than the ones for decidable languages.

Note that we already know one closure property that *fails:* complementation. Namely, if language L is semidecidable then it is not necessarily true that $\overline{L}$ is semidecidable.

The next theorem collects some results which *are* true, however.

30.11 Theorem. *The semidecidable languages are closed under the operations of*

1. *intersection,*
2. *union,*
3. *concatenation, and*
4. *asterate (the star operation).*

Proof. 1. Suppose A and B are semidecidable languages; we wish to show that $A \cap B$ is semidecidable. 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 are accepting, then return (1), else return
(0).
```

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 return (1) (it may fail to halt or it may halt with a different return value). This is clear from inspection of the code.

2. Suppose A and B are semidecidable languages; we wish to show that $A \cup B$ is semidecidable. 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 acceptance,
    then return (1)
// If we get here we have failed...
return (0).
```

   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 accepting w precisely if at least one of p_A accepts w or p_B accepts w.

3. Next is concatenation. Suppose A and B are semidecidable languages; we wish to show that AB is semidecidable. 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 are accepting, then return
        (1)
// If we get here we have failed...
return (0).
```

   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 $w = w_1w_2$, 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 $w = w_1w_2$, 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 asterate. Suppose A is a semidecidable language; we wish to show that A^* is semidecidable. 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 $w_1, w_2, \dots w_n$
such that $w_1w_2 \dots w_n = w$:
    run $p_A$ on $w_1$;
    run $p_A$ on $w_2$;
    ...
    run $p_A$ on $w_n$;
    if each of these runs are accepting, then return (1)
// If we get here we have failed...
return (0).
```

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 semidecidable then $\overline{A}$ is semidecidable. If it *were* true that whenever A were semidecidable then $\overline{A}$ were semidecidable as well, then it would follow that whenever A is semidecidable then in fact A is decidable. And that is certainly not true.

30.3 Exercises

***Exercise* 201.** In this exercise we will develop different proofs that the decidable sets are closed under intersection and union. They are not *better* proofs than the direct ones in the text, but they will be good proof-practice for you.

In the proofs 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.

1. 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 semidecidable, and (ii) $\overline{A \cup B}$ is semidecidable.

   To prove (i) : Since A and B are decidable, we have A and B are semi-decidable. We have established that $A \cup B$ is semidecidable.

   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.* ///

2. To prove: If A and B are decidable then $A \cup B$ is decidable.

   ... use the same ideas a in the previous part ...

***Exercise* 202.** Let L_1 and L_2 be languages such that

(i) L_1 and L_2 are each semidecidable,

(ii) $L_1 \cup L_2 = \{0,1\}^*$, and

(iii) $L_1 \cap L_2 = \emptyset$.

We know that this implies that both L_1 and L_2 are decidable.

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.

***Exercise* 203.** *Semidecidable partition* Suppose $L_1, L_2, \ldots L_k$ are semidecidable 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 semidecidable.

***Exercise* 204.** Define K to be $\{p \mid \text{ for some input } x,\ p[x] \downarrow \lambda\}$. Show that K is semidecidable.

***Exercise* 205.** *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 semidecidable but not decidable.
3. L is decidable and $\overline{L}$ is not semidecidable.
4. L is semidecidable but not decidable and $\overline{L}$ is decidable.
5. L is semidecidable but not decidable and $\overline{L}$ is semidecidable but not decidable.
6. L is semidecidable but not decidable and $\overline{L}$ is not semidecidable.
7. L is not semidecidable and $\overline{L}$ is decidable.
8. L is not semidecidable and $\overline{L}$ is semidecidable but not decidable.
9. L is not semidecidable and $\overline{L}$ is not semidecidable.

***Exercise* 206.** *Semidecidable splitting* Suppose that L is semidecidable but not decidable. (So $\overline{L}$ is not semidecidable...) Consider the language

$$L' \overset{\text{def}}{=} \{0x \mid x \text{ is in } L\} \cup \{1x \mid x \text{ is not in } L\}$$

Can you say for certain (without knowing more about L) whether L' is decidable, semidecidable, or non-semidecidable? Justify your answer.

***Exercise* 207.** *Semidecidable subset*

1. Prove or disprove: If L is semidecidable and $K \subseteq L$ then K is semidecidable.
2. Prove or disprove: If L is semidecidable and $L \subseteq K$ then K is semidecidable.

***Exercise* 208.** 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.

31 Enumerability

Sometimes semidecidable 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 "semidecidable" clarifies things.

31.1 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 character, 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.)

31.1 Enumerability is Equivalent to Semidecidability

The main theorem of this section is that a language is semidecidable if and only if it is enumerated by some program. (Hence the traditional name recursively enumerable is not so dumb after all.)

31.2 Theorem. *A language $A \subseteq \Sigma_2^*$ is semidecidable if and only if there is a computer program which enumerates A.*

The other direction is slightly tricky. Suppose A is semidecidable, 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 30.

Proof of the Theorem. For the "if" direction: Suppose there is a program e enumerating A. Here is a program p that semi-decides e.

```
on input $x$;
start $e$ and watch the output;
if and when $x$ ever appears on the output of $e$, return (1).
```

For the other direction, suppose that p that semi-decides A. Here is a program e which enumerates A. Let $x_0, x_1, \dots$ be an enumeration of all the string in Σ_2^*,

```
dovetail $p[x_0], p[x_1], p[x_2], \dots$;
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?

31.2 Exercises

***Exercise* 209.** *Enumerations without repetitions*

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.

***Exercise* 210.** *Enumerations and decidability I*

We know that if A is decidable then it is semidecidable, 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.

***Exercise* 211.** *Enumerations and decidability II*

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.)

***Exercise* 212.** Consider the following claim.

If $A_1, A_2, \ldots$ is a countably infinite set of semidecidable languages, then their union $A_1 \cup A_2 \cup \ldots$ is a semidecidable language.

1. Show that this claim is false.

   Hint. This is easy: remember that any singleton set is a semidecidable language!

2. What is wrong with the following supposed "proof" of the claim?

   Each A_1 is semidecidable, 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.

32 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 semidecidable.

32.1 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.

32.2 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.

$$\mathsf{Term} = \{p \mid p \text{ halts on all inputs }\} \quad \text{is not semidecidable.}$$

Proof. Recall that a language A is semidecidable 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.

///

33 Reducibility

We have done several proofs structured as follows. To show some language X to be undecidable, we (i) first dreamed up some langage U that we knew to be undecidable, then (ii) gave an argument if *if* X were decidable then U would be decidable. This contradiction establishes that X cannot be decidable after all.

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 the beginnings of such a formalization.

The technique is called *reducibility*.

33.1 Definition. *A language A is* m-reducible *to language B, written $A \preceq_m B$, if there is a computable terminating function $f : \Sigma_2^* \to \Sigma_2^*$ such that for every w,*

$$w \in A \text{ if and only if } f(w) \in B$$

33.2 Example. For any language A we have $A \preceq_m A^R$. The function is simply this:

$$f(x) \stackrel{\text{def}}{=} x^R$$

It is clear that f is terminating and computable, and that $x \in A$ if and only if $f(x) \in A^R$.

33.3 Example. Let $A = \{w \mid |w| \text{ is even}\}$, let $B = \{w \mid |w| \text{ is odd}\}$. Then $A \preceq_m B$. Here is a f that works: $f(x) = x1$. [that is, append a "1"]

33.4 Example. Let $A = \{a^n b^n c^n \mid n \geq 0\}$, let $B = \{a^{2n} b^n \mid n \geq 0\}$. Then $A \preceq_m B$. 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

33.5 Check Your Reading. *Show that in Example 33.4 we also have $B \preceq_m A$. (This is not typical!)*

The next easy observation is the crucial fact about reducibility. It is our universal tool for proving things undecidable.

33.6 Theorem. *Suppose A is m-reducible to B. If B is decidable then A is decidable.*

Proof. Suppose that $A \preceq_m B$ via function f; let p_f be a program that computes f. Now let p 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 $p$ on the result
(return that answer)
```

This halts on all w since f are p_f are terminating . 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 terminating , *i.e.* that p_f always returns, 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.

33.7 Corollary. *Suppose A is m-reducible to B. If A is undecidable then B is undecidable.*

Corollary 33.7 is an invaluable tool for showing languages to be undecidable.

33.8 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 $\mathsf{SelfHalt}$. (Remember that $\mathsf{SelfHalt}$, 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 it suffices—by Theorem 33.6—to prove $\mathsf{SelfHalt} \preceq_m \mathsf{SelfHalt}^R$. For that it suffices to construct a computable terminating 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

33.9 Example. For any A, A is undecidable if and only if A^R is undecidable.

Since: If A is undecidable then $A \preceq_m A^R$ shows that A^R is undecidable. If A^R is undecidable then $A^R \preceq_m (A^R)^R = A$ shows that A is undecidable.

33.10 Example. Let A be decidable and let B be any language which is not $\emptyset$ and which is not Σ_2^*. Then $A \preceq_m B$.

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 terminating 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.

33.1 Reducibility and Semidecidability

In fact we can use reducibility to tell us things about semidecidability as well.

33.11 Theorem. *Suppose A is m-reducible to B. If B is semidecidable then A is semidecidable.*

Proof. Suppose that $A \preceq_m B$ 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 33.6. 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 semidecidable. ///

The following is immediate from the theorem.

33.12 Corollary. *Suppose A is m-reducible to B. If A is not semidecidable then B is not semidecidable.*

33.2 Two Potential Gotchas

The direction of the reduction matters. Suppose you have $A \preceq_m B$. 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 terminating 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 $A \preceq_m B$ then you *cannot* say something like, "the definition f on input x is: if $x \in A$ then [blah, blah ...] but if $x \notin A$ then [blah, blah, blah ...]". This pseudocode will be bogus *unless* you know A is decidable!

33.3 Transitivity of $\preceq_m$

This result is easy but it is used constantly.

33.13 Lemma. *Suppose $A \preceq_m B$ and $B \preceq_m C$. Then $A \preceq_m C$.*

Proof. Let f witness $A \preceq_m B$ and let g witness $B \preceq_m C$. Claim: the function $(g \circ f)$ witnesses $A \preceq_m C$.

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 $(g \circ f)(w) \in C$. ///

33.4 The Acceptance Problem Revisited

Recall this decision problem:

> *Program-Acceptance*
>
> INPUT: A program p and an input x
>
> QUESTION: Does p accept x?

Here is Acc, the language corresponding to this decision problem:

$$\mathsf{Acc} \overset{\text{def}}{=} \{\langle p,x\rangle \mid \text{program } p \text{ accepts input } x.\}$$

We showed earlier by a direct argument that this language is not decidable.

With the technique of reducibility at our disposal, we can show a strong connection between the Halting Problem and the Acceptance Problem.

33.14 Theorem. $\mathsf{Halt} \preceq_m \mathsf{Acc}$ *and* $\mathsf{Acc} \preceq_m \mathsf{Halt}$.

Proof. We show the first and leave the second as an exercise (Exercise 215).

For the first: we construct a computable terminating function f witnessing $\mathsf{Halt} \preceq_m \mathsf{Acc}$. The subtlety is that we will be building a function f that returns strings *that we will take seriously as programs and inputs.* Roughly speaking, the function f involves a program transformer.

The key idea is to note that if p is any program then we can build a related program $\mathsf{make\text{-}acc}(p)$ that behaves as follows:

> on any input y,
> $\mathsf{make\text{-}acc}(p)$ simulates p on y,
> if p halts on y, then $\mathsf{make\text{-}acc}(p)$ returns 1 (no matter what p returned).

That is:

p(y)	(make-acc(p))(y)
halt and acc	acc
halt and rej	acc
loop	loop

Of course if p fails to halt on y then $\mathsf{make\text{-}acc}(p)$ will fail to halt as well, since we are just simulating. But in any event it should be clear that if we are given code for p we can build code for $\mathsf{make\text{-}acc}(p)$.

So here is the definition of the function f: on input of the form $\langle p,y\rangle$ return $\langle \mathsf{make\text{-}acc}(p),y\rangle$. On strings not of the form $\langle p,y\rangle$, f is the identity.

$$f(w) = \begin{cases} \langle \mathsf{make\text{-}acc}(p),x\rangle & \text{if } w \text{ is of the form } \langle p,x\rangle \\ w & \text{otherwise} \end{cases}$$

Clearly f is a computable terminating function. To see that it witnesses $\mathsf{Halt} \preceq_m \mathsf{Acc}$ we just observe that for any w

$$w \in \mathsf{Halt} \text{ if and only if } f(w) \in \mathsf{Acc}$$

Thus $\mathsf{Halt} \preceq_m \mathsf{Acc}$ via f. ///

33.5 Why the Acceptance Problem is Special

An informal way to describe the next result is that it shows that the Acceptance Problem is a "hardest" semidecidable problem: everything semidecidable reduces to it.

Anyway, we will prove directly in this section that if A is any semidecidable language then $A \preceq_m \mathsf{Acc}$. Since $\mathsf{Acc} \preceq_m \mathsf{Halt}$ it will also follow that $A \preceq_m \mathsf{Halt}$; but it is easier to prove $A \preceq_m \mathsf{Acc}$.

The proof is embarrassingly simple.

33.15 Theorem. *Let A be semidecidable. Then $A \preceq_m$* Acc.

Proof. Let p be a program with $L(p) = A$. Define the reduction function f to be defined simply as

$$f(x) = \langle p,x\rangle$$

It is clear that f is terminating and computable, and that $x \in A$ if and only if $f(x) = \langle p,x\rangle \in \mathsf{Acc}$ ///

Another title for this section could be: why the Halting problem is special. Here's why. Since $\mathsf{Acc} \preceq_m \mathsf{Halt}$, we conclude immediately that for every semidecidable language A, we have $A \preceq_m \mathsf{Halt}$. It just so happens that the proof goes through more smoothly if one uses that Acceptance Problem as the thing to be reduced to directly. But the phrase "*The Halting Problem*" has become a catch-phrase and indeed it is standard to hear people say that "any semidecidabale language can be reduced to the Halting Problem".

By the way, if you are familiar with the complexity class **NP** and the notion of **NP**-completeness, please observe that Theorem 33.15 says that the language Acc is "semidecidable complete" in exactly the same sense that we speak of a language being **NP**-complete. The notion of "semidecidable complete" came first, though, in the 1930's!

33.6 Emptiness Is Not Semidecidable

That result about Program NonEmptiness now gives us more information about Program *Emptiness.*

33.16 Theorem. Emp *is not semidecidable.*

Proof. We have seen that the complement of Emp, namely, NonEmp $\overset{\text{def}}{=} \{p \mid \mathsf{dom}(\mathsf{pfn}(p)) \neq \emptyset\}$, is semidecidable. So if Emp were semidecidable, we could conclude that it was actually decidable, using the fundamental fact (Theorem 30.9) relating decidable and semidecidable. ///

33.7 Reducibility for Complexity

The notion of reducibility, first formulated for studying undecidability as we use it here, can be adapted to study *computational complexity* as well. One defines a variant of 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 reducibility is the most common technique for establishing complexity results. You can read more about this in any textbook about complexity.

33.8 Exercises

***Exercise* 213.** Prove that the following language is not decidable.

$$\{0p \mid p \in \mathsf{SelfHalt}\}$$

This is the set of all programs in SelfHalt but with the single character "0" prepended to each one.

***Exercise* 214.** These are about reading the definition of $\preceq_m$ carefully.

1. Suppose $A \preceq_m \emptyset$. What can you say about A? Prove your answer.
2. Suppose $A \preceq_m \Sigma_2^*$. What can you say about A? Prove your answer.
3. Suppose $\emptyset \preceq_m A$. What can you say about A? Prove your answer.
4. Suppose $\Sigma_2^* \preceq_m A$. What can you say about A? Prove your answer.

***Exercise* 215.** Finish the proof of Theorem 33.14, by showing $\mathsf{Acc} \preceq_m \mathsf{Halt}$.

***Exercise* 216.** Prove or disprove: For all A and B, $A \preceq_m (A \cap B)$

***Exercise* 217.** The relation $\preceq_m$ is almost a partial order. We proved that it is transitive.

1. Prove that $\preceq_m$ is reflexive: we always have $A \preceq_m A$
2. Show by example that $\preceq_m$ is not anti-symmetric: $A \preceq_m B$ and $B \preceq_m A$ does not imply $A = B$.

This last is the sense in which $\preceq_m$ is not a partial order; relations that are reflexive and transitive but not necessarily anti-symmetric are called "preorders".

***Exercise* 218.** Prove that If $A \preceq_m B$ then $\overline{A} \preceq_m \overline{B}$. (super easy.)

***Exercise* 219.** Suppose A is semidecidable and $A \preceq_m \overline{A}$. Prove that A is decidable.

***Exercise* 220.** Consider the question: is it true that for every A we have $A \preceq_m \overline{A}$? That seems like a reasonable conjecture. But it's always good to test things on the "extreme" cases... so let's consider $A = \emptyset$. Oops, we see that $\emptyset \preceq_m \Sigma_2^*$ is pretty obviously false.

But that isn't very satisfying, the empty set might be a special degenerate case. So let's now consider the more refined conjecture: *if $A \neq \emptyset$ and $A \neq \Sigma_2^*$ then we have $A \preceq_m \overline{A}$.*

Show that this fails. *Hint.* Recall Exercise 219

***Exercise* 221.** Let *Odd* be the set of all bit strings of odd length. Show that $Odd \preceq_m \mathsf{SelfHalt}$

Hint. Remember that we emphasized that (in general) in proving $X \preceq_m Y$ 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 exercise is to realize that since we are working with *Odd*, clearly a decidable language, our function f to reduce *Odd* to $\mathsf{SelfHalt}$ can, if we wish, involve a test for membership in *Odd*.

***Exercise* 222.** 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.

***Exercise* 223.** This exercise is here mainly to be useful in exercise 224

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 $Y \preceq_m X$.

***Exercise* 224.** Prove or disprove: If A is not decidable then A_{odd} is not decidable. (Compare Exercise 222)

***Exercise* 225.** 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 $\mathsf{SelfHalt}\,\mathsf{SelfHalt}$

***Exercise* 226.** 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 $A \preceq_m 2A$.
2. Explain why it is *not* true that $2\Sigma_2^* \preceq_m \Sigma_2^*$
3. Prove that if $A \neq \Sigma_2^*$ we have $2A \preceq_m A$.

   Hint. You will need to identify a particular element w_0 known to not be in A.

***Exercise* 227.** An interesting contrast to exercise 226. Do we always have $A \preceq_m AA$? The answer is no. In fact it is possible to have

- A undecidable, but

- AA decidable

Give an example.

***Exercise* 228.** Give a direct proof of the fact that *for any semidecidable language A, $A \preceq_m$* Halt.

As we noted, this follows from the facts that $A \preceq_m$ Acc and Acc $\preceq_m$ Halt but it is good practice for you to construct a reduction function directly.

34 Post's Correspondence Problem

Here is a puzzle. Consider the following "dominos", which are just pairs of bit strings that we have chosen to write one on top of the other.

$$\begin{pmatrix} 1 \\ 111 \end{pmatrix} \qquad \begin{pmatrix} 10111 \\ 10 \end{pmatrix} \qquad \begin{pmatrix} 10 \\ 0 \end{pmatrix}$$

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

$$\begin{pmatrix} 10111 \\ 10 \end{pmatrix}\begin{pmatrix} 1 \\ 111 \end{pmatrix}\begin{pmatrix} 1 \\ 111 \end{pmatrix}\begin{pmatrix} 10 \\ 0 \end{pmatrix}$$

You can check that the first row and the second both make the string 101111110.

34.1 Examples

34.1 Example. Here is another such problem. We take our dominos to be

$$\begin{pmatrix} 1 \\ 111 \end{pmatrix} \qquad \begin{pmatrix} 10 \\ 10111 \end{pmatrix} \qquad \begin{pmatrix} 10 \\ 00 \end{pmatrix}$$

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 dominos, 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.

34.2 Example. Suppose our dominos are

$$\begin{pmatrix} 100 \\ 11 \end{pmatrix} \qquad \begin{pmatrix} 10 \\ 1011 \end{pmatrix} \qquad \begin{pmatrix} 010 \\ 011 \end{pmatrix}$$

Is there a solution?

The answer here is again no. One way to see this is that a necessary condition for a PCP instance to have a solution is that at least one of the dominos has one string that is a prefix if the other (you have to start with such a domino). It's also necessary that at least one of the dominos has one string that is a postfix of the other (you have to end with such a domino). The above problem pases the first test but not the second.

34.3 Example. One more example. Suppose our dominos are

$$\binom{0}{1} \qquad \binom{1}{011} \qquad \binom{011}{0}$$

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 dominos, and that is the shortest solution.[17]

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 dominos, 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. Our previous examples have all been over bitstrings. But the problem makes sense over any alphabet Σ, and it is sometimes convenient to use other alphabets, so we build that generality into the definition.

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.

[17]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

34.2 Undecidability of PCP

34.4 Theorem. *Post's Correspondence Problem is undecidable.*

Proof. (Idea) One gives a complex construction that shows that, if one had a decision procedure for the PCP, one could use it as a subroutine to give a decision procedure for the Halting Problem. The proof is facilitated by working with the Halting Problem for Turing machines, as opposed to the Halting Problem for programs. So the proof also depends on the fact that any program can be captured by a Turing machine. ///

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. In this set of problems you will play with some PCP instances, develop some intuition for the general problem, and prove a few results about special cases.

34.3 Exercises

***Exercise* 229.** *(Warm Up)* For each instance of the PCP below, either give a solution or argue why no solution exists.

1. $$\binom{abc}{ab}\binom{aca}{ca}\binom{b}{acab}$$

2. $$\binom{abc}{ab}\binom{abba}{c}\binom{c}{ccc}\binom{bbba}{cbbb}\binom{abcc}{aab}$$

3. $$\binom{c}{cb}\binom{b}{aba}\binom{a}{bab}\binom{aba}{b}\binom{bab}{a}$$

4. $$\binom{ab}{a}\binom{aa}{bab}\binom{b}{aa}\binom{ba}{ab}$$

***Exercise* 230.** Show that is a PCP instance $\mathcal{P}$ has a solution, then it has infinitely many solutions. (This is easy)

***Exercise* 231.** True or false: a necessary condition for a PCP instance to have a solution is that there be one domino whose top string is longer than its bottom string and another domino whose bottom string is longer than its top string. (Ignore trivially solvable PCP instances in which some domino has identical top and bottom strings.)

***Exercise* 232.** Show that for the purposes of investigating which PCP problems have solutions we may without loss of generality restrict attent to the alphabet $\{0,1\}$. Specifically, show that we can algorithmically transform any PCP problem $\mathcal{P}$ over an arbitrary (finite of course) 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. You can do this without changing the number of dominos defining the problem.

***Exercise* 233.** Show that the PCP is decidable if the alphabet has only one symbol. (Give pseudocode for a decision procedure.)

***Exercise* 234.** 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 dominos. 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 $\mathcal{P}$ is a *solvable* PCP instance, let $l(\mathcal{P})$ 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.

$$\mathit{bound}(n) \overset{\text{def}}{=} \text{the largest value of } l(\mathcal{P}) \text{ among all the solvable PCP instances of length } n$$

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.

***Exercise* 235.** *(Hard.)* Show that the subclass of PCP instances that have only 2 dominos is decidable. (Give pseudocode for a decision procedure.)

Feel free to use the result of Exercise 232

By the way, it is known that the subclass of PCP instances with 5 or more dominos is undecidable. Decidability is unknown for $k = 3$ or 4.

This is a hard problem. It was first proved in [EKR82]

35 Undecidability Results about Context-Free Grammars

35.1 PCP meets CFG

If $\mathcal{P}$ is an instance of the PCP there are two context-free grammars easily derived from $\mathcal{P}$, 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 $\Sigma \cup \{d_1, d_2, \dots, d_n\}$, 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 $\mathcal{P}$ has dominos $\binom{x_1}{y_1}, \dots, \binom{x_n}{y_n}$ 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 $\mathcal{P}$, 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 $G_T^{\mathcal{P}}$ and $G_B^{\mathcal{P}}$, 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.

35.1 Check Your Reading. *Pick a PCP instance and write down the grammars. Play some dominos. Then write down the derivation in G_T and the derivation in G_B that correspond to that play.*

Here's a little technical point we need in Section 35.3.

35.2 Lemma. *For every PCP instance $\mathcal{P}$, 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 characters 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. ///

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. We will use this fact in Section 35.4.

35.3 Lemma. *For every PCP instance $\mathcal{P}$, 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]. ///

The point to defining G_T and G_B is that they make a connection between the world of PCP and the world of grammars, which we will exploit in the next few sections.

35.2 Context-Free Language Intersection

Here we show that it is undecidable whether two given context-free grammars generate any strings in common.

35.4 Lemma. *Let $\mathcal{P}$ be a PCP instance. Then $\mathcal{P}$ has a solution if and only if $L(G_T) \cap L(G_B) \neq \emptyset$.*

Proof. If $\mathcal{P}$ has a solution, this means that there is a sequence $s = i_1, \dots, i_m$ 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 $i_1, \dots, i_m$, 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 an undecidability result immediately.

35.5 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 $\mathcal{P}$ of the PCP;
- Build G_T and G_B from $\mathcal{P}$;
- 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 35.4. 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. ///

35.3 Context-Free Grammar Ambiguity

Here we show that it is undecidable whether a given context-free grammar is ambiguous.

Let $\mathcal{P}$ be a PCP instance. We create another grammar G_A from $\mathcal{P}$. This time we start with G_T and G_B derived from $\mathcal{P}$ as before, but now combine them into one grammar that generates the union of their languages. Namely, add one new variable S, and add

$$S \rightarrow S_T \mid S_B$$

to the productions for G_T and G_B.

What strings are generated by G_A? Precisely those strings

$$a_1 a_2 \dots a_k d_{i_m} \cdots d_{i_1} \qquad \text{// here the } a_j \text{ are } \Sigma\text{-symbols}$$

such that, if the dominos are played according to the sequence $d_{i_1} \cdots d_{i_m}$, then $a_1 a_2 \dots a_k$ is *either* the string along the top *or* the string along the bottom. This is because the first step in the derivation of $a_1 a_2 \dots a_k d_{i_m} \cdots d_{i_1}$ 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.

35.6 Lemma. *Let $\mathcal{P}$ be a PCP instance. Then $\mathcal{P}$ has a solution if and only if the grammar G_A is ambiguous.*

Proof. If $\mathcal{P}$ has a solution, this means that there is a sequence $s = i_1, \dots, i_m$ with

$$x_{i_1} \cdots x_{i_m} = y_{i_1} \cdots y_{i_m}$$

Just as in Lemma 35.4 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 $\mathcal{P}$ has a solution. This is where we need the observation that each of G_T and G_B are unambiguous on their own. 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 35.4 this means that $\mathcal{P}$ has a solution. ///

The proof of the next theorem follows the same pattern as the proof of Theorem 35.5. We have used the same wording as much as possible, to emphasize this.

35.7 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 $\mathcal{P}$ of the PCP;
- Build G_A from $\mathcal{P}$;
- 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 35.6. 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. ///

35.4 Context-Free 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 the asymmetry is even more striking when it come to decision problems: it is undecidable whether a *CFG* generates all strings.

Let $\mathcal{P}$ be a PCP instance. We create yet one more another grammar G_U from $\mathcal{P}$. Here is where we use the fact about G_T and G_B (Lemma 35.3) 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, $L(G_U) = L(G'_T) \cup L(G'_B) = \overline{L(G_T)} \cup \overline{L(G'_B)}$.

35.8 Lemma. *Let $\mathcal{P}$ be a PCP instance. The $\mathcal{P}$ 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 35.4 that $\mathcal{P}$ 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:

$$\begin{aligned} \mathcal{P} \text{ has a solution } & \text{ if and only if } \overline{L(G_T) \cap L(G_B)} \neq \Sigma^* \\ & \text{ if and only if } \overline{L(G_T)} \cup \overline{L(G_B)} \neq \Sigma^* \\ & \text{ if and only if } L(G'_T) \cup L(G'_B) \neq \Sigma^* \end{aligned}$$

which is what we wanted to show.

///

Again the proof of the next theorem follows the same pattern as the proof of Theorem 35.5.

35.9 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 $\mathcal{P}$ of the PCP;
- Build G_U from $\mathcal{P}$;
- 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 35.8 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. ///

35.5 Exercises

***Exercise* 236.** Consider the following two decision problems.

INPUT: A context-free grammar G and a DFA M

QUESTION: Is $L(M) \subseteq L(G)$?

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.

***Exercise* 237.** Show that the following problems is undecidable.

CFG Infinite Intersection

INPUT: Two $CFGs$ G_1 and G_2

QUESTION: Is $L(G_1) \cap L(G_2)$ infinite?

Hint. This isn't hard. Use Exercise 230.

36 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, semidecidable, 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: $\{p/q \mid p,q \in \mathbb{Z}\}$.
- $\mathbb{R}$, 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!

36.1 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 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 $\mathbb{Z}$ or $\mathbb{Q}$, but the answer is “yes” if we ask about $\mathbb{R}$.

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.

36.2 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.

36.2.1 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 $\mathbb{N}$.

36.2.2 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.

36.1 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; . . . , and so on.

Now we can show

36.2 Lemma. *Polynomial solvability over $\mathbb{Z}$ is semidecidable.*

Proof. It will be sufficient, and more convenient, to show that polynomial solvability over $\mathbb{N}$ is semidecidable. 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 are able to prove that a given *decision problem* is undecidable then of course there will be no hope of actually *computing solutions.*

36.2.3 Diophantine Sets

Our main technique will be to construct a connection between polynomials and semidecidable sets. The crucial definition is the next one.

36.3 Definition. *A set $A \subseteq \mathbb{N}$ is a* Diophantine set[18] *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.*

By the way, it makes perfect sense to speak of Diophantine sets of ordered pairs, ordered triples, etc. We just use several variables $x_1, z_2, \ldots$ 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 36.3.

36.4 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 is Diophantine; the polynomial is $z - (x_1 + 2)(x_2 + 2)$

[18] Diophantus was a 3rd century Greek mathematician who studied polynomial equations

More examples are Exercise 243

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 $S = \{n_1, n_2, \ldots\}$ is a set of natural numbers, we will write $\{b_{n_1}, b_{n_2}, \ldots\}$ to refer to the corresponding set of bitstrings.

36.5 Check Your Reading. *Explain why, if a set $S = \{n_1, n_2, \ldots\}$ of natural numbers is Diophantine then the corresponding language $\{b_{n_1}, b_{n_2}, \ldots\}$ is semidecidable.*

The following amazing theorem is the key result for us. The converse of the observation that every Diophantine set is semidecidable 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:

36.6 Theorem (The DPRM Theorem)**.** *A set $S = \{n_1, n_2, \ldots\}$ of natural numbers is Diophantine if and only if the corresponding language $\{b_{n_1}, b_{n_2}, \ldots\}$ is semidecidable.*

Once we know Theorem 36.6 we can draw our undecidability conclusion.

36.7 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 36.1, specifically sub-Section 36.2.1, this will imply that polynomial solvability over $\mathbb{Z}$ is undecidable as well.

Choose a language that is semidecidable 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:

$b_n \in$ SelfHalt if and only if the polynomial $p(n, x_1, \ldots, x_k)$ 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 decidability 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: Exercise 242 asks you to prove that one-variable Diophantine solvability is decidable.
- Polynomial solvability is undecidable for $n \geq 11$.
- Polynomial solvability is decidable for $d = 2$.
- Any polynomial is equivalent to one of degree at most 4. Therefore polynomial solvability is undecidable for $d \geq 4$.
- There exist fixed n and d such that the problem of deciding solvability for polynomials with n variables and total degree d is undecidable. We do not currently have tight bounds for n or d.

36.2.4 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?

36.8 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 36.10.

36.2.5 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?

36.3 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* [19]. 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.

[19]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) $\mathbb{Z}$.

36.3.1 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 36.7

36.9 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 semidecidable, nor co-semidecidable. 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.

36.3.2 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 semidecidable. 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 real numbers are infinte objects. Or, what amounts to the same thing, there are uncountably many real numbers, so we can't list them in a way algorithms can work with them.

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.

36.10 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, and so improving the efficiency of such algorithms is still an area of research.

We immediately get:

36.11 Corollary. *Polynomial solvability over $\mathbb{R}$ is decidable.*

36.3.3 Logical Truth over the Rational Numbers

When it comes to logical truth, the rationals behave more like the intergers than like the reals.

> *Logical Truth over $\mathbb{Q}$*
>
> INPUT: An arithmetic sentence S
>
> QUESTION: Is S true about $\mathbb{Q}$?

36.12 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.

36.4 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 $\mathbb{C}$ istrivial: 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 $\mathbb{C}$.

36.5 Exercises

***Exercise* 238.** If we cared to, we could effectively enumerate all the k-tuples over $\mathbb{Z}$. Give a method to do this.

***Exercise* 239.** Find a polynomial with integer coefficients that roots over $\mathbb{Q}$ but no roots over $\mathbb{Z}$.

Find a polynomial with integer coefficients that roots over $\mathbb{Q}$ but no roots over $\mathbb{Z}$.

***Exercise* 240.**

1. 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 $\mathbf{Z}$ if and only if q has roots in $\mathbb{N}$. What is that q?

2. 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?

***Exercise* 241.** This problem is about one-variable polynomials.

Prove: if p is $c_nx^n + c_{n-1}x^{n-1} + \dots c_1x + c_0$ and a is an integer root of p, then a divides c_0.

Hint. 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.

***Exercise* 242.** Show that the problem of solvability of *one-variable* polynomials over $\mathbf{Z}$ is decidable.

Hint. Use Exercise 241

***Exercise* 243.** Each of the following polynomials $p(z, x_1, \dots x_n)$ defines an easy-to-describe (Diophantine) set $A \subseteq \mathbb{N}$, by

$$A = \{n \in \mathbb{N} \mid \exists x, \dots, x_k : p(n, x_1, \dots x_k) = 0\}$$

Describe each set. You needn't prove your answer, just generate enough instances so that you are confident that you know the set.

1. $(z - 17)(x_1 + 1)$
2. $z + x_1 = 10$
3. $z - 7x_1$
4. $(z - 2x_1)(z - 3x_2)$
5. $z - (2x + 3)(y + 1)$
6. $x_1^2 - z(x_2 + 1)^2 - 1$

   Hint. This one isn't easy. Note for starters that z can't be a perfect square...(why?)

Hint. For several of these you can do this by isolating z and just plugging in values for the x_i. Even when you can't isolate z you can just start plugging in values for the x_i to get intuitions.

***Exercise* 244.**

1. Suppose $p(z, x_1, \dots x_k)$ and $q(z, y_1, \dots y_p)$ are polynomials.

   Explain why, for any n and tuples $(a_1, \dots, a_k)$ and $(b_1, \dots, b_p)$,

   $$p(n, a_1, \dots a_k) = 0 \text{ and also } q(n, b_1, \dots b_p) = 0$$

   if and only if

   $$(p(n, a_1, \dots a_k))^2 + (q(n, b_1, \dots b_p))^2 = 0$$

2. Using this, prove (without quoting Theorem 36.6) that the intersection of two Diophantine sets is Diophantine.

***Exercise* 245.** Prove (without quoting Theorem 36.6) that the union of two Diophantine sets is Diophantine.

Hint. Proceed in the spirit of Exercise 244, that is, find an appropriate way to combine polynomials.

***Exercise* 246.** Prove or disprove: The complement of a Diophantine set is Diophantine.

References

[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.

[Eri] Jeff Erickson. Models of computation. `http://www.cs.illinois.edu/~jeffe/teaching/algorithms`.

[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.

[RR67] Hartley Rogers and H 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.