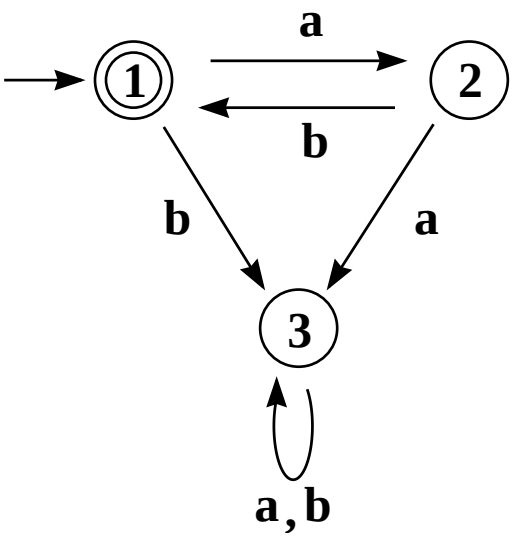
The Theory of Languages and Computation

Jean Gallier
jean@saul.cis.upenn.edu

Andrew Hicks
rah@grip.cis.upenn.edu

Department of Computer and Information Science
University of Pennsylvania

Preliminary notes - Please do not distribute.



Contents

Chapter 1

Automata

1.1 Notation

The following conventions are useful and standard.

$\neg$ stands for "not" or "the negation of".
$\forall$ stands for "for all".
$\exists$ stands for "there exists".
$\ni$ stands for "such that".
s.t. stands for "such that".
$\Rightarrow$ stands for "implies" as in $A \Rightarrow B$ ("A implies B").
$\Leftrightarrow$ stands for "is equivalent to" as in $A \Leftrightarrow B$ ("A is equivalent to B").
iff is the same as $\Leftrightarrow$.

1.2 Proofs

The best way to learn what proofs are and how to do them is to see examples. If you try to find a definition of a proof or you going around asking people what they think a proof is, then you will quickly find that you are asking a hard question. Our approach will be to avoid defining proofs (something we couldn't do anyway), and instead do a bunch so you can see what we mean.
Often students say "I don't know how to do proofs". But they do. Almost everyone could do the following:

Theorem $x = 5$ is a solution of $2x = 10$.
Proof $2 \cdot 5 = 10$.

So in some sense, EVERYONE can do a proof. Things get stickier though if it is not clear what you allowed to use. For example, the following theorem is often proved in elementary real analysis courses:

Theorem $1 > 0$.
Just given that theorem *out of context*, it really isn't clear that there is anything to prove. But, in a analysis course a definition of 1 and 0 is given so that it is sensible to give a proof. In other words the basic assumptions are made clear. One of our goals in the next few sections is to clarify what is to be considered a "basic assumption".

1.3 Set Theory

Most people are introduced to computer science by using a real computer of course, and for the most part this requires a knowledge of only some basic algebra. But as one starts to learn more about about the *theory*

of computer science, it becomes apparent that a kind of mathematics different from algebra must be used. What is needed is a way of stating problems precisely and a way to deal with things that are more abstract than basic algebra. Fortunately, there is a lot of mathematics available to do this. In fact, there are even different choices that one can use for the foundations, although the framework that we will use is used by almost everyone. That framework is classical set theory as was invented by Cantor in the 19th century.
We should emphasize that one reason people start with set theory as their foundations is that the idea of a set seems pretty natural to most people, and so we can communicate with each other fairly well since we seem to all have the same thing in mind. But what we take as an axiom, as opposed to what we construct, is a matter of taste. For example, some objects that we define below, such as the ordered pair or the function, could be taken as part of the axioms. There is no universal place where one can say the foundations should begin. It seems to be the case though, that when most people read the definition of a set, they understand it, in the sense that they talk to other people about sets and seem to be talking about the same thing.

Definition 1.3.1 A **set** is a collection of objects. The objects of a set are called the **elements** of that set.

Notation If we are given a set A such that x is in A iff x has property P, then we write

$$A = \{x \mid x \text{ has property } P\},$$

or some obvious variation on the above. If x is an element of A then we may write $x \in A$.

Example 1.3.2 Thus,

$$P = \{x \mid x \text{ is a prime number}\}$$

is the set of prime numbers. Similarly, if we want to only look at prime numbers of the form $n^2 + 1$ we could write

$$\{x \in P \mid \text{there exists a natural number } n, \text{ such that } x = n^2 + 1.\}$$

A more compact way of denoting this set is

$$\{x \in P \mid (\exists n \in N) \ni (x = n^2 + 1)\}$$

or even simply

$$\{x \in P \mid (\exists n \in N)(x = n^2 + 1)\}.$$

Although these last two ways of defining the above set is more compact, if many things are written in this manner it soon becomes hard to read them.
For finite sets we can just list the elements, so the set with the elements 1, 2 and 3 is $\{1, 2, 3\}$. On the other hand to list the numbers from 1 to 100 we could write $\{1, 2, 3, ..., 99, 100\}$.

Notation Some sets of numbers are used so often that they are given their own names. We will use N for the set of natural numbers $\{0, 1, 2, 3...\}$, Z for the set of integers $\{0, 1, -1, 2, -2, ...\}$, Z^+ for the set of positive integers, Q for the set of rational numbers $\{\frac{p}{q} | p, q \in Z, q \neq 0\}$, and R for the set of real numbers. Another common notation is that for any set A of numbers, if n is an integer then $nA = \{na | a \in A\}$. Thus, for example, $2Z$ is the set of even numbers. Incidentally, we have not indicated how to construct these sets, and we have no intention of doing so. One can start from scratch, and define the natural numbers, and then the integers and the rationals etc. This is a very interesting process, and one can continue it in many different directions, defining the real numbers, the p-adic numbers, the complex numbers and the quaternions. Some of these objects have applications in computer science, but the closest we will get to the foundational aspects of numbers systems is when we study the natural numbers below.

Definition 1.3.3 If A and B are sets and for every $x \in A$ we have that $x \in B$ then we say that A **is a subset of** B and we write $A \subset B$.

Definition 1.3.4 We consider two sets to be **equal** if they have the same elements, i.e. if $A \subset B$ and $B \subset A$. If A and B are equal then we write $A = B$.

It might seem strange to define what it means for two things to be equal. A familiar example of a situation where it is not so clear as to what is meant by equality is how to define equality between two objects that have the same type in a programming language (i.e. the notion of equality for a given data structure). For example, given two pointers, are they equal if they point at the same memory location, or are they equal if they point at memory locations that have the same contents ? Thus in programming one needs to be careful about this, but from now on since everything will be defined in terms of sets, we can use the above definition.

Definition 1.3.5 The **union** of the sets A and B is the set

$$A \cup B = \{x \mid x \in A \text{ or } x \in B\}.$$

More generally, for any set $\mathcal{C}$ we define

$$\cup\mathcal{C} = \{x | (\exists A \in \mathcal{C}) \ni (x \in A)\}.$$

For example, if $A = \{1, 2, 6, \{10, 100\}, \{0\}, \{\{3.1415\}\}\}$ then $\cup A = \{10, 100, 0, \{3.1415\}\}$. There are a number of variants of this notation. For example, suppose we have a set of 10 sets $\mathcal{C} = \{A_1, ..., A_{10}\}$. Then the union, $\bigcup \mathcal{C}$, can also be written as

$$\bigcup_{i=1}^{10} A_i.$$

Lemma 1.3.6 $A \cup B = B \cup A$.

Proof What needs to be shown here ? The assertion is that two sets are equal. Thus we need to show that $x \in A \cup B \iff x \in B \cup A$. This means that we need to do two little proofs: one that $x \in A \cup B \rightarrow x \in B \cup A$ and one that $x \in A \cup B \leftarrow x \in B \cup A$.

$\rightarrow$: If $x \in A \cup B$ then we know that x is in either A or B. We want to show that x is in either B or A. But from logic we know that these are equivalent statements, i.e. "p or q" is logically the same as "q or p". So we are done with this part of the proof.

$\leftarrow$: This proof is very much like the one above.

Definition 1.3.7 The **intersection** of the sets A and B is the set

$$A \cap B = \{x \mid x \in A \text{ and } x \in B\}.$$

Definition 1.3.8 The **difference** of the sets A and B is the set

$$A - B = \{x \mid x \in A \text{ and } x \notin B\}.$$

Definition 1.3.9 By an **ordered pair** (a, b) we mean the set $\{\{a\}, \{a, b\}\}$.

This definition of ordered pair is due to Kuratowski. At this point, a good problem for the reader is to prove theorem 1.3.10. Just as the **cons** operator is the foundation of Lisp like programming languages, the pair is the foundation of most mathematics, and for essentially the same reasons. The following is the essential reason for the importance of the ordered pair.

Theorem 1.3.10 If $(a, b) = (c, d)$ then $a = c$ and $b = d$.

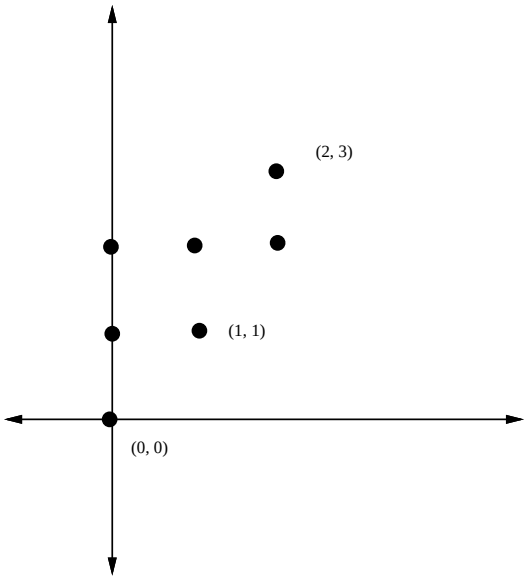


Figure 1.1: Here we see a few of the points from the lattice of integers in the plane

Proof See exercise 4 for a hint.

Notation It is possible to give many different definitions of an n-tuple $(a_1, a_2, ..., a_n)$. For example, we could define a triple (a, b, c) as $(a, (b, c))$ or $((a, b), c)$. Which definition we choose doesn't really matter - the only thing that is important is that the definition implies that if two n-tuples are equal, then their entries are equal. From this point on we will assume that we are using one of these definitions of an n-tuple.

Definition 1.3.11 We define the **Cartesian product** of A and B as

$$A \times B = \{(a, b) \mid a \in A \text{ and } b \in B\}.$$

A convenient notation is to write A^2 for $A \times A$. In general, if we take the product of a set A with itself n times, then we will sometimes write it as A^n.

Example 1.3.12 Probably the most familiar example of a Cartesian product is the plane, $R^2 = R \times R$. This is partly since it was the "first" example, due to Decartes, who invented what we now call **Cartesian coordinates**. The idea of this important breakthrough is to think of the plane as a product. Then the position of a point can be encoded as a pair of numbers. A similar example is the **lattice** $Z \times Z$, which is a subset of $R \times R$. This lattice is simply the set of points in the plane with integer coordinates (see figure 1.1). Can you picture $Z^3 \subset R^3$?

Example 1.3.13 For finite sets, one can "list" the elements in a Cartesian product. Thus $\{1, 2, 3\} \times \{x, y\} = \{(1, x), (1, y), (2, x), (2, y), (3, x), (3, y)\}$. Do you see how to list the elements of a Cartesian product using two "nested loops" ?

Example 1.3.14 For any set A, $A \times \emptyset = \emptyset \times A = \emptyset$.

Definition 1.3.15 A **relation** is a set of ordered pairs. By a **relation on** A we mean a subset of $A \times A$.

Notation If R is an relation on A and $(a, b) \in R$ then we sometimes write aRb.

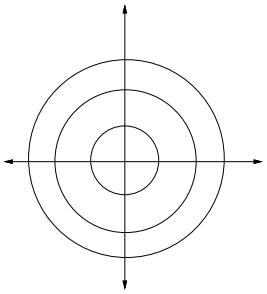


Figure 1.2: Here we consider two points in the plane equivalent if they have the same distance to the origin. Thus every equivalence class is either a circle or the set containing the origin.

Definition 1.3.16 An **equivalence relation** on A is a relation $\sim$ on A satisfying
(1) $a \sim a$ for all $a \in A$,
(2) if $a \sim b$ then $b \sim a$,
(3) $a \sim b$ and $b \sim c$ imply that $a \sim c$.

The equivalence relation is an abstraction of equality. You have probably encountered equivalence relations when you studied plane geometry: congruence and similarity of triangles are the two most common examples. The point of the concept of congruence of two triangles is that even if two triangles are not equal as subsets of the plane, for some purposes they can be considered as being "the same". Later in this chapter we will consider an equivalence relation on the states of a finite state machine. In that situation two states will be considered equivalent if they react in the same way to the same input.

Example 1.3.17 A classic example of an equivalence relation is congruence modulo an integer. (Sometimes this is taught to children in the case of $n = 12$ and called clock arithmetic, since it is like adding hours on a clock, i.e. 4 hours after 11 o'clock is 3 o'clock.) Here, we fix an integer, n, and we consider two other integers to be equivalent if when we divide n into each of them "as much as possible", then they have the same remainders. Thus, if $n = 7$ then 15 and 22 are consider to be equivalent. Also 7 is equivalent to 0, 14, 21,... , i.e. 7 is equivalent to all multiple of 7. 1 is equivalent to all multiples of 7, plus 1. Hence, 1 is equivalent to -6. We will elaborate on this notion later, since it is very relevant to the study of finite state machines.

Example 1.3.18 Suppose one declares two points in the plane to be equivalent if they have the same distance to the origin. Then this is clearly an equivalence relation. If one fixes a point, the set of other points in the plane that are equivalent to that point is the set of points lying on the circle containing the point that is centered about the origin, unless of course the point is the origin, which is equivalent only to itself. (See figure 1.2).

Definition 1.3.19 If $\sim$ is an equivalence relation on A and $a \in A$, then we define the equivalence class of a, $[a]$, as $\{b \in A | b \sim a\}$. Notice that $a \sim b$ iff $[a] = [b]$. If it is not true that $a \sim b$, then $[a] \cap [b] = \emptyset$.

Definition 1.3.20 A **partition** of a set A is a collection of non-empty subsets of A, P, with the properties
(1) For all $x \in A$, there exists $U \in P$ such that $x \in U$. (P "covers" A.)
(2) If $U, V \in P$ then either $U = V$ or $U \cap V = \emptyset$.

The elements of P are sometimes referred to as **blocks**. Very often though the set has some other structure and rather than "block" or "equivalence class" another term is used, such as residue class, coset or fiber. The typical diagram that goes along with the definition of a partition can be seen in figure 1.3 . We draw

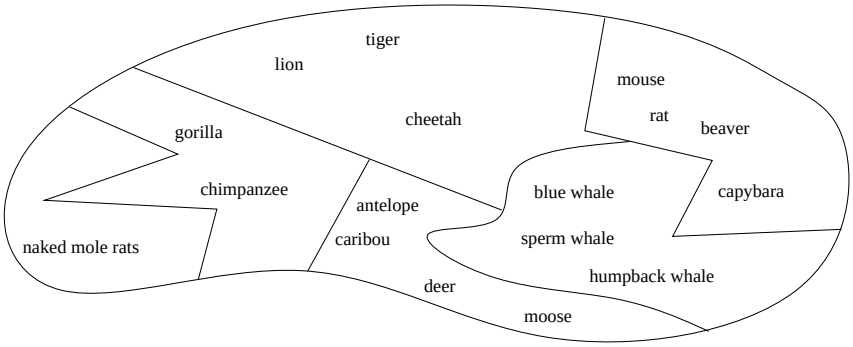


Figure 1.3: A simple classification of animals provides an example of an equivalence relation. Thus, one may view each of the usual biological classifications such as kingdom, phylum, genus, species, etc. as equivalence relations or partitions on the set of living things.

the set A as a blob, and break it up into compartments, each compartment corresponding to an element of P.
There is a canonical correspondence between partitions of a set and equivalence relations on that set. Namely, given an equivalence relation, one can define a partition as the set of equivalence classes. On the other hand, given a partition of a set, one can define two elements of the set to be equivalent if they lie in the same block.

Definition 1.3.21 A **function** is a set of ordered pairs such that any two pairs with the same first member also have the same second member.

The **domain** of a function, f, **dom(f)**, is the set $\{a \mid \exists b \text{ s.t. } (a,b) \in f\}$. The **range** of f, **ran(f)**, is the set $\{b \mid \exists a \text{ s.t. } (a,b) \in f\}$. If the domain of f is A and the range of f is contained in b then we may write

$$f : A \longrightarrow B.$$

Several comments need to be made about this definition. First, a function is a special kind of relation. Therefore we can use the relation notation afb to say that $(a,b) \in f$. This in not standard notation for functions! It is more common in this case to write $b = f(a)$ or $f(a) = b$. This brings us to our second comment.
The reader is probably used to specifying a function with a formula, like $y = x^2$, or $f(x) = e^{cos(x)}$, or with the more modern notation $x \mapsto x^2$. Even though this notation doesn't indicate what the domain and range of a function is, it is usually clear from the context. You may remember that a common exercise in calculus courses is to determine the domain and range of a function given by such a formula, assuming that the domain and range lie in the real numbers. For example, what is the domain and range of $y = \dfrac{1}{x^2 - 1}$?
In this book we usually will need to be more explicit about such things, but that does not mean that the reader's past experience with functions is not useful.
Finally, consider the difference between what is meant by a function in a programming language as opposed to our definition. Our definition doesn't make any reference to a method for computing the range values from the domain values. In fact this may be what people find the most confusing about this sort of abstract definition the first time they encounter it. The point is that the function exists *independently* of any method or formula that might be used to compute it. Notice that is very much in the philosophy of modern programming: functions should be given just with specs on what they will do, and the user need not know anything about the specific implementation. Another way to think of this is that for a given function there may be *many* programs that compute, and one should take care to distinguish between a function and a

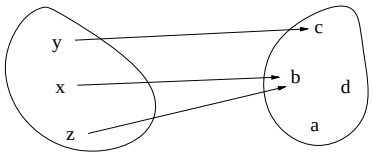


Figure 1.4: A schematic depiction of a function. The blob on the left represents the domain, with elements x, y and z, and the blob on the right contains the range. Thus, for example, $f(x) = b$. This function is not injective or surjective.

Figure 1.5: The function on the left is injective, but not surjective. The function on the right is surjective but not injective.

program that computes this function. In fact, below in example 1.3.40 you will see that there are functions which can't be computed at all!

Example 1.3.22 The set $\{(1,5),(2,4),(3,7),(6,7)\}$ is a function.

Example 1.3.23 The set $\{(1,5),(2,4),(3,7),(3,7)\}$ is not a function. It is a relation however.

Example 1.3.24 Take f to be the set $\{(n,p_n)|n \in Z^+,\ p_n \text{ is the nth prime number}\}$. Many algorithms are know for computing this function, although this is an active area of research. No "formula" is known for this function.

Example 1.3.25 Take f to be the set $\{(x,\sqrt{x})|x \in R,\ x \geq 0\}$. This is the usual square root function.

Example 1.3.26 The empty set may be considered a function. This may look silly, but it actually allows for certain things to work out very nicely, such as theorem 1.3.42.

Definition 1.3.27 A function (A,f,B) is **1-1** or **injective** if any two pairs of f with the same second member have the same first member. f is **onto** B, or **surjective** , if the range of f is B.

Definition 1.3.28 A function $f : A \longrightarrow B$ that is both injective and surjective is called a **bijection** and we say that **there is a bijection between A and B**.

The idea of a bijection between two sets is that the elements of the sets can be matched up in a unique way. This provides a more general way of comparing the sizes of things than counting them. For example, if you are given two groups of people and you want to see if they were the same size, then you could count them. But another way to check if they are the same size is to start pairing them off and see if you run out of people in one of the groups before the other. If you run out of people in both groups at the same time, then the groups are of the same size. This means that you have constructed a bijection between the two groups (sets, really) of people. The advantage of this technique is that it avoids counting. For finite sets this is a useful technique often used in the subject of **combinatorics** (see exercise). But it has the nice feature that it also provides a way to check if two **infinite** sets are the same size! A remarkable result of all this is that one finds that some infinite objects are bigger than others. See example 1.3.40.

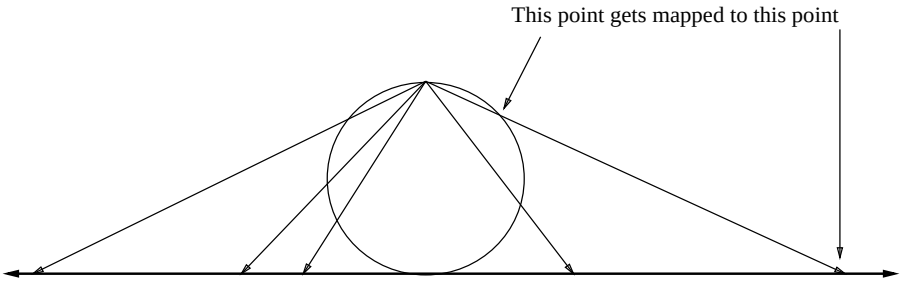


Figure 1.6: A bijection from the circle minus a point to the real line.

Example 1.3.29 It turns out that most of the infinite sets that we meet are either the size of the integers or the size of the real numbers. Consider the circle $x^2 + (y-1)^2 = 1$ with the point $(0, 2)$ removed. We claim that this set has the same size as the real numbers. One can geometrically see a bijection between the two sets in figure 1.6. We leave it as an exercise to the reader to write down the equations for this function and prove that it is a bijection.

Example 1.3.30 Another surprising consequence of our notion of size is that the plane and the real line have the same number of points! One way to see this is to construct a bijection from $(0, 1) \times (0, 1) \longrightarrow (0, 1)$, i.e. from the unit square to the unit interval. To do this we use the decimal representation of the real numbers: each real number in $(0, 1)$ can be written in the form $.d_1d_2d_3...$ where d is an integer from 0 to 9. We then define the bijection as

$$(.d_1d_2d_3..., .c_1c_2c_3...) \longrightarrow .d_1c_1d_2c_2d_3c_3....$$

There is something to worry about here - namely that some numbers have two decimal representations. For example $.1\bar{9} = .2$. We leave it to the reader as an exercise to provide the appropriate adjustments so that the above map is a well-defined bijection.

Definition 1.3.31 A **sequence** is a function whose domain is N. If $f : N \longrightarrow A$ then we say that f **is a sequence of elements of** A.

Intuitively, one thinks of a sequence as an infinite list, because given a sequence f we can list the "elements" of the sequence: $f(0), f(1), f(2), ...$ We will be loose with the definition of sequence. For example, if the domain of f is the positive integers, will will also consider it a sequence, or if the domain is something like $\{1, 2, ...k\}$ we will refer to f as a finite sequence.

Definition 1.3.32 If there is a bijection between A and B then we say that A and B have the same **cardinality**. If there is a bijection between A and the set of natural numbers, then we say that A is **denumerable**. If a set is finite or denumerable, it may also be referred to as **countable**.

Notice that we have not defined what is means for a set to be finite. There are at least two ways to do this. One way is to declare a set to be infinite if there is a bijection from the set to a proper subset of that set. This makes it easy to show, for example that the integers are infinite: just use $x \mapsto 2x$. Then a set is said to be finite if it is not infinite. This works out OK, but seems a bit indirect. Another way to define finite is to say that there is a bijection between it and a subset of the natural numbers of the form $\{x|x \leq n\}$, where n is a fixed natural number. In any case, we will not go into these issues in any more detail.
Note that, in light of our definition of a sequence, a set is countable is it's elements can all be put on an infinite list. This tells us immediately, for example, that the set of programs from a fixed language is countable, since they can all be listed (list then alphabetically).

Definition 1.3.33 If $f : A \longrightarrow B$ is a bijection then we write $A \sim B$. If $f : A \longrightarrow B$ is an injection and there is no bijection between A and B then we write $A < B$.

Definition 1.3.34 The **power set** of a set A is the set

$$\wp(A) = \{B \mid B \subset A\}.$$

Example 1.3.35 If $A = \{1, 2, 3\}$ then $\wp(A) = \{\{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}\{2, 3\}, \{1, 2, 3\}, \emptyset\}$.

Example 1.3.36 If $A = \emptyset$ then $\wp(A) = \{\emptyset\}$.

The above example indicates that there is a natural injection from a set to it's power set, i.e. $x \to \{x\}$. The fact that no such surjection exists is the next theorem.

Theorem 1.3.37 For any set A, $A < \wp(A)$.

Proof It would be a crime to deprive the reader of the opportunity to work on this problem. It is a tough problem, but even if you don't solve it, you will benefit from working on it. See the exercises if you want a hint.

Definition 1.3.38 For any two sets A and B we define

$$A^B = \{f \mid f : B \longrightarrow A\}.$$

Example 1.3.39 $\{0, 1\}^{\{1,2,3\}}$ has eight elements. We can see this by counting. If we are given a function $f : \{1, 2, 3\} \longrightarrow \{1, 2\}$, then how many choices do we have for $f(1)$? Two, since it can be 0 or 1. Likewise for $f(2)$ and $f(3)$. Thus there are $2 \times 2 \times 2 = 8$ different functions.

Example 1.3.40 Here we show that $N < N^N$, illustrating a technique called **Cantor diagnolization**. This method can also be used to show that $N < R$. We do the proof by contradiction. Suppose that $N \sim N^N$. Thus there is a bijection $F : N \longrightarrow N^N$. This means that we can "list" the elements of N^N: $F(0), F(1), F(2)...$ (remember - $F(k)$ is an function, not a number!) We now show that there is a function not on the list. Define $g : N \longrightarrow N$ by the equation

$$g(n) = F(n)(n) + 1.$$

Then g must be on the list, so it is $F(k)$ for some k. But then $F(k)(k) = g(k) = F(k)(k)+1$, a contradiction. Therefore no such bijection can exist, i.e. N^N is not denumerable.

Food for thought It would seem that if one picked a programming language, then the set of programs that one could write in that language would be denumerable. On the other hand, N^N is not denumerable, so it would appear to be the case that there are more functions around then there are programs. This is the first evidence we have that there are things around which just can't be computed.

Definition 1.3.41 If $A \subset B$ then we define the characteristic function, χ_A of A (with respect to B) for each $x \in B$ by letting $\chi_A(x) = 1$ if $x \in B$ and $\chi_A(x) = 0$ otherwise. Notice that χ_A is a function on B, not A, i.e. $\chi_A : B \longrightarrow \{0, 1\}$. .

In our above notation, The set of characteristics functions on a set A is just $\{0, 1\}^A$.

Theorem 1.3.42 For any set A, $\{0, 1\}^A \sim \wp(A)$.

Proof We need to find a function between these two sets and then prove that it is a 1-1 and onto function. So, give a characteristic function, f, what is a natural way to associate to it a subset of A ? Since f is a characteristic function, there exists a set A so that $f = \chi_A$. The idea is then to map f to A. In other words define $F : \{0,1\}^A \longrightarrow \wp(A)$ as $F(f) = F(\chi_A) = A$.
Several things have to be done now. First of all how do we know that this function is well defined ? Maybe for a given characteristic function there are two unequal sets A and B and $\chi_A = \chi_B$. But it is easy to see that this can't happen (why ?). So now we know we have a well defined mapping.
Next, we show that F is onto. Thus, for any $U \in \wp(A)$ we need to produce an $f \in \{0,1\}^A$ such that $F(f) = U$. Here we can just let $U = \{x \in A | f(x) = 1\}$.
To show that F is 1-1 suppose that $F(f) = F(g)$. Say $f = \chi_A$ and $g = \chi_B$. Thus $A = B$. But then certainly $\chi_A = \chi_B$, i.e. $f = g$.

Definition 1.3.43 A **partial order** on A is a relation $\leq$ on A satisfying the following: for every $a, b, c \in A$
(1) $a \leq a$,
(2) $a \leq b$ and $b \leq c$ implies that $a \leq c$.
(3) $a \leq b$ and $b \leq a$ implies that $a = b$.

Example 1.3.44 The usual order on the set of real numbers is an example of a partial order that everyone is familiar with. To prove that (1), (2) and (3) hold requires that one knows how the real numbers are defined. We will just take them as given, so in this case we can't "prove" that the usual order is a total order.

Example 1.3.45 A natural partial order that we have already encountered is the subset partial order, $\subset$ known also as inclusion. The reader should check (1), (2) and (3) do in fact hold.

Definition 1.3.46 Given a set A, $a, b \in$ and a partial order $\leq$ on A, we say that a and b are **compatible** if either $a \leq b$ or $b \leq a$. A **total order** on a set is a partial order on that set in which any two elements of the set are compatible.

Example 1.3.47 In example (1.3.44) the order is total, but in example (1.3.45) is is not. It is possible to draw a picture of this - see figure 1.7. Notice that there is one containment that we have not included in this diagram, the fact that $\{1\} \subset \{1,3\}$, since it was not typographically feasible using the containment sign. It is clear from this example that partially ordered sets are things that somehow can't be arranged in a linear fashion. For this reason total orders are also sometime called **linear orderings**.

Example 1.3.48 The **Sharkovsy ordering** of N is:

$$3 < 5 < 7 < 9 < \cdots < 3 \cdot 2 < 5 \cdot 2 < 7 \cdot 2 < 9 \cdot 2 < \cdots < 3 \cdot 2^2 < 5 \cdot 2^2 < 7 \cdot 2^2 < 9 \cdot 2^2 < \cdots < 2^3 < 2^2 < 2 < 1.$$

Notice that in this ordering of the natural numbers, there is a largest and a smallest element. This ordering has remarkable applications to theory of dynamical systems.

Example 1.3.49 The dictionary ordering on $Z \times Z$, a total order, is defined as follows. $(a, b) < (c, d)$ if $a < c$ or $a = c$ and $b < d$. This type of order can actually be defined on any number of products of any totally ordered sets.

We finish this section with a discussion of the **pigeon hole** principle. The pigeon hole principle is probably the single most important elementary tool needed for the study of finite state automata. The idea is that if there are $n + 1$ pigeons, and n pigeon holes for them to sit in, then if all the pigeon go into the holes it then it must be the case that two pigeons occupy the same hole. It seems obvious, and it is. Nevertheless it turns out to be the crucial fact in proving many non-trivial things about finite state automata.
The formal mathematical statement of the pigeon hole principle is that if $f : A \longrightarrow B$ and A and B are finite sets A has more elements then B, then f is not injective. How does one prove this ? We leave this to the curious reader, who wants to investigate the foundations some more.

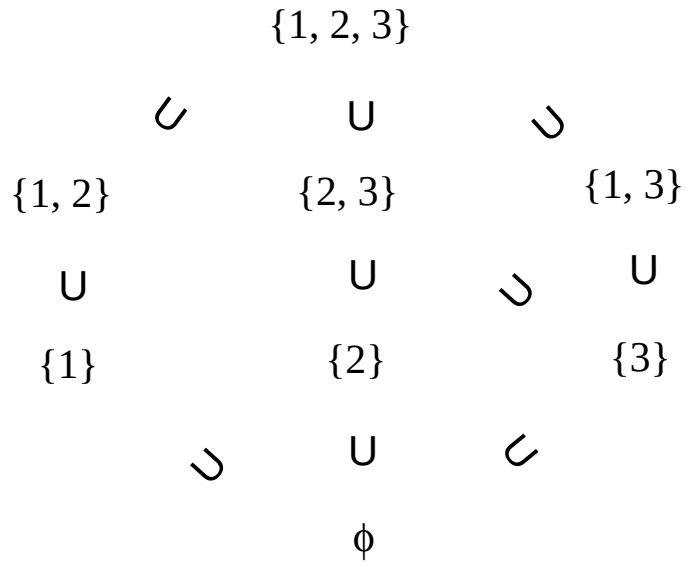


Figure 1.7: Containment is a partial ordering on the power set of a set.

Exercises

1 Show that $A \cap B = B \cap A$.

2 Show that $(A \cup B) \cup C = A \cup (B \cup C)$.

3 Show that $(A \cap B) \cap C = A \cap (B \cap C)$.

4 Show that $(a, b) = (c, d)$ implies that $a = c$ and $b = d$. (Hint - consider separately the cases of $a = b$ and $a \neq b$.)

5 Show that for any sets A, B and C that
(i) $A \sim A$,
(ii) $A \sim B$ implies $B \sim A$, and
(iii) $A \sim B$ and $B \sim C$ implies that $A \sim C$.

(Given that (i), (ii) and (iii) are true it is tempting to say that $\sim$ is an equivalence relation on the set of sets. But exercise 21 shows why this can cause technical difficulties.)

6 Show the following:
(i) $Z \sim N$
(ii) $nZ \sim Z$ for any integer n.

7 Show that $N \times N \sim N$.

8 Show that $Q \sim N$.

9 Show that $A \sim C$ and $B \sim D$ implies that $A \times B \sim C \times D$.

10 Show that $A \times (B \cup C) = (A \times B) \cup (A \times C)$ and that $A \times (B \cap C) = (A \times B) \cap (A \times C)$

11 Show that $A^{(B^C)} \sim A^{B \times C}$. Do you know any computer languages where this bijection occurs ?

12 How many elements are in $\emptyset^\emptyset$?

13 Show that $\bigcup\{N^n | n \in Z^+\} \sim N$. (This is the essence of what is called *Godel numbering.*)

14 Show that a countable union of countable sets is countable.

15 Show that if A has n elements and B has m elements, $n, m \in Z^+$, then A^B has n^m elements. What does this have to do with exercise 12 ?

16 Use exercise 15 to show that if A has $n \in N$ elements, then $\mathcal{P}(\mathcal{A})$ has 2^n elements.

17 Recall that we define $0! = 1$ and for $n > 0$ we define $n! = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1$. We define $Perm(A) = \{f \in A^A | f \text{ is a bijection}\}$. Show that if A has n elements then $Perm(A)$ has $n!$ elements.

18 Define the relation $div = \{(a, b) | a, b \in N, a \text{ is a divisor of } b\}$. Show that div is a partial order on N.

19 We define intervals of real numbers in the usual way: $[a, b] = \{x \in R | a \leq x \leq b\}, (a, b) = \{x \in R | a < x < b\}$, etc. Show that if $a < b$ and $c < d$ that $(a, b) \sim (c, d)$ and that $[a, b] \sim [c, d]$.

20 Show that $[0, 1) \sim [0, 1]$.

21 Russel's Paradox Consider the set $U = \{A | A \notin A\}$. Show that $U \in U$ iff $U \notin U$.

Does this paradox mean that set theory is flawed ? The answer is "yes", in the sense that if you are not careful you will find yourself in trouble. For example, you may be able to generate contradictions if you don't follow the rules of your set theory. Notice that we didn't state what the rules were and we ran into a paradox! This is what happened after Cantor introduced set theory: he didn't have any restrictions on what could be done with sets and it led to the above famous paradox, discover by Bertrand Russell. As a result of this paradox, people started to look very closely at the foundations of mathematics, and thereafter logic and set theory grew at a much more rapid pace than it had previously. Mathematicians came up with ways to avoid the paradoxes in a way that preserved the essence of Cantor's original set theory. In the mean time, most of the mathematicians who were not working on the foundations, but on subjects like Topology and Differential Equations ignored the paradoxes, and nothing bad happened. We will be able to do the same thing, i.e. we may ignore the technical difficulties that can arise in set theory. The reason for this can be well expressed by the story of the patient who complains to his doctor of some pain:
Patient : "My hand hurts when I play the piano. What should I do ?"
Doctor: "Don't play the piano."
In other words, it will be *OK* for us to just use set theory if we just avoid doing certain things, like forming $\{A | A \notin A\}$! or forming sets like "the set that contains all sets". This will definitely cause some trouble. Luckily, nothing we are interested in doing requires anything peculiar enough to cause this kind of trouble.

22 Show that for any set A, $A < \wp(A)$. (Hint-consider the subset of A $\{a \in A | a \notin f(a)\}$. Keep Russel's paradox in mind.)

23 Let A be a set of open intervals in the real line with the property that no two of them intersect. Show that A is countable. Generalize to higher dimensions.

24 One way to define when a set is infinite is if there is a bijection between the set and a proper subset of the set. Using this definition show that N, Q and R are infinite.

25 Using the definition of infinite from exercise 24, show that if a set contains an infinite subset, then it is infinite.

26 Again, using the definition of infinite from exercise 24, show that if $a < b$ then (a, b) and $[a, b]$ are infinite.

27 Show that that the Sharkovsky ordering of N is a total ordering.

28 Find an explicit way to well-order Q. Can you generalize this result ?

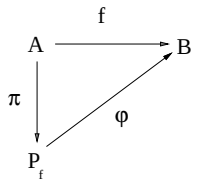
29 A *partition* of a positive integer is a decomposition of that integer into a sum of positive integers. For example, $5+5$, and $2+6+1+1$ are both partitions of 10. Here order doesn't count, so we consider $1+2$ and $2+1$ as the same partition of 3. The numbers that appear in a partitions are called the *parts* of the partition. Thus the partition of 12, $3+2+2+5$, has four parts: $3, 2, 2$ and 5. Show that the number of ways to partition n into partitions with m parts is equal to the number of partitions of n where the largest part in any of the partitions is m. (Hint - Find a geometric way to represent a partition.) Can you re-phrase this exercise for sets ?

30 Cantor-Bernstein Theorem Show that if there is an injection from A into B, and an injection from B into A, that there is a bijection between A and B.

31 If P is a partition of A then there is a natural map $\pi : A \longrightarrow P$ defined by $a \mapsto [a]$. Show that π is surjective.

32 The First Isomorphism Theorem for Sets Given a function between two sets $f : A \longrightarrow B$, there is a natural partition P_f with corresponding equivalence relation $\sim$ induced on A by defining $[a] = f^{-1}(f(a))$. Prove that
(i) P_f is a partition of A,
(ii) For all $a, b \in A, a \sim b$ iff $f(a) = f(b)$,
(ii) there is a bijection $\phi : P_f \longrightarrow ran(f)$ with the property that $f = \phi\pi$.

This last property is sometimes phrased as saying that the following diagram "commutes":



This theorem occur in several different forms, most notably in group theory.

33 Generalize example 1.3.29 to 2-dimensions by considering a sphere with its north pole deleted sitting on a plane. Write down the equations for the map and prove that it is a bijection. This famous map is called the *stereographic projection*. It was once used by map makers, but is very useful in mathematics (particularly complex analysis) due to it's numerous special properties. For example, it maps circles on the sphere to circles or lines in the plane.

1.4 The Natural numbers and Induction

Now that the reader has had a brief (and very intense) introduction to set theory, we now look closely and the set of natural numbers and the method of proof called induction. What is funny about induction is that from studying the natural numbers one discovers a new way to do proofs. You may then wonder, what really comes first - the set theory or the logic. Again, we won't address these issues, since they would bring us too far astray.
We introduce the reader to induction with two examples, and then give a formal statement. The first example seems simple, and on first meeting induction through it, the reader may think "how can such a simple idea be useful ?". We hope that the second example, which is a more subtle application of induction,

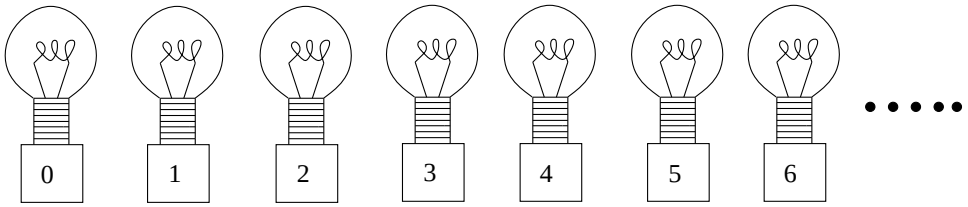


Figure 1.8: Induction via light bulbs

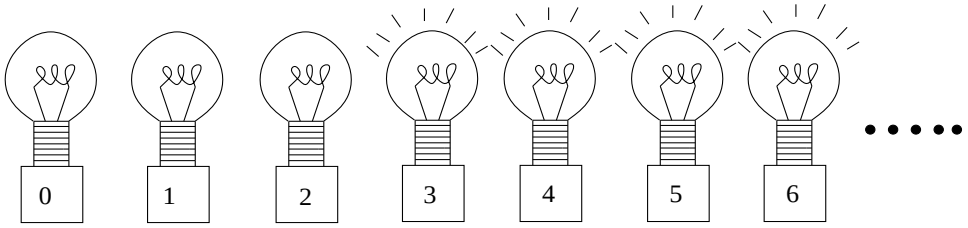


Figure 1.9: If bulb 3 is on, then so it 4, 5, etc.

answers this question. Induction is not merely useful - it is one of the most powerful techniques available to mathematicians and computers scientists, not to mention one of the most commonly used. Not only can induction be used to prove things, but it usually gives a constructive method for doing so. As a result, induction is closely related to *recursion*, a crucial concept (to say the least!) in computer science. Our first example will illustrate the idea of induction, and the second and third are applications.

Suppose that there is an infinite line of light bulbs, which we can think of as going from left to right, and labeled with 0,1,2,.... The light bulbs are all in the same circuit, and are wired to obey the following rule: If a given light bulb is lit, then the light bulb to the right of it will also be lit.

Given this, what can be concluded if one is told that a given light bulb is lit ? Clearly, *all* of the bulbs to the right of that bulb will also be lit.

In particular, what will happen if the first light bulb in the line is turned on ? It seems to be an obvious conclusion that they must all be on. If the you understands this, then you understand one form induction. The next step is to learn how to apply it.

Our second example is a game sometimes called "the towers of Hanoi". The game consists of three spindles, A, B and C, and a stack of n disks, which uniformly diminish in size from the first to the last disk. The disks have holes in them and in the start of the game are stacked on spindle A, with the largest on the bottom to the smallest on the top, as in figure. You are allowed to move a disk and put it on a second spindle provided that there is no smaller disk already there. The goal of the game is to move all of the disks to spindle B.

If one plays this game for a little while, it becomes clear that it is pretty tricky. But if it is approached systematically, there is an elegant solution that is nice example of induction.

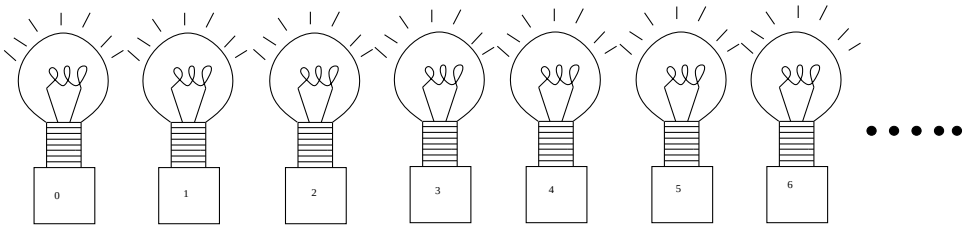


Figure 1.10: If bulb 0 is lit, then they all are!

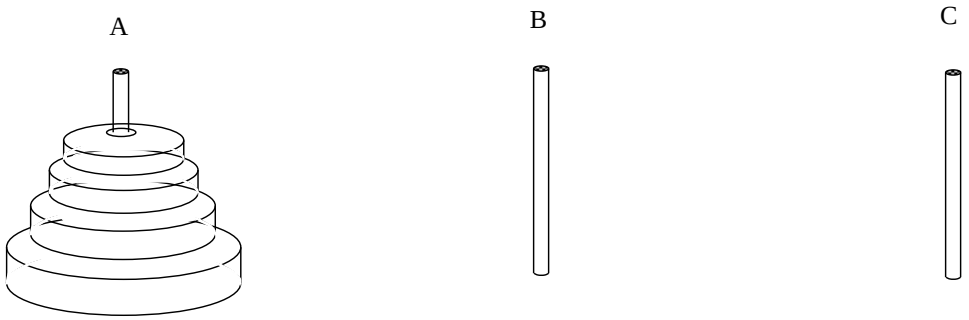


Figure 1.11: The Towers of Hanoi

First, try solving the game with just two disks. This is easily accomplished through the following moves: disk 1 first goes to C, then disk 2 goes to B, then disk 1 goes to B. Notice that there is nothing special about spindle B here - we could have moved all of the disks to spindle C in the same manner.
Now consider the game with three disks. We come to this problem with the solution of the first, and we use it as follows. We know that we are free to move the top two disks around however we like, keeping the smaller one above larger one. So use this to move them to spindle C. Then move the remaining third disk to spindle B. Finally, apply our method of moving the two disks again to the disks on C, moving them to B. This wins the game for us.
The game with three disks then allows us to solve the game with four disks. (Do you see how ?) At this point, if you see the general pattern you probably will say "ahh! you can then always solve the game for any number of disks!". This means that the induction is so clear to you, that you don't even realize that it is there! Let's look closely at the logic of the game.
Suppose that P_n is the statement "the game with n disks can be solved". We saw that P_2 was a true statement. And after seeing that $P_n \Rightarrow P_{n+1}$ we immediately concluded that P_n was always true. Well, this certainly is intuitive. And you may even see how this is really the same as the light bulb example. So we now ask, what is it that allows us to conclude that P_n is true for all n ? Or, what is it that allows us to conclude that all of the lightbulbs are on if the first one is on ? Well, if one wants a formal proof, the right way to start is to say "It's so obvious, how could it be otherwise ?". This is key to illuminating what is going on here. Assume that for some positive integer, m, P_m was false. Then it must be that P_{m-1} is also false, or else we would have a contradiction. You may be tempted to try the same reasoning on P_{m-1}, concluding that P_{m-2} is false. Where does it end ? If there was a smallest value k for which P_k was false, then we would know that P_{k-1} was true. But we know that if P_{k-1} is true that then P_k must be true, a contradiction. To sum up, if one of the P_n's was false, then we can make a contradiction by looking at the smallest one that is false, so they must all be true.
This is all perfectly correct, and the fact that we are using that is so crucial is the following:

Every non-empty subset of N has a smallest member.

The above fact is known as the *well-ordering property of N*, and we will take this as an axiom. [1] To sum up, the principle of induction is as follows: if one know that P_1 is true and that $P_n \Rightarrow P_{n+1}$ for every n, then P_n is true for every n.
Incidentally, you might find it useful to ask yourself if there are any sets besides N for which the above is true. For example, it is certainly not true for the integers. It turns out that any set can be ordered in such a way that the above property holds, if you have the right axioms of set theory. An ordering with this property is called a *well-ordering*. If you try to imagine a well-ordering of R, you will see that this is a very strange business!

[1] There are many different ways to approach this - one can even start with a set of axioms for the real numbers and define N and then prove the well-ordering property.

Example 1.4.1 Show that $1 + 2 + \cdots + n = \dfrac{n(n-1)}{2}$.

Proof Let the above statement be P_n. Clearly it is true if $n = 1$. If we know P_n is true then adding $n + 1$ to both sides of the equality gives

$$1 + 2 + \cdots + n + n + 1 = \frac{n(n-1)}{2} + n + 1,$$

but some algebra shows that this is just P_{n+1}. Therefore, $P_n \Rightarrow P_{n+1}$, so by induction the formula is always true for all $n = 1, 2, ...$

This is a classic example, and probably the single most used example in teaching induction. There is one troubling thing about this though, namely where does the formula come from ? This is actually the hard part! In this case there is a geometric way to "guess" the above formula (see exercise 4). On one hand, mathematical induction provides a way of proving statements that you can guess, so it doesn't provide anything new except in the verification of a statement. But on the other hand it is closely related to recursion, and can be used in a constructive fashion to solve a problem.
If we stopped at this point, you might have gotten the impression from the above example that induction is a trick for proving that certain algebraic formulas or identities are true. Induction is a very useful tool for proving such things, but it can handle much more general things, such as the following example.

Example 1.4.2 Recall that an integer > 1 is said to be *prime* if its only divisors are 1 and itself. Show that every integer > 1 is a product of prime numbers.

Proof The proof is by induction. Let P_n be the statement "every integer from 2 and n is a product of primes". The base case, P_2, "2 is a product of primes", is certainly true. We want to show that if we assume that P_n is true that we can prove P_{n+1}, so we assume that every integer from 2 and n is a product of primes. P_{n+1} is the statement "every integer from 2 to $n + 1$ is a product of primes". This means that we need to show that $n + 1$ is a product of primes. If $n + 1$ is a prime number then certainly this is true. Otherwise, $n + 1 = ab$ where a and b are integers with $2 \leq a, b \leq n$. But now we can use P_n to conclude that a and b are products of primes. But then clearly $n + 1$ is a product of primes since $n + 1 = ab$.

Exercises

1 Prove that

$$\frac{1}{1 \cdot 2} + \frac{1}{2 \cdot 3} + \frac{1}{3 \cdot 4} \cdots + \frac{1}{n(n+1)} = 1 - \frac{1}{n+1}.$$

Can you give a proof without induction ?

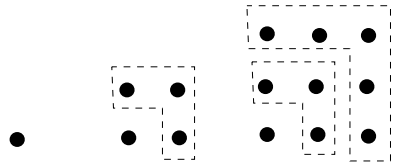
2 Write a computer program that wins the towers of Hanoi game. Do you see the relationship between induction and recursion ?

3 How many moves does it take to win the towers of Hanoi game if it has n disks ?

4 Find a formula for the sum $1 + 3 + 5 + \cdots + 2n - 1$, and prove it by induction. There are at least two ways to do this, one using a geometric representation of the sum as is indicated below.

5 Find a geometric representation, like that given in exercise 4, for the formula in example 1.4.1.

6 Use induction to prove the formula for the sum of a *geometric series*: $1 + x + x^2 + x^3 + \cdots + x^n = \dfrac{1 - x^{n+1}}{1 - x}$.
It is also possible to prove this directly, using algebra. For what x is this formula true (e.g. does it work when x is a real number, a complex number, a matrix, an integer mod k, etc.) ?



7 Use induction to show that $1^2 + 2^2 + \cdots + n^2 = \dfrac{n(n+1)(2n+1)}{6}$. What about for sums of higher powers ?

8 Notice that

$$1^3 = (1)^2,$$
$$1^3 + 2^3 = (1+2)^2,$$
$$1^3 + 2^3 + 3^3 = (1+2+3)^2.$$

Can you formulate a general statement and prove it with induction ?

9 Recall that we define $0! = 1$ and for $n > 0$ we define $n! = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1$. Notice that

$$1! = 1!,$$
$$1! \cdot 3! = 3!,$$
$$1! \cdot 3! \cdot 5! = 6!,$$
$$1! \cdot 3! \cdot 5! \cdot 7! = 10!.$$

Can you formulate a general statement and prove it with induction ?

10 We define $C_{n,k} = \dfrac{n!}{k!(n-k)!}$. Use induction to show that $C_{n,0} + C_{n,1} + \cdots + C_{n,n} = 2^n$.

11 If one had n objects, then the number of way of picking k objects from the n objects is in fact $C_{n,k}$. Use this fact to give another solution to exercise 10.

12 In section 1.3, exercises 15, 16 and 17, did you use induction in your proofs ? If not, redo them using induction.

13 For a fixed $n = 1, 2, ...$ we define an *n-vector* $\mathbf{x} = (x_1, ..., x_n)$ as an ordered n-tuple of real numbers. The *norm* of $||\mathbf{x}||$ is the number $\sqrt{x_1^2 + \cdots + x_n^2}$. We may add and subtract vectors componentwise. The *triangle inequality* states that for any three n-vectors $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{z}$

$$||\mathbf{x} - \mathbf{z}|| \leq ||\mathbf{x} - \mathbf{y}|| + ||\mathbf{y} - \mathbf{z}||.$$

Using this and induction prove the generalized triangle inequality: If $\mathbf{x_1}, ..., \mathbf{x_m}$ are n-vectors, then

$$||\mathbf{x_1} - \mathbf{x_m}|| \leq ||\mathbf{x_1} - \mathbf{x_2}|| + ||\mathbf{x_2} - \mathbf{x_3}|| + \cdots + ||\mathbf{x_{m-1}} - \mathbf{x_m}||.$$

14 (*This exercise requires some knowledge of linear algebra)* Show that the determinant of an upper triangular matrix is the product of its diagnol entries.

1.5 Foundations of Language Theory

We now begin to lay the mathematical foundations of languages that we will use throughout the rest of this book. Our viewpoint a language is a set of strings. In turn, a string is a finite sequence of letters from some alphabet. These concepts are defined rigorously as follows.

Definition 1.5.1 An *alphabet* is any finite set. We will usually use the symbol Σ to represent an alphabet and write $\Sigma = \{a_1, \ldots, a_k\}$. The a_i are called the *symbols* of the alphabet.

Definition 1.5.2 A *string* (over Σ) is a function $u : \{1, ..., n\} \longrightarrow \Sigma$ or the function $\epsilon : \emptyset \longrightarrow \Sigma$. The latter is called the *empty string* or *null string* and is sometimes denoted by λ, Λ, e or 1. If a string is non-empty then we may write it by listing the elements of it's range in order.

Example 1.5.3 $\Sigma = \{a, b\}, u : \{1, 2, 3\} \longrightarrow \Sigma$ by $u(1) = a, u(2) = b$ and $u(3) = a$. We write this string as aba.

The set of all strings over Σ is denoted as Σ^*. Thus $\{a\}^* = \{a^n | n = 0, 1, ...\}$, where we have introduced the convention that $a^0 = \epsilon$. Observe that Σ^* is a countable set.

It is a useful convention to use letters from the beginning of the alphabet to represent single letters and letters from the end of the alphabet to represent strings.

Warning Although letters like a and b are used to represent specific elements of an alphabet, they may also be used to represent *variable* elements of an alphabet, i.e. one may encounter a statement like 'Suppose that $\Sigma = \{0, 1\}$ and let $a \in \Sigma$'.

A *language* (over Σ) is a subset of Σ^*. *Concatenation* is a binary operation $\cdot$ on the strings over a given alphabet Σ. If $u : \{1, ..., m\} \longrightarrow \Sigma$ and $v : \{1, ..., n\} \longrightarrow \Sigma$ then we define $u \cdot v : \{1, ..., m + n\} \longrightarrow \Sigma$ as $u(1)...u(m)v(1)...v(n)$ or

$$(u \cdot v)(i) = \begin{cases} u(i) & \text{for } 1 \leq i \leq m \\ v(i - m) & \text{for } m + 1 \leq i \leq m + n. \end{cases}$$

If u is the empty string then we define $u \cdot v = v$ and similarly if v is the empty string. Generally the dot $\cdot$ will not be written between letters.

Remarks Concatenation is not commutative, e.g. $(ab)(bb) \neq (bb)(ab)$. But it is true that for any string u, $u^n u^m = u^m u^n$. Concatenation is *associative*, i.e. $u(vw) = (uv)w$.

u is a *prefix of* v if there exists y such that $v = uy$. u is a *suffix of* v if there exists x such that $v = xu$. u is a *substring of* v if there exists x and y such that $v = xuy$. We say that u *is a proper prefix (suffix, substring) of* v iff u is a prefix (suffix, substring) of v and $u \neq v$.
Each of the above relations are partial orders on Σ^*. Given that Σ is a totally ordered set, e.g. $\Sigma = \{a_1, ..., a_n\}$, then there is a natural extension to a total order on Σ^*, called the *lexicographic ordering*. We define $u \leq v$ if u is a prefix of v or there exists $x, y, z \in \Sigma^*$ and $a_i, a_j \in \Sigma$ such that in the order of Σ we have that $a_i < a_j$ and $u = xa_i y$ and $v = xa_j z$.

Exercises

1 Given a string w, its reversal w^R is defined inductively as follows: $\epsilon^R = \epsilon$, $(ua)^R = au^R$, where $a \in \Sigma$. Also, recall that $u^0 = \epsilon$, and $u^{n+1} = u^n u$. Prove that $(w^n)^R = (w^R)^n$.

2 Suppose that a and b are two different members of an alphabet. Prove that $ab \neq ba$.

3 Suppose that u and v are non-empty strings over an alphabet. Prove that if $uv = vu$ then there is a string w and natural numbers m, n such that $u = w^m, v = w^n$.

4 Prove that for any alphabet Σ, Σ^* is a countable set.

5 Lurking behind the notions of alphabet and language is the idea of a *semi-group*, i.e. a set equipped with an associative law of composition that has an identity element. Σ^* is the *free* semi-group over Σ. Is a given language over Σ necessarily a semi-group ?

1.6 Operations on Languages

A way of building more complex languages from simpler ones is to combine them using various operations. The union and intersection operations we have already seen.

Given some alphabet Σ, for any two languages S, T over Σ, the *difference* $S - T$ of S and T is the language

$$S - T = \{w \in \Sigma^* \mid w \in S \quad \text{and} \quad w \notin T\}.$$

The difference is also called the *relative complement*. A special case of the difference is obtained when $S = \Sigma^*$, in which case we define the *complement* $\overline{L}$ of a language L as

$$\overline{L} = \{w \in \Sigma^* \mid w \notin L\}.$$

The above operations do not make use the structure of strings. T he following operations make use of concatenation.

Definition 1.6.1 Given an alphabet Σ, for any two languages S, T over Σ, the *concatenation* ST *of* S *and* T is the language

$$ST = \{w \in \Sigma^* \mid \exists u \in S,\ \exists v \in T,\ w = uv\}.$$

For any language L, we define L^n as follows:

$$L^0 = \{\epsilon\},$$

$$L^{n+1} = L^n L.$$

Example 1.6.2 For example, if $S = \{a, b, ab\}$, $T = \{ba, b, ab\}$ and $U = \{a, a^2, a^3\}$ then

$$S^2 = \{aa, ab, aab, ba, bb, bab, aba, abb, abab\},$$

$$T^2 = \{baba, bab, baab, bba, bb, abba, abb, abab\},$$

$$U^2 = \{a^2, a^3, a^4, a^5, a^6\},$$

$$ST = \{aba, ab, aab, bba, bab, bb, abba, abb\}.$$

Notice that even though S, T and U have the same number of elements, their squares all have different numbers of elements. See the exercises for more on this funny phenomenon.

Multiplication of languages has lots of nice properties, such as $L\emptyset = \emptyset$, and $L\{\epsilon\} = L$.

In general, $ST \neq TS$.

So far, all of the operations that we have introduced preserve the finiteness of languages. This is not the case for the next two operations.

Definition 1.6.3 Given an alphabet Σ, for any language L over Σ, the *Kleene $*$-closure L^* of L* is the infinite union

$$L^* = L^0 \cup L^1 \cup L^2 \cup \ldots \cup L^n \cup \ldots.$$

The *Kleene $+$-closure L^+ of L* is the infinite union

$$L^+ = L^1 \cup L^2 \cup \ldots \cup L^n \cup \ldots.$$

Since $L^1 = L$, both L^* and L^+ contain L. Also, notice that since $L^0 = \{\epsilon\}$, the language L^* always contains ϵ, and we have

$$L^* = L^+ \cup \{\epsilon\}.$$

However, if $\epsilon \notin L$, then $\epsilon \notin L^+$.

Remark Σ^* has already been defined when Σ is an alphabet. Modulo some set theory, the Kleene *-closure of Σ coincides with this previous definition if we view Σ as a language over itself. Therefore the Kleene *-closure is an extension of our original * operation.

Exercises

1 Prove the following identities:

(i) $L\emptyset = \emptyset$,

(ii) $\emptyset L = \emptyset$,

(iii) $L\{\epsilon\} = L$,

(iv) $\{\epsilon\}L = L$,

(v) $(S \cup \{\epsilon\})T = ST \cup T$,

(vi) $S(T \cup \{\epsilon\}) = ST \cup S$,

(vii) $L^n L = LL^n$.

(viii) $\emptyset^* = \{\epsilon\}$,

(ix) $L^+ = L^* L$,

(x) $L^{**} = L^*$,

(xi) $L^* L^* = L^*$.

2 Given a language L over Σ, we define the reverse of L as $L^R = \{w^R \mid w \in L\}$. For each of the following, either prove equality or provide a counter example. Which of the false equalities can be made true by replacing $=$ with a containment sign ?

(i)$(S \cup T)^R = S^R \cup T^R$;

(ii)$(ST)^R = T^R S^R$;

(iii)$(L^*)^R = (L^R)^*$.

(iv)$(S \cup T)^* = S^* \cup T^*$.

(v)$(ST)^* = T^* S^*$

3 Prove that if $LL = L$ then L contains the empty string or $L = \emptyset$.

4 Suppose that L_1 and T are languages over a two letter alphabet. If $ST = TS$ is $S = T$?

5 Does $A^* = B^*$ imply that $A = B$? Find a counter example or provide a proof.

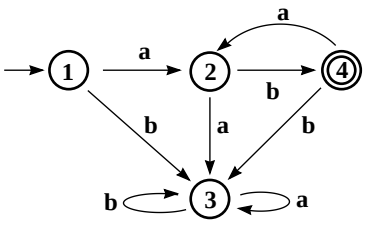


Figure 1.12: A dfa

6 Let $L_k = \{a, a^2, ..., a^k\}$. How many elements are in L_k^2 ?

7 If L is a finite language with k elements, show that L^2 has at most k^2 elements. For each positive integer k find a language with k elements whose square has k^2 elements. Can this be done with a language over a one letter alphabet ?

8 If L is a finite language with k elements, show that L^2 has at least k elements. How close can you come to this lower bound with an example ?

1.7 Deterministic Finite Automata

We are now ready to define the basic type of machine, the *Deterministic Finite Automaton*, or *DFA*. These objects will take a string and either 'accept' or 'reject' it, and thus define a language. Our task is to rigorously define these objects and then explain what it means for one of them to accept a language.

Example 1.7.1 Let $\Sigma = \{a, b\}$, $S = \{ab, abab, ababab, ...\}$. A machine to accept S is

A few comments are necessary. First of all, the small arrow on the left of the diagram is pointing to the *start state*, 1, of the machine. This is where we 'input' strings. The circle on the far right with the smaller circle and the 4 in it is a *final state*, which is where we need to finish if a string is to be accepted. Our convention is that we always point a little arrow to the start state and put little circles in the final states (there may be more than one).

How does this machine process strings ? Take *abb* for example. We start at state 1 and examine the *leftmost* letter of the string first. This is an a so we move from state 1 to 2. Then we consider the second leftmost letter, b, which according to the machine, moves us from state 2 to state 4. Finally, we read a b, which moves us from state 4 to state 3. State 3 is not a final state, so the string is rejected. If our string had been *ab*, we would have finished in state 4, and so the string would be accepted.

What roles is played by state 3 ? If a string has been partially read into the machine and a sequence of *ab*'s has been encountered then we don't know yet if we want to keep the string, until we get to the end or we get an *aa* or *bb*. So we bounce back and forth between states 2 and 4. But if, for example, we encounter the letters *bb* in our string, then we know that we don't want to accept it. Then we go to a state that is not final, 3, and stay there. State 3 is an example of a *dead state*, i.e. a non-final state where all of the outgoing arrows point back at the state.

Example 1.7.2 Let $T = S \cup \{\epsilon\}$. A machine that accepts T is

The point here is that if we allow the empty string we can simplify the machine. The interpretation of processing the empty string is simply that we start at state 1 and move to state 1. Thus, if the start state is also a final state, then empty string is accepted by the machine.
The formal definition of a DFA should now more accessible to the reader.

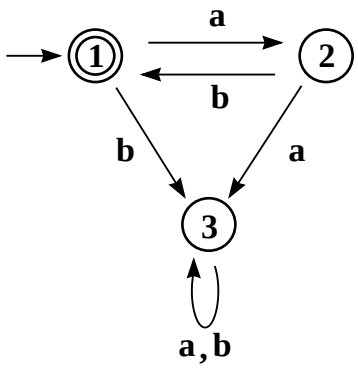


Figure 1.13:

Definition 1.7.3 A deterministic finite automata is a 5-tuple

$$(Q, \Sigma, \delta, q_0, F)$$

where Q and Σ are finite sets, called the *states* and the *alphabet*, $\delta : Q \times \Sigma \longrightarrow Q$ is the *transition function*, $q_0 \in Q$ is a distinguished state called the *start state* and F is a subset of the set of states, known as the set of *final states*.

Notice that our definition doesn't say anything about how to compute with a DFA. To do that we have to make more definitions. The function δ obviously corresponds to the labeled arrows in the examples we have seen: given that we are in a state p, if we receive a letter a then we move to $\delta(p, a)$. But this doesn't tell us what to do with an element of Σ^*. We need to extend δ to a function δ^* where

$$\delta^* : Q \times \Sigma^* \longrightarrow Q.$$

This is done inductively by letting $\delta^*(q, \epsilon) = q$ and $\delta^*(q, ua) = \delta(\delta^*(q, u), a)$ where $a \in \Sigma$ and $u \in \Sigma^*$. We then have

Definition 1.7.4 If $M = (Q, \Sigma, \delta, q_0, F)$ is a DFA then we define the language accepted by M to be

$$L(M) = \{w \in \Sigma^* \mid \delta^*(q_0, w) \in F\}.$$

A language L is said to be *regular* if there is a DFA M such that $L = L(M)$. Sometimes we say that M *recognizes* L.

The fact that not every language over a given alphabet is regular can be proved by a cardinality argument. The number of possible dfas that can be constructed over a given alphabet is a countable set, but the total number of languages over any alphabet is uncountable. So in fact most languages are not regular!. The class of regular languages is the smallest class of languages in a hierarchy of classes that we will consider. To explicitly give an example of a language that is not regular though, we will need something called the pumping lemma. But first we will give more examples of DFAs and their languages.

Example 1.7.5 If $L = \{w \in \{a, b\}^* \mid w$ contains an odd number of $a's\}$ then a DFA specifying L is

Example 1.7.6 If $L = \{w \in \{a, b\}^* \mid w$ ends in the string $abb\}$ then a DFA specifying L is

A useful concept is the *length* of a string w, denoted $|w|$, which is defined to be the total number of letters in a string if the string is non-empty, and 0 is the string is empty.

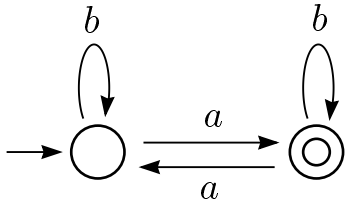


Figure 1.14:

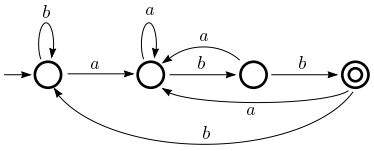


Figure 1.15:

Example 1.7.7 If $L = \{w \in \{a,b\}^* \mid |w| = 3\ \}$ then a DFA specifying L is

Example 1.7.8 If $L = \{w \in \{a,b\}^* \mid w = a^n b^m, n, m > 0\ \}$ then a DFA specifying L is

Example 1.7.9 If $L = \{w \in \{a\}^* \mid |w| = a^{4k+1}, k \geq 0\ \}$ then a DFA specifying L is

Exercises

1 Write a computer program taking as input a DFA $D = (Q, \Sigma, \delta, q_0, F)$ and a string w, and returning the sequence of states traversed along the path specified by w (from the start state). The program should also indicate whether or not the string is accepted.

2 Show that if L is regular then L^R is regular.

3 Construct DFA's for the following languages:
(a) $\{w \mid w \in \{a,b\}^*,\ w$ has neither aa nor bb as a substring$\}$.
(b) $\{w \mid w \in \{a,b\}^*,\ w$ has an odd number of b's and an even number of a's$\}$.

4 Let L be a regular language over some alphabet Σ.

(a) Is the language L_1 consisting of all strings in L of length ≤ 200 a regular language?

(b) Is the language L_2 consisting of all strings in L of length > 200 a regular language?

Justify your answer in both cases.

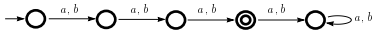


Figure 1.16:

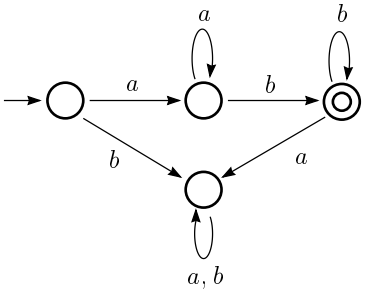


Figure 1.17:

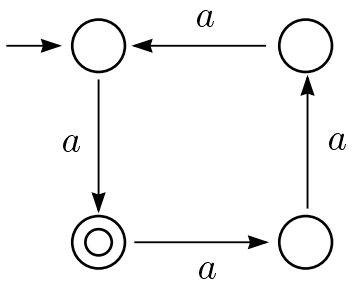


Figure 1.18:

5 Classify all regular languages on a one letter alphabet.

6 Suppose that L is a language over and one letter alphabet and $L = L^*$. Show that L is regular.

7 How many distinct DFAs are there on a given set of n states over an alphabet with k letters ?

8 Show that every finite language is regular.

9 Suppose that a language L is finite. What is the minimum number of states that a machine accepting L need have ?

10 Let Σ be an alphabet, and let L_1, L_2, L be languages over Σ. Prove or disprove the following statements (if false, then provide a counter example).

(i) If $L_1 \cup L_2$ is a regular language, then either L_1 or L_2 is regular.

(ii) If $L_1 L_2$ is a regular language, then either L_1 or L_2 is regular.

(iii) If L^* is a regular language, then L is regular.

11 Define a language to be *one-state* if there is a DFA accepting it that has only one final state. Show that every regular language is a finite union of one-state languages. Give an example of a language that is not one-state and prove that it is not.

12 A DFA is said to be *connected* if given $q \in Q$ there is a string $w \in \Sigma^*$ such that $\delta^*(q_0, w) = q$. Show that if a language is regular, then there is a connected DFA accepting that language.

13 What effect does the changing of the start state of a given machine have on the language accepted by that machine ?

14 What effect does the changing of the final states of a given machine have on the language accepted by that machine ?

15 In our definition of a DFA we had that $\delta : \Sigma \times Q \longrightarrow Q$. We could have 'curried' our map, and instead defined $\delta : \Sigma \longrightarrow Q^Q$, i.e. every letter a gives rise to a map $f_a : Q \longrightarrow Q$ where $f_a(q) = \delta(a, q)$ (see the exercises of 1.2). We may then define $\delta^* : \Sigma^* \longrightarrow Q^Q$. δ^* is an example of a *monoid action*.
For a given machine it may be the case that $\delta^*(\Sigma^*) \subset Perm(Q)$, where $Perm(Q)$ is the set of bijections from Q to itself. Show that if this is the case and the machine is connected that for each letter a of Σ and $n \in N$ there is a string accepted by the machine which contains a power of a greater than n.

16 For any language L over Σ, define the *prefix closure* of L as

$$Pre(L) = \{u \in \Sigma^* \mid \exists v \in \Sigma^* such\ that\ uv \in L\}.$$

Is it true that L being regular implies $Pre(L)$ is regular ? What about the converse ?

17 Show that $\{a^n b^m | n, m \in N\ are\ relatively\ prime\}$ is not regular.

1.8 The Cross Product Construction

Now that we have defined what it means for a language to be regular over an alphabet, it is natural to ask what sort of closure properties this collection has under some of the defined properties, i.e. is it closed under unions, intersection, reversal, etc. To answer these questions we are led to some new constructions.
The most natural question to ask is whether the union of regular languages is regular. In this case we must restrict ourselves to finite unions, since every language is the union of finite languages. It is in fact true that the union of two regular languages over a common alphabet is regular. To show this we introduce the notion of the *cross product* of DFAs.
Let $\Sigma = \{a_1, \ldots, a_m\}$ be an alphabet and suppose that we are given two DFA's $D_1 = (Q_1, \Sigma, \delta_1, q_{0,1}, F_1)$ and $D_2 = (Q_2, \Sigma, \delta_2, q_{0,2}, F_2)$, accepting L_1 and L_2 respectively. We will show that the union, the intersection, and the relative complement of regular languages is a regular language.

First we will explain how to construct a DFA accepting the intersection $L_1 \cap L_2$. The idea is to construct a DFA simulating D_1 and D_2 in 'parallel'. This can be done by using states which are pairs $(p_1, p_2) \in Q_1 \times Q_2$. Define a DFA, D, as follows:

$$D = (Q_1 \times Q_2, \Sigma, \delta, (q_{0,1}, q_{0,2}), F_1 \times F_2),$$

where the transition function $\delta : (Q_1 \times Q_2) \times \Sigma \to Q_1 \times Q_2$ is defined by

$$\delta((p_1, p_2),\, a) = (\delta_1(p_1, a),\, \delta_2(p_2, a)),$$

for all $p_1 \in Q_1$, $p_2 \in Q_2$, and $a \in \Sigma$.

Clearly D is a DFA, since D_1 and D_2 are. Also, by the definition of δ, we have

$$\delta^*((p_1, p_2),\, w) = ((\delta_1^*(p_1, w),\, \delta_2^*(p_2, w)),$$

for all $p_1 \in Q_1$, $p_2 \in Q_2$, and $w \in \Sigma^*$.

Example 1.8.1 A product of two DFAs with two states each is given below.

We have that $w \in L(D_1) \cap L(D_2)$ iff $w \in L(D_1)$ and $w \in L(D_2)$,iff $\delta_1^*(q_{0,1}, w) \in F_1$ and $\delta_2^*(q_{0,2}, w) \in F_2$, iff $\delta^*((q_{0,1}, q_{0,2}),\ , w) \in F_1 \times F_2$, iff $w \in L(D)$. Thus $L(D) = L(D_1) \cap L(D_2)$.

We can now modify D very easily to accept $L(D_1) \cup L(D_2)$. We change the set of final to $(F_1 \times Q_2) \cup (Q_1 \times F_2)$. Then

cross product figure

Figure 1.19:

$$
\begin{gathered}
w \in L(D_1) \cup L(D_2) \text{iff} \\
w \in L(D_1) \text{ or } w \in L(D_2) \text{iff} \\
\delta_1^*(q_{0,1}, w) \in F_1 \text{ or} \\
\delta_2^*(q_{0,2}, w) \in F_2, \text{iff} \\
\delta^*((q_{0,1}, q_{0,2}),\, w) \in (F_1 \times Q_2) \cup (Q_1 \times F_2), \text{iff} \\
w \in L(D).
\end{gathered}
$$

Thus, $L(D) = L(D_1) \cup L(D_2)$.

We can also modify D to accept $L(D_1) - L(D_2)$. We change the set of final states to $F_1 \times (Q_2 - F_2)$. Then

$$
\begin{gathered}
w \in L(D_1) - L(D_2) \text{iff} \\
w \in L(D_1) \text{ and } w \notin L(D_2), \text{iff} \\
\delta_1^*(q_{0,1}, w) \in F_1 \text{ and} \delta_2^*(q_{0,2}, w) \notin F_2, \text{iff} \\
\delta^*((q_{0,1}, q_{0,2}),\, w) \in F_1 \times (Q_2 - F_2), \text{iff} \\
w \in L(D).
\end{gathered}
$$

Thus, $L(D) = L(D_1) - L(D_2)$.

In all cases if D_1 has n_1 states and D_2 has n_2 states then the DFA D has $n_1 n_2$ states.

Exercises

1 A morphism between two DFA's $D_1 = (Q_1, \Sigma, \delta_1, q_{0,1}, F_1)$ and $D_2 = (Q_2, \Sigma, \delta_2, q_{0,2}, F_2)$ is a function $f : Q_1 \longrightarrow Q_2$ such that $f(\delta_1(q, a)) = \delta_2(f(q), a)$ for all $q \in Q_1, a \in \Sigma$, $f(q_{0,1}) = q_{0,2}$ and $f(F_1) \subset F_2$ (note that we require D_1 and D_2 to have the same alphabets). If the morphism is surjective then we say that D_2 is the *homomorphic image* of D_1.
i) Show that if $u \in \Sigma^*$ then $f(\delta_1(q, u)) = \delta_2(f(q), u)$ for all $q \in Q_1, a \in \Sigma$.
ii) Show that if $f : D_1 \longrightarrow D_2$ is a morphism, then $L(D_1) \subset L(D_2)$. When is $L(D_1) = L(D_2)$?
iii) By the *product* of D_1 and D_2, denoted $D_1 \times D_2$, we mean the product machine described above with $F_1 \times F_2$ as final states. Show D_1 and D_2 are homomorphic images of $D_1 \times D_2$.

2 A morphism f between D_1 and D_2 is called an *isomorphism* if it is bijective and $f(F_1) = F_2$. If a isomorphism between D_1 and D_2 exists then D_1 and D_2 are said to be *isomorphic* and we write $D_1 \approx D_2$.
i) Show that the inverse of an isomorphism is a morphism, hence is an isomorphism.
ii) Show that $D_1 \approx D_2$ implies that $L(D_1) = L(D_2)$.
iii) Show that for a given alphabet there is a machine I over that alphabet such that for any other DFA D over the same alphabet we have that $D \times I \approx I \times D \approx D$.

3 (For readers who like Category theory) Show that the collection of DFAs over a fixed alphabet forms a category. Are there products ? Coproducts ? Is there a terminal object ?

1.9 Non-Deterministic Finite Automata

There would appear to be a number of obvious variations on the definition of a DFA. One might allow for example, that the transition function not necessarily be defined for every state and letter, i.e. not every state has $|\Sigma|$ arrows coming out of it. Or perhaps we could allow many arrows out of a state labeled with the same letter. Whatever we define though, we have the same issue to confront that we did for DFAs, namely, what does mean for a machine to accept a language? After this is done, we will see that all the little variations don't give us any new languages, i.e. the new machines are not computing anything different then the old. Then why bother with them ? Because they make the constructions of some things much easier and often make proofs clear where they were not at all clear earlier.
The object that we define below has the features that we just described, plus we will now allow arrows to be labeled with ϵ. Roughly, the idea here is that you can move along an arrow labeled ϵ if that is what you want to do.

Definition 1.9.1 A *non-deterministic finite automata* (or NFA) is a five-tuple

$$(Q, \Sigma, \delta, q_0, F)$$

where Q and Σ are finite sets, called the *states* and the *alphabet*, $\delta : Q \times (\Sigma \cup \{\epsilon\}) \longrightarrow 2^Q$ is the *transition function*, $q_0 \in Q$ is a distinguished states, called the *start state* and F is a subset of the set of states, known as the set of *final states*.

There are three funny things about this definition. First of all is the non-determinism, i.e. given a string there are many paths to choose from. This is probably the hardest to swallow, as it seems too powerful. The next thing is that we have $\epsilon-$*moves*. One way to think of this is that we take our string and stick in epsilons wherever we want, and then feed it to the machine. Thirdly, since the range of δ is the power set of Q, a pair (q, a) may be mapped to the null set, which in terms of arrows and a diagram means that no arrow out the state p has the label a.
We would like to define the language accepted by N, and for this, we need to extend the transition function $\delta : Q \times (\Sigma \cup \{\epsilon\})2^Q$ to a function

$$\delta^* : Q \times \Sigma^* 2^Q.$$

The presence of ϵ-transitions (i.e., when $q \in \delta(p, \epsilon)$) causes technical problems. To overcome these problems we introduce the notion of ϵ-closure.

Definition 1.9.2 For any state p of an NFA we define the ϵ-*closure of* p to be set ϵ-closure(p) consisting of all states q such that there is a path from p to q whose spelling is ϵ. This means that either $q = p$, or that all the edges on the path from p to q have the label ϵ.

We can compute ϵ-closure(p) using a sequence of approximations as follows. Define the sequence of sets of states $(\epsilon\text{-clo}_i(p))_{i\geq 0}$ as follows:

$$\begin{aligned} \epsilon\text{-clo}_0(p) &= \{p\}, \\ \epsilon\text{-clo}_{i+1}(p) &= \epsilon\text{-clo}_i(p) \cup \{q \in Q \mid \exists s \in \epsilon\text{-clo}_i(p),\ q \in \delta(s, \epsilon)\}. \end{aligned}$$

Since $\epsilon\text{-clo}_i(p) \subseteq \epsilon\text{-clo}_{i+1}(p)$, $\epsilon\text{-clo}_i(p) \subseteq Q$, for all $i \geq 0$, and Q is finite, there is a smallest i, say i_0, such that

$$\epsilon\text{-clo}_{i_0}(p) = \epsilon\text{-clo}_{i_0+1}(p),$$

and it is immediately verified that

$$\epsilon\text{-closure}(p) = \epsilon\text{-clo}_{i_0}(p).$$

(It should be noted that there are more efficient ways of computing ϵ-closure(p), for example, using a stack (basically, a kind of depth-first search.)) When N has no ϵ-transitions, i.e., when $\delta(p, \epsilon) = \emptyset$ for all $p \in Q$ (which means that δ can be viewed as a function $\delta : Q \times \Sigma 2^Q$) we have

$$\epsilon\text{-closure}(p) = \{p\}.$$

Given a subset S of Q, we define ϵ-closure(S) as

$$\epsilon\text{-closure}(S) = \bigcup_{p \in S} \epsilon\text{-closure}(p).$$

When N has no ϵ-transitions we have

$$\epsilon\text{-closure}(S) = S.$$

We are now ready to define the extension $\delta^* : Q \times \Sigma^* 2^Q$ of the transition function $\delta : Q \times (\Sigma \cup \{\epsilon\})2^Q$. The intuition behind the extended transition function is that $\delta^*(p, w)$ is the set of all states reachable from p by a path whose spelling is w.

Definition 1.9.3 Given an NFA $N = (Q, \Sigma, \delta, q_0, F)$ (with ϵ-transitions), the *extended transition function* $\delta^* : Q \times \Sigma^* 2^Q$ is defined as follows: for every $p \in Q$, every $u \in \Sigma^*$ and every $a \in \Sigma$,

$$\begin{aligned} \delta^*(p, \epsilon) &= \epsilon\text{-closure}(\{p\}), \\ \delta^*(p, ua) &= \epsilon\text{-closure}\Big(\bigcup_{s \in \delta^*(p,u)} \delta(s, a) \Big). \end{aligned}$$

The *language $L(N)$ accepted by an NFA N* is the set

$$L(N) = \{w \in \Sigma^* \mid \delta^*(q_0, w) \cap F \neq \emptyset\}.$$

We now extend $\delta^* : Q \times \Sigma^* 2^Q$ to a function $\widehat{\delta} : 2^Q \times \Sigma^* 2^Q$ defined as follows: for every subset S of Q, for every $w \in \Sigma^*$,

$$\widehat{\delta}(S, w) = \bigcup_{p \in S} \delta^*(p, w).$$

Let $\mathcal{Q}$ be the subset of 2^Q consisting of those subsets S of Q that are ϵ-closed, i.e., such that $S = \epsilon$-closure(S). If we consider the restriction $\Delta : \mathcal{Q} \times \pm\mathcal{Q}$ of $\widehat{\delta} : 2^Q \times \Sigma^* 2^Q$ to $\mathcal{Q}$ and Σ, we observe that Δ is the transition function of a DFA. Indeed, this is the transition function of a DFA accepting $L(N)$. It is easy to show that Δ is defined directly as follows (on subsets S in $\mathcal{Q}$):

$$\Delta(S, a) = \epsilon\text{-closure}\Big(\bigcup_{s \in S} \delta(s, a) \Big).$$

Then, the DFA D is defined as follows:

$$D = (\mathcal{Q}, \pm, \cdot, \epsilon\text{-closure}(\{\amalg_\prime\}), \mathcal{F}),$$

where $\mathcal{F} = \{\mathcal{S} \in \mathcal{Q} \mid \mathcal{S} \cap \mathcal{F} \neq \emptyset\}$.

It is not difficult to show that $L(D) = L(N)$, that is, D is a DFA accepting $L(N)$. Thus we have converted the NFA N into a DFA D (and gotten rid of ϵ-transitions).

The states of the DFA D equivalent to N are ϵ-closed subsets of Q. For this reason the above construction is often called the "subset construction". It is due to Rabin and Scott. Although theoretically fine, the method may construct useless sets S that are not reachable from the start state ϵ-closure($\{q_0\}$). A more economical construction is given below.

An Algorithm to convert an NFA into a DFA: The "subset construction"

Given an input NFA $N = (Q, \Sigma, \delta, q_0, F)$, a DFA $D = (K, \Sigma, \Delta, S_0, \mathcal{F})$ is constructed. It is assumed that Δ is a list of triples (S, a, T), with $S, T \in K$, and $a \in \Sigma$.

$S_0 := \epsilon\text{-closure}(\{q_0\})$; $K := \{S_0\}$; $total := 1$;
$marked := 0$; $\Delta := nil$;
while $marked < total$ **do**;
  $marked := marked + 1$; $S := K[marked]$;
  for each $a \in \Sigma$ **do**
    $U := \cup_{s \in S}\, \delta(s, a)$; $T := \epsilon\text{-closure}(U)$;
    if $T \notin K$ **then**
      $total := total + 1$; $K[total] := T$
    endif;
    $\Delta := append[\Delta, (S, a, T)]$
  endfor
endwhile;
$\mathcal{F} := \{S \in K \mid S \cap F \neq \emptyset\}$

Exercises

1 Implement the "subset algorithm" for converting an NFA with ϵ-transitions to a DFA. Pay particular attention to the input and output format. Explain your data structures. How do you deal with set comparisons?

2 Let $\Sigma = \{a_1, \ldots, a_n\}$ be an alphabet of n symbols.
(i) Construct an NFA with $2n + 1$ states accepting the set L_n of strings over Σ such that, every string in L_n has an odd number of a_i, for some $a_i \in \Sigma$. Equivalently, if L_n^i is the set of all strings over Σ with an odd number of a_i, then $L_n = L_n^1 \cup \cdots \cup L_n^n$.
(ii) Prove that there is a DFA with 2^n states accepting the language L_n.

3 Prove that every DFA accepting L_n (from problem 2) has at least 2^n states. *Hint*: If a DFA D with $k < 2^n$ states accepts L_n, show that there are two strings u, v with the property that, for some $a_i \in \Sigma$, u contains an odd number of a_i's, v contains an even number of a_i's, and D ends in the same state after processing u and v. From this, conclude that D accepts incorrect strings.

4 (i) Given two alphabets Σ and Ω with $\Sigma \subset \Omega$, then we may define a map $e : \Omega^* \longrightarrow \Sigma^*$ by simply taking a string in Ω^* and removing all occurrences of elements of $\Omega - \Sigma$, i.e. just erase the letters that aren't in Ω. Show that if L is regular over Ω then $e(L)$ is regular over Σ.
(ii) On the other hand, suppose that $L \subset \Sigma^*$. Then for any $a \in \Omega$, define the *a-blow-up of* L to be $L^a = \{a^{k_1}u_1...a^{k_n}u_n \,|u_1...u_n \in Ł \text{ and } k_1, .., k_n \geq 0\}$. Show that L^a is regular if L is regular.
(iii) Again, suppose that $L \subset \Sigma^*$. Define the *blow-up of L relative to* Ω to be $L^\Omega = \{w_1u_1...w_nu_n|u_1...u_n \in Ł \text{ and } w_1, ...w_n \in \Omega^*\}$. Is L^Ω regular over Ω ?

1.10 Directed Graphs and Paths

It is often useful to view DFA's and NFA's as labeled directed graphs. The purpose of this section is to review some of these concepts. We begin with directed graphs. Our definition is very flexible, since it allows parallel edges and self loops.

Definition 1.10.1 A *directed graph* is a quadruple $G = (V, E, s, t)$, where V is a set of *vertices, or nodes*, E is a set of *edges, or arcs*, and $s, t: E \to V$ are two functions, s being called the *source* function, and t the *target* function. Given an edge $e \in E$, we also call $s(e)$ the *origin* (or *source*) of e, and $t(e)$ the *endpoint* (or *target*) of e.

Remark The functions s, t need not be injective or surjective. Thus, we allow "isolated vertices".

Example 1.10.2 Let G be the directed graph defined such that
$E = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8\}$, $V = \{v_1, v_2, v_3, v_4, v_5, v_6\}$, and

$$s(e_1) = v_1, s(e_2) = v_2, s(e_3) = v_3, s(e_4) = v_4, s(e_5) = v_2, s(e_6) = v_5, s(e_7) = v_5, s(e_8) = v_5,$$

$$t(e_1) = v_2, t(e_2) = v_3, t(e_3) = v_4, t(e_4) = v_2, t(e_5) = v_5, t(e_6) = v_5, t(e_7) = v_6, t(e_8) = v_6.$$

Such a graph can be represented by the following diagram:

In drawing directed graphs, we will usually omit edge names (the e_i), and sometimes even the node names (the v_j). We now define paths in a directed graph.

Definition 1.10.3 Given a directed graph $G = (V, E, s, t)$ for any two nodes $u, v \in V$, a *path from* u *to* v is a triple $\pi = (u, e_1 \dots e_n, v)$, where $e_1 \dots e_n$ is a string (sequence) of edges in E such that, $s(e_1) = u$, $t(e_n) = v$, and $t(e_i) = s(e_{i+1})$, for all i such that $1 \leq i \leq n-1$. When $n = 0$, we must have $u = v$, and the path (u, ϵ, u) is called the *null path from* u *to* u. The number n is the *length* of the path. We also call u the *source* (or *origin*) of the path, and v the *target* (or *endpoint*) of the path. When there is a nonnull path π from u to v, we say that u *and* v *are connected*.

Remark In a path $\pi = (u, e_1 \dots e_n, v)$, the expression $e_1 \dots e_n$ is a **sequence**, and thus, the e_i are **not** necessarily distinct.

For example, the following are paths:

$$\pi_1 = (v_1, e_1e_5e_7, v_6),$$

$$\pi_2 = (v_2, e_2e_3e_4e_2e_3e_4e_2e_3e_4, v_2),$$

and

$$\pi_2 = (v_1, e_1e_2e_3e_4e_2e_3e_4e_5e_6e_6e_8, v_6).$$

Clearly, π_2 and π_3 are of a different nature from π_1. Indeed, they contain cycles. This is formalized as follows.

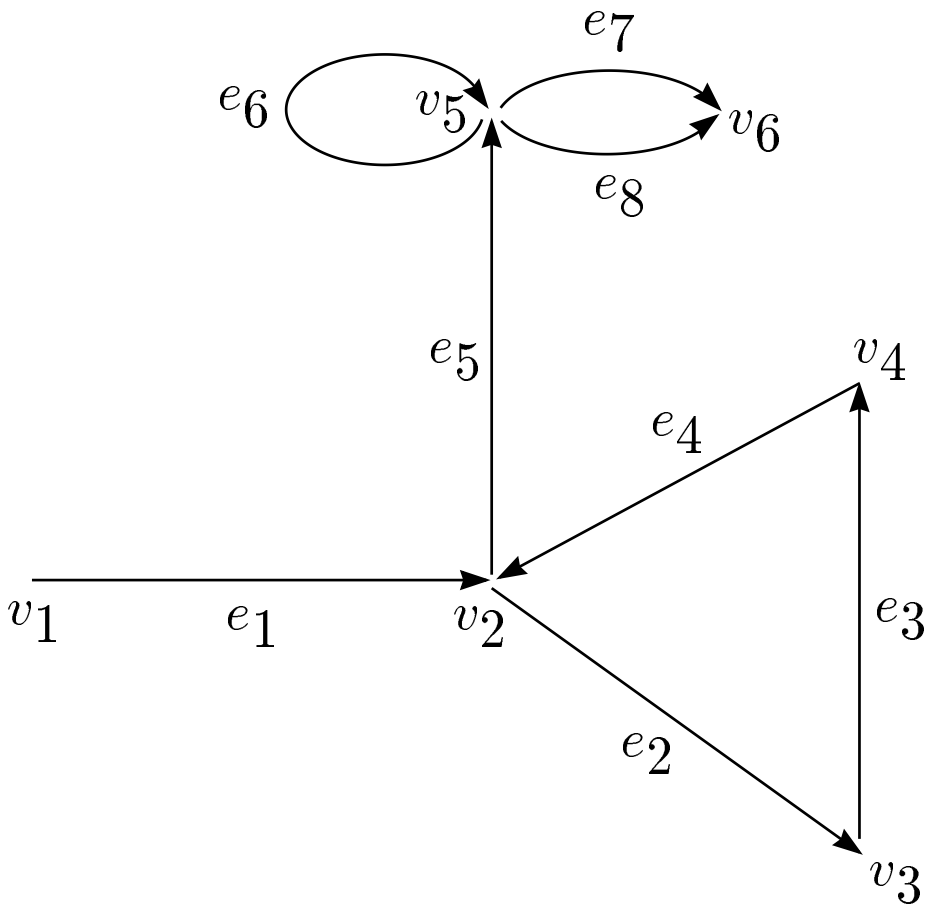


Figure 1.20:

Definition 1.10.4 Given a directed graph $G = (V, E, s, t)$, for any node $u \in E$ a *cycle (or loop) through* u is a nonnull path of the form $\pi = (u, e_1 \dots e_n, u)$ (equivalently, $t(e_n) = s(e_1)$). More generally, a nonnull path $\pi = (u, e_1 \dots e_n, v)$ *contains a cycle* iff for some i, j, with $1 \leq i \leq j \leq n$, $t(e_j) = s(e_i)$. In this case, letting $w = t(e_j) = s(e_i)$, the path $(w, e_i \dots e_j, w)$ is a cycle through w. A path π is *acyclic* iff it does not contain any cycle. Note that each null path (u, ϵ, u) is acyclic.

Obviously, a cycle $\pi = (u, e_1 \dots e_n, u)$ through u is also a cycle through every node $t(e_i)$. Also, a path π may contain several different cycles. Paths can be concatenated as follows.

Definition 1.10.5 Given a directed graph $G = (V, E, s, t)$, two paths $\pi_1 = (u, e_1 \dots e_m, v)$ and $\pi_2 = (u', e'_1 \dots e'_n, v')$ can be *concatenated* provided that $v = u'$, in which case their *concatenation* is the path

$$\pi_1 \pi_2 = (u, e_1 \dots e_m e'_1 \dots e'_n, v').$$

It is immediately verified that the concatenation of paths is associative, and that the composition of the path $\pi = (u, e_1 \dots e_m, v)$ with the null path (u, ϵ, u) or with the null path (v, ϵ, v) is the path π itself.

The following fact, although almost trivial, is used all the time, and is worth stating precisely.

Lemma 1.10.6 Given a directed graph $G = (V, E, s, t)$, if the set of nodes V has $m \geq 1$ elements (in particular, it is finite), then every path π of length at least m contains some cycle.

Proof. Let $\pi = (u, e_1 \dots e_n, v)$. By the hypothesis, $n \geq m$. Consider the sequence of nodes

$$(u, t(e_1), \dots, t(e_{n-1}), t(e_n) = v).$$

This sequence contains $n + 1$ elements. Since $n \geq m$, we have $n + 1 > m$, and by the so-called "pigeonhole principle", since V only contains m distinct nodes, some node in the sequence must appear twice. This shows that either $t(e_j) = u = s(e_1)$ for some j with $1 \leq j \leq n$, or $t(e_j) = t(e_i)$, for some i, j, with $1 \leq i < j \leq n$, and thus $t(e_j) = s(e_{i+1})$, with $1 \leq i < j \leq n$. Combining both cases, we have $t(e_j) = s(e_i)$ for some i, j, with $1 \leq i \leq j \leq n$, which yields a cycle. □

A consequence of lemma 1.10.6 is that in a finite graph with m nodes, given any two nodes $u, v \in V$, in order to find out whether there is a path from u to v, it is enough to consider paths of length $\leq m - 1$. Indeed, if there is path between u and v, then there is some path π of minimal length (not necessarily unique, but this doesn't matter). If this minimal path has length at least m, then by the lemma, it contains a cycle. However, by deleting this cycle from the path π, we get an even shorter path from u to v, contradicting the minimality of π.

Exercises

1 Let $D = (Q, \Sigma, \delta, q_0, F)$ be a DFA. Suppose that every path in the graph of D from the start state to some final state is acyclic. Does it follow that $L(D)$ is a finite language?

1.11 Labeled Graphs and Automata

Definition 1.11.1 A *labeled directed graph* is a tuple $G = (V, E, L, s, t, \lambda)$, where V is a set of *vertices, or nodes*, E is a set of *edges, or arcs*, L is a set of *labels*, $s, t \colon E \to V$ are two functions, s being called the *source* function, and t the *target* function, and $\lambda \colon E \to L$ is the *labeling function*. Given an edge $e \in E$, we also call $s(e)$ the *origin* (or *source*) of e, $t(e)$ the *endpoint* (or *target*) of e, and $\lambda(e)$ the *label* of e.

Note that the function λ need not be injective or surjective. Thus, distinct edges may have the same label.

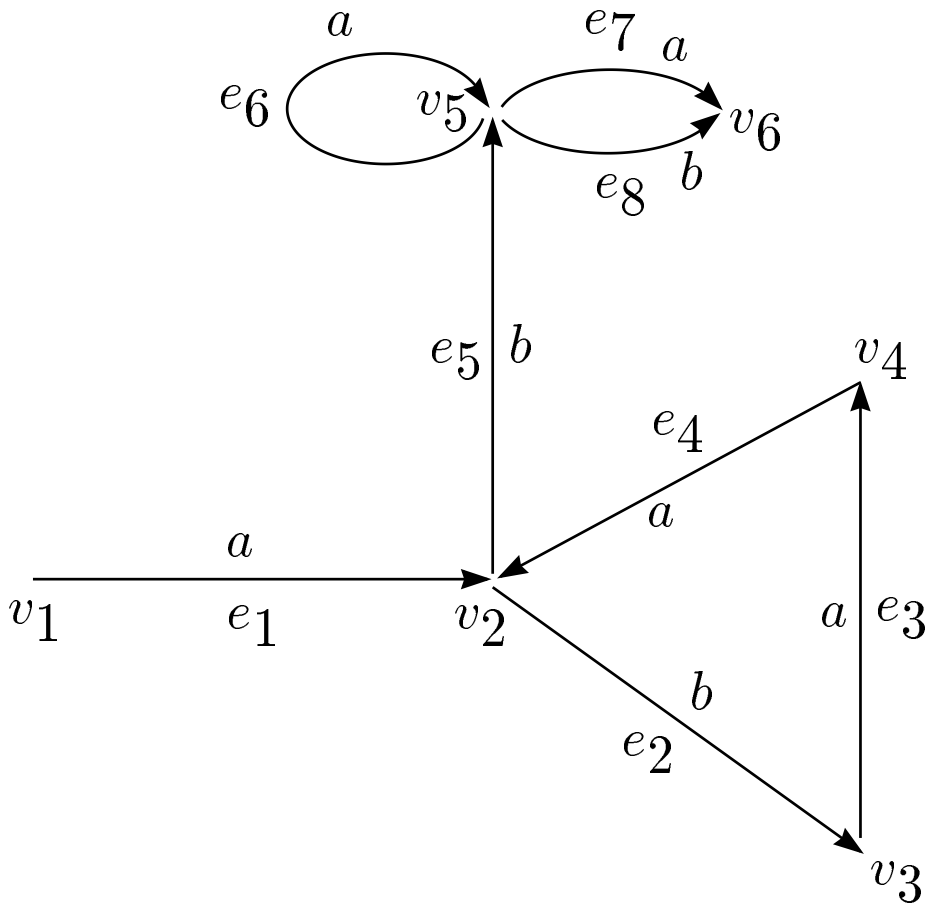


Figure 1.21:

Example 1.11.2 *Let G be the directed graph defined such that*

$E = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8\}$, $V = \{v_1, v_2, v_3, v_4, v_5, v_6\}$, $L = \{a, b\}$, *and*

$$s(e_1) = v_1, s(e_2) = v_2, s(e_3) = v_3, s(e_4) = v_4, s(e_5) = v_2, s(e_6) = v_5, s(e_7) = v_5, s(e_8) = v_5,$$

$$t(e_1) = v_2, t(e_2) = v_3, t(e_3) = v_4, t(e_4) = v_2, t(e_5) = v_5, t(e_6) = v_5, t(e_7) = v_6, t(e_8) = v_6,$$

$$\lambda(e_1) = a, \lambda(e_2) = b, \lambda(e_3) = a, \lambda(e_4) = a, \lambda(e_5) = b, \lambda(e_6) = a, \lambda(e_7) = a, \lambda(e_8) = b.$$

Such a labeled graph can be represented by the following diagram:

In drawing labeled graphs, we will usually omit edge names (the e_i), and sometimes even the node names (the v_j). Paths, cycles, and concatenation of paths are defined just as before (that is, we ignore the labels). However, we can now define the *spelling* of a path.

Definition 1.11.3 Given a labeled directed graph $G = (V, E, L, s, t, \lambda)$ for any two nodes $u, v \in V$, for any path $\pi = (u, e_1 \dots e_n, v)$, the *spelling of the path* π is the string of labels

$$\lambda(e_1) \dots \lambda(e_n).$$

When $n = 0$, the spelling of the null path (u, ϵ, u) is the null string ϵ.

For example, the spelling of the path

$$\pi_2 = (v_1, e_1e_2e_3e_4e_2e_3e_4e_5e_6e_6e_8, v_6)$$

is

$$abaabaabaab.$$

Every DFA and every NFA can be viewed as a labeled graph, in such a way that the set of spellings of paths from the start state to some final state is the language accepted by the automaton in question.

Given a DFA $D = (Q, \Sigma, \delta, q_0, F)$, where $\delta: Q \times \Sigma \to Q$, we associate the labeled directed graph $G_D = (V, E, L, s, t, \lambda)$ defined as follows:

$$V = Q, \quad E = \{(p, a, q) \mid q = \delta(p, a),\ p, q \in Q,\ a \in \Sigma\},$$

$L = \Sigma$, $s((p, a, q)) = p$, $t((p, a, q)) = q$, and $\lambda((p, a, q)) = a$.

Such labeled graphs have a special structure that can easily be characterized.

It is easily shown that a string $w \in \Sigma^*$ is in the language $L(D) = \{w \in \Sigma^* \mid \delta^*(q_0, w) \in F\}$ iff w is the spelling of some path in G_D from q_0 to some final state.

Similarly, given an NFA $N = (Q, \Sigma, \delta, q_0, F)$, where $\delta: Q \times (\Sigma \cup \{\epsilon\}) \to 2^Q$, we associate the labeled directed graph $G_N = (V, E, L, s, t, \lambda)$ defined as follows:

$$V = Q, \quad E = \{(p, a, q) \mid q \in \delta(p, a),\ p, q \in Q,\ a \in \Sigma \cup \{\epsilon\}\},$$

$L = \Sigma \cup \{\epsilon\}$, $s((p, a, q)) = p$, $t((p, a, q)) = q$, $\lambda((p, a, q)) = a$.

Remark When N has no ϵ-transitions, we can let $L = \Sigma$. Such labeled graphs have also a special structure that can easily be characterized. A string $w \in \Sigma^*$ is in the language $L(N) = \{w \in \Sigma^* \mid \delta^*(q_0, w) \cap F \neq \emptyset\}$ iff w is the spelling of some path in G_N from q_0 to some final state.
Conversely, if we are given a labeled directed graph it may be viewed as an NFA if we pick a start state and a set of final states. The relationship between NFAs and labeled directed graphs could be made more formal then this, say, using category theory, but it is sufficiently simple that that is probably unnecessary.
Let $\Sigma = \{a_1, \ldots, a_m\}$ be an alphabet. We define a family R_n, of sets of languages as follows:

$$R_0 = \{\{a_1\}, \ldots, \{a_m\}, \emptyset, \{\epsilon\}\},$$

$$R_{n+1} = R_n \cup \{L_1 \cup L_2,\ L_1L_2,\ L^*\ | L_1, L_2, L \in R_n\}.$$

We define R as

$$R = \bigcup_{n \geq 0} R_n = R_0 \cup R_1 \cup R_2 \cup \cdots.$$

R is the **family of regular languages over** Σ. The reason for this name is because R is precisely the set of regular languages over Σ. The elements of R_0 are called the **atomic languages**. In this section we show that the regular languages are those languages that are 'finitely generated' using the operations union, concatenation and Kleene-*. A regular expression is a natural way of denoting how these operations are used to generate a language.
One thing to be careful about is that R depends on the alphabet Σ, although our notation doesn't reflect this fact. If for any reason it is unclear from the context which R we are referring to, we can use the notation $R(\Sigma)$ to denote R.

Example 1.11.4 Suppose we take $\Sigma = \{a, b\}$. Then

$$R_1 = \{\{a\}, \{b\}, \emptyset, \{\epsilon\}, \{a, b\}, \{ab\}, \{ba\}, \{\epsilon, a, a^2, ...\}, \{\epsilon, b, b^2, ...\}\}.$$

Observe that in general, R_n is a finite set. In this case it contains 9 languages, 7 of which are finite and two infinite ones.

Lemma 1.11.5 The family R is the smallest family of languages that contains the (atomic) languages $\{a_1\}$, $\ldots, \{a_m\}$, $\emptyset$, $\{\epsilon\}$, and is closed under union, concatenation, and Kleene $*$.

Proof Use induction on n.

Note that a given language may be "built" up in different ways. For example,

$$\{a, b\}^* = (\{a\}^*\{b\}^*)^*.$$

Given an alphabet $\Sigma = \{a_1, \ldots, a_m\}$, consider the new alphabet

$$\mathcal{D} = \{+, \cdot, *, (,), \emptyset, \epsilon\}.$$

Next, we define a family $\mathcal{R}_n$ of languages over $\mathcal{D}$ as follows:

$$\mathcal{R}_0 = \{a_1, \ldots, a_m, \emptyset, \epsilon\},$$

$$\mathcal{R}_{n+1} = \mathcal{R}_n \cup \{(S + T),\ (S \cdot T),\ (U^*)\ | S, T, U \in \mathcal{R}_n\}.$$

Finally, we define $\mathcal{R}$ as

$$\mathcal{R} = \mathcal{R}_0 \cup \mathcal{R}_1 \cup \mathcal{R}_2 \cup \cdots$$

$\mathcal{R}$ is the set of **regular expressions** (over Σ).

Lemma 1.11.6 The language $\mathcal{R}$ is the smallest language which contains the symbols $a_1, \ldots, a_m$, $\emptyset$, ϵ, from $\mathcal{D}$, such that $(S + T)$, $(S \cdot T)$, and U^* belong to $\mathcal{R}$, when $S, T, U \in \mathcal{R}$.

Proof Exercise.

For simplicity of notation we write (R_1R_2) instead of $(R_1 \cdot R_2)$.

Example 1.11.7 $R = (a + b)^*$, $S = (a^*b^*)^*$.

Every regular expression $S \in \mathcal{R}$ can be viewed as the **name**, or **denotation**, of some regular language $L \in R$. Similarly, every regular language $L \in R$ is the **interpretation** (or **meaning**) of some regular expression $S \in \mathcal{R}$. This is made rigorous by defining a function

$$\mathcal{L}: \mathcal{R} \to \mathcal{P}(\Sigma^*),$$

where $\mathcal{P}(\Sigma^*)$ is the set of subsets of Σ^*. We may think of $\mathcal{L}$ as standing for 'the language denoted by'. This function can be defined recursively by the equations

$$\begin{aligned}
\mathcal{L}[a_i] &= \{a_i\}, \\
\mathcal{L}[\emptyset] &= \emptyset, \\
\mathcal{L}[\epsilon] &= \{\epsilon\}, \\
\mathcal{L}[(S + T)] &= \mathcal{L}[S] \cup \mathcal{L}[T], \\
\mathcal{L}[(ST)] &= \mathcal{L}[S]\mathcal{L}[T], \\
\mathcal{L}[U^*] &= \mathcal{L}[U]^*.
\end{aligned}$$

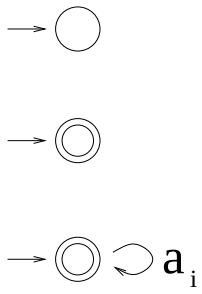


Figure 1.22: Here we see the three types of machines that accept the atomic languages. The top machine accepts the empty set because it has no final states. The middle machine accepts only ϵ since it has no arrows leaving it or going into it. The last machine accepts only a fixed letter a_i (i.e. there is one machine for each letter).

Remark The function $\mathcal{L}$ is not injective. For example, if $S = (a + b)^*$, $T = (a^*b^*)^*$, then

$$\mathcal{L}[S] = \mathcal{L}[T] = \{a, b\}^*.$$

For simplicity we often denote $\mathcal{L}[S]$ as L_S.

Theorem 1.11.8 For every regular expression $S \in \mathcal{R}$, the language $\mathcal{L}[S]$ is regular ($\mathcal{L}[S] \in R$). Conversely, for every regular language $L \in R$, there is some regular expression $S \in \mathcal{R}$ such that $L = \mathcal{L}[S]$. In other words, the range of $\mathcal{L}$ is exactly R.

We break the theorem up into two lemmas, which actually say a bit more then the theorem.

Lemma 1.11.9 There is an algorithm, which, given any regular expression $S \in \mathcal{R}$, constructs an NFA N_S accepting L_S, i.e. such that $L_S = L(N_S)$.

Proof The proof is by induction on the strata of $\mathcal{R}$. We show that for each n if $R \in \mathcal{R}_n$ then there is an NFA that accepts the language of R, and that this NFA has the properties that
(i) There are no edges entering the start state, and
(ii) there is one final state, which has no outgoing edges.

Without loss of generality, assume that $\Sigma = \{a_1, ..., a_k\}$. NFAs for $\mathcal{R}_0$ are given in figure (1.22).
Next, suppose that our hypothesis is true for $\mathcal{R}_m$ where $m < n$. Let $R \in \mathcal{R}_n$ be a regular expression. Then R is either the Kleene-* of a regular expression in $\mathcal{R}_{n-1}$ or the sum or product of two regular expressions in $\mathcal{R}_{n-1}$.
Suppose that $R = S^*, S \in \mathcal{R}_{n-1}$. Then by induction there is an NFA accepting the language denoted by S that has a single start state with no incoming edges and a single final state with no outgoing edges (see figure (1.23). This can always be achieved by adding a single start state and final state and using the appropriate epsilon transitions. In figure (1.23), and the figures to follow, we draw this type a machine with a "blob" in the middle and imagine that inside the blob is a collection of states and transitions.
To construct a machine that will recognize the language denoted by R we alter the above machine to create the machine that appears in figure (1.24).

Next, suppose that it is the case that $R = (ST)$, where S and T are in $\mathcal{R}_{n-1}$. Then by induction there are two NFAs accepting the languages denoted by S and T respectively, and as above me may assume that they have the form depicted in figure (1.23). From these two machines we construct an NFA that will accept the appropriate language - see figure (1.25).

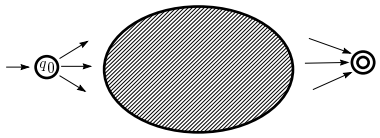


Figure 1.23: For any regular language there is an NFA accepting of the form depicted above, namely a single start and final state, each of which has no incoming arrows.

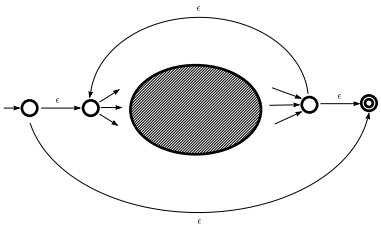


Figure 1.24: If the machine in figure (1.23) is altered to appear as above, then the language accepted by the new machine will be the Kleene-* of the language accepted by the machine in figure (1.23).

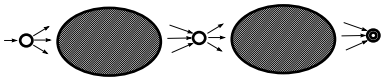


Figure 1.25: How to construct an NFA to accept the product of two languages, given an NFA for each of the languages.

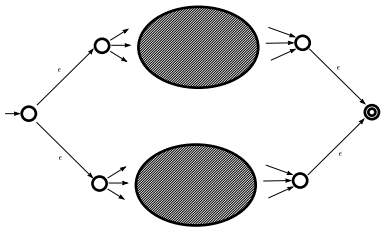


Figure 1.26:

Finally, suppose that it is the case that $R = (S + T), S, T \in \mathcal{R}_{n-1}$. Again, by induction there are two NFAs accepting the languages denoted by S and T respectively. which look like the above two. A machine corresponding to their sum is given in figure (1.26).
Of course, it is crucial that in each of the above three constructions the induction hypothesis is satisfied, i.e. the resulting machine has a start state with no incoming edges and a single final state with no outgoing edges. Therefore the proof of the first lemma is complete.

Lemma 1.11.10 There is an algorithm, which, given any NFA N, constructs a regular expression $S \in \mathcal{R}$, denoting $L(N)$, i.e., such that $L_S = L(N)$.

This is the **node elimination algorithm**. The idea is to allow more general labels on the edges of an NFA, namely, regular expressions. We will call these machines **generalized NFAs**.

Example 1.11.11 Consider a generalized NFA with two state p and q where q is final, and the edge between them is labeled by a regular expression S. The language accepted by this generalized NFA is exactly $\mathcal{L}(S)$!

The proof will proceed as follows. We start with an NFA and view it as a generalized NFA. We will then 'contract' this generalized NFA, step by step, preserving the recognized language at each step, until we have a generalized NFA which has only one edge, as in the above example. The expression that labels the edge will correspond to the language of the original NFA that we started with. But before we start this procedure we must put the NFA the 'normal form' that we used earlier.

Preprocessing, phase 1:

Consider a given NFA. If there are incoming edges to the start state then add a new start state with an ϵ-transition to the original start state.

If necessary, we need to add a new (unique) final state, with ϵ-transitions from each of the old final states to the new final state, if there are outgoing edges from any of the old final states.

At the end of this phase, the start state s is a source (no incoming edges), and the final state t is a sink (no outgoing edges).
Preprocessing, phase 2:
We need to "flatten" parallel edges. For any pair of states (p, q) ($p = q$ is possible), if there are k edges from p to q labeled $u_1, \ldots, u_k$, then create a single edge labeled with the regular expression

$$u_1 + \cdots + u_k.$$

For any pair of states (p, q) ($p = q$ is possible) such that there is **no** edge from p to q, we put an edge labeled $\emptyset$.
At the end of this phase, the resulting generalized NFA has the property that for any pair of states (p, q) (where $p = q$ is possible), there is a unique edge labeled with some regular expression denoted as $R_{p,q}$. When $R_{p,q} = \emptyset$, this really means that there is no edge from p to q in the original NFA N.
By interpreting each $R_{p,q}$ as a function call (really, a macro) to the NFA $N_{p,q}$ accepting $\mathcal{L}[R_{p,q}]$ (constructed using the previous algorithm), we can verify that the original language $L(N)$ is accepted by this new generalized NFA.
Node elimination This algorithm only applies if the generalized NFA has at least one node distinct from s and t.
Pick any node r distinct from s and t. For every pair (p, q) where $p \neq r$ and $q \neq r$, consider the "triangle" formed by these states. If $p \neq q$ then we may draw a diagram like the one in figure (1.27). Then, replace the label of the edge from p to q with the expression $R_{p,q} + R_{p,r}R^*_{r,r}R_{r,q}$ (see figure (1.28)).
We emphasize that this operation must be performed simultaneously for **every pair** (p, q) . Also, keep in mind that if in the original machine there was no edge from p to q then we now have an edge between them

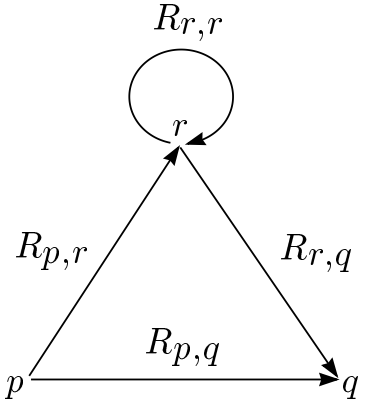


Figure 1.27:

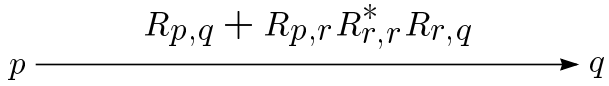


Figure 1.28:

labeled with the empty set. If we don't pay attention to other edges coming and going from r then our algorithm probably won't work properly.
At the end of this step delete the node r and all edges adjacent to r. Observe that it is possible that $p = q$, in which case the triangle is "flat". It is also possible that $p = s$ or $q = t$. Also, this step is performed for all **pairs** (p, q), which means that both (p, q) and (q, p) are considered (when $p \neq q$). Note that this step only has an effect if there are edges from p to r and from r to q in the original NFA N. Otherwise r can simply be deleted, as well as the edges adjacent to r. Other simplifications can be made. For example, when $R_{r,r} = \emptyset$ we can simplify $R_{p,r}R^*_{r,r}R_{r,q}$ to $R_{p,r}R_{r,q}$. When $R_{p,q} = \emptyset$ we have $R_{p,r}R^*_{r,r}R_{r,q}$.
The order in which the nodes are eliminated is irrelevant, although it effects the size of the final expression. The algorithm stops when the only remaining nodes are s and t. Then the label R of the edge from s to t is a regular expression denoting $L(N)$. This completes the proof of the second lemma.
In figure (1.29) we see an example of node elimination applied to a machine that accepts all binary strings whose second to last letter is the digit 1. Another example, involving simultaneous edges during the elimination process can be seen in figure (**??**).

Definition 1.11.12 Two regular expressions $S, T \in \mathcal{R}$ are *equivalent*, denoted as $S \cong T$, iff $\mathcal{L}[S] = \mathcal{L}[T]$.

It is easy to prove that $\cong$ is an equivalence relation. The relation $\cong$ satisfies some (nice) identities. For example:

$$\begin{aligned}((S + T) + U) &\cong (S + (T + U)),\\ ((ST)U) &\cong (S(TU)),\\ (S + T) &\cong (T + S),\\ (S^*S^*) &\cong S^*,\\ (S^*)^* &\cong S^*.\end{aligned}$$

Exercises

1 (a) Find a regular expression denoting the set of all strings over $\Sigma = \{a, b\}$ with no more than three a's. (b) Find a regular expression denoting the set of all strings over $\Sigma = \{a, b\}$ such that the number of a's is a multiple of three (including 0).

2 Let R be any regular language. Prove that the language

$$L = \{w \mid www \in R\}$$

is regular (this is a bit tricky!).

3 Recall that for two regular expressions R and S, $R \cong S$ means that $\mathcal{L}(R) = \mathcal{L}(S)$, i.e. R and S denote the same language.

(i) Show that if $R \cong S^*.T$, then $R \cong S.R + T$.

(ii) Assume that ϵ (the empty string) is not in the language denoted by the regular expression S.

Hint: Prove that $x \in R$ if and only if $x \in S^*.T$, by observing that $R \cong S.R + T$ implies that for every $k \geq 0$,

$$R \cong S^{k+1}.R + (S^k + S^{k-1} + \ldots + S^2 + S + \epsilon).T,$$

and that since $\epsilon \notin S$, every string in $S^{k+1}.R$ has length at least $k + 1$.

4 Recall that two regular expressions R and S are equivalent, denoted as $R \cong S$, iff they denote the same regular language $\mathcal{L}(R) = \mathcal{L}(S)$. Show that the following identities hold for regular expressions:

$$\begin{aligned}
(R + S).T &\cong R.T + S.T \\
T.(R + S) &\cong T.R + T.S \\
R^*.R^* &\cong R^* \\
(R^*)^* &\cong R^* \\
(R + S)^* &\cong (R^*.S)^*.R^*
\end{aligned}$$

5 Show that the recursion equations for $\mathcal{L}$ can be used to define $\mathcal{L}$. Give a rigorous proof !

1.12 The Theorem of Myhill and Nerode

The purpose of this section is to give one more characterization of the regular languages in terms of certain kinds of equivalence relations on strings. Pushing this characterization a bit further, we will be able to prove the existence of minimal DFA's.

Let $D = (Q, \Sigma, \delta, q_0, F)$ be a DFA. The DFA D may be redundant. For example, there may be states that are not accessible from the start state. This motivates the following definition. The set Q_{acc} of *accessible states* is the subset of Q defined as

$$Q_{acc} = \{p \in Q \mid \exists w \in \Sigma^*,\ \delta^*(q_0, w) = p\}.$$

The set Q_{acc} can be easily computed by stages. If $Q \neq Q_{acc}$, we can "clean up" D by deleting the states in $Q - Q_{acc}$ and restricting the transition function δ to Q_{acc}. We then get an 'equivalent' DFA D', i.e. $L(D) = L(D')$, where all the states of D' are accessible. Therefore, without loss of generality, we assume that we are dealing with DFA's such that $Q = Q_{acc}$.

Recall that an *equivalence relation* $\simeq$ on a set A is a relation which is *reflexive*, *symmetric*, and *transitive*. Given any $a \in A$, the set

$$\{b \in A \mid a \simeq b\}$$

is called the *equivalence class of* a, and it is denoted as $[a]_{\simeq}$, or even as $[a]$. Recall that for any two elements $a, b \in A$, $[a] \cap [b] = \emptyset$ iff $a \not\simeq b$, and $[a] = [b]$ iff $a \simeq b$. The set of equivalence classes associated with the equivalence relation $\simeq$ is a *partition* Π of A (also denoted as $A/\simeq$). This means that it is a family of nonempty pairwise disjoint sets whose union is equal to A itself. The equivalence classes are also called the *blocks* of the partition Π. The number of blocks in the partition Π is called the *index* of $\simeq$ (and Π).

Given any two equivalence relations $\simeq_1$ and $\simeq_2$ with associated partitions Π_1 and Π_2,

$$\simeq_1 \subseteq \simeq_2$$

iff every block of the partition Π_1 is contained in some block of the partition Π_2. Then every block of the partition Π_2 is the union of blocks of the partition Π_1 and we say that $\simeq_1$ is a *refinement* of $\simeq_2$ (and similarly, Π_1 is a refinement of Π_2). Note that Π_2 has at most as many blocks as Π_1 does.

We now define an equivalence relation on strings induced by a DFA. We say that two strings u, v are equivalent iff when feeding first u and then v to the DFA, u and v drive the DFA to the same state. From the point of view of the observer u and v have the same effect (reaching the same state).

Definition 1.12.1 Given a DFA $D = (Q, \Sigma, \delta, q_0, F)$ we define the relation $\simeq_D$ on Σ^* as follows: for any two strings $u, v \in \Sigma^*$,

$$u \simeq_D v \quad \text{iff} \quad \delta^*(q_0, u) = \delta^*(q_0, v).$$

The relation $\simeq_D$ turns out to have some interesting properties. In particular it is *right-invariant*, which means that for all $u, v, w \in \Sigma^*$ if $u \simeq v$ then $uw \simeq vw$.

Lemma 1.12.2 *Given any (accessible) DFA $D = (Q, \Sigma, \delta, q_0, F)$, the relation $\simeq_D$ is an equivalence relation which is right-invariant and has finite index. Furthermore, if Q has n states, then the index of $\simeq_D$ is n and every equivalence class of $\simeq_D$ is a regular language. Finally, $L(D)$ is the union of some of the equivalence classes of $\simeq_D$.*

Proof. The fact that $\simeq_D$ is an equivalence relation is trivial. To prove that $\simeq_D$ is right-invariant we need the fact that that for all $u, v \in \Sigma^*$and for all $p \in Q$,

$$\delta^*(p, uv) = \delta^*(\delta^*(p, u),\, v).$$

Then if $u \simeq_D v$, which means that $\delta^*(q_0, u) = \delta^*(q_0, v)$, we have that

$$\delta^*(q_0, uw) = \delta^*(\delta^*(q_0, u),\, w) = \delta^*(\delta^*(q_0, v),\, w) = \delta^*(q_0, vw),$$

which means that $uw \simeq_D vw$. Thus $\simeq_D$ is right-invariant. We still have to prove that $\simeq_D$ has index n. Define the function $f\colon \Sigma^* \to Q$ such that

$$f(u) = \delta^*(q_0, u).$$

Note that if $u \simeq_D v$, which means that $\delta^*(q_0, u) = \delta^*(q_0, v)$, then $f(u) = f(v)$. Thus the function $f\colon \Sigma^* \to Q$ induces a function $\widehat{f}\colon \Pi \to Q$ defined such that

$$\widehat{f}([u]) = f(u),$$

for every equivalence class $[u] \in \Pi$, where $\Pi = \Sigma^*/\simeq$ is the partition associated with $\simeq_D$. However, the function $\widehat{f}\colon \Pi \to Q$ is injective (one-to-one) since $\widehat{f}([u]) = \widehat{f}([v])$ means that $\delta^*(q_0, u) = \delta^*(q_0, v)$, which

means precisely that $u \simeq_D v$, i.e., $[u] = [v]$. Since Q has n states Π has at most n blocks. But since every state is accessible, Π has exactly n blocks.

Finally, every equivalence class of Π is a set of strings of the form

$$\{w \in \Sigma^* \mid \delta^*(q_0, w) = p\},$$

for some $p \in Q$, which is accepted by the DFA obtained from D by changing F to $\{p\}$. Thus every equivalence class is a regular language and $L(D)$ is the union of the equivalence classes corresponding to the final states in F. □

The remarkable fact due to Myhill and Nerode is that lemma 1.12.2 has a converse.

Lemma 1.12.3 *Given any equivalence relation $\simeq$ on Σ^*, if $\simeq$ is right-invariant and has finite index n then every equivalence class (block) in the partition Π associated with $\simeq$ is a regular language.*

Proof. Let $C_1, \ldots, C_n$ be the blocks of Π, and assume that $C_1 = [\epsilon]$ is the equivalence class of the empty string. First, we claim that for every block C_i and every $w \in \Sigma^*$ there is a unique block C_j such that $C_i w \subseteq C_j$, where $C_i w = \{uw \mid u \in C_i\}$.

For every $u \in C_i$ the string uw belongs to one and only one of the blocks of Π, say C_j. For any other string $v \in C_i$, by definition we have that $u \simeq v$ and so by right invariance it follows that $uw \simeq vw$. But since $uw \in C_j$ and C_j is an equivalence class we also have $vw \in C_j$. This proves the first claim.

We also claim that for every $w \in \Sigma^*$ and every block C_i,

$$C_1 w \subseteq C_i \quad \text{iff} \quad w \in C_i.$$

If $C_1 w \subseteq C_i$ then it follows that $\epsilon w = w \in C_i$ since $C_1 = [\epsilon]$. Conversely, if $w \in C_i$ then for any $v \in C_1 = [\epsilon]$ since $\epsilon \simeq v$. By right invariance we have $w \simeq vw$ and therefore $vw \in C_i$, which shows that $C_1 w \subseteq C_i$.

For every class C_i, let

$$D_i = (\{1, \ldots, n\}, \Sigma, \delta, 1, \{i\}),$$

where $\delta(i, a) = j$ iff $C_i a \subseteq C_j$.

Using claims 1 and 2 it is immediately verified that $L(D_i) = C_i$, proving that every block C_i is a regular language. □

In general it is false that $C_i a = C_j$ for some block C_j. We can only claim that $C_i a \subseteq C_j$.

We can combine lemma 1.12.2 and lemma 1.12.3 to get the following characterization of a regular language due to Myhill and Nerode.

Theorem 1.12.4 A language L (over an alphabet Σ) is a regular language iff it is the union of some of the equivalence classes of an equivalence relation $\simeq$ on Σ^* which is right-invariant and has finite index.

Theorem 1.12.4 can also be used to prove that certain languages are not regular. As an example we prove that $L = \{a^n b^n \mid n \geq 1\}$ is not regular.

Assume that L is regular. Then there is some equivalence relation $\simeq$ which is right-invariant and of finite index which has the property that L is the union of some of the classes of $\simeq$. Since the set

$$\{a, aa, aaa, \ldots, a^i, \ldots\}$$

is infinite and $\simeq$ has a finite number of classes, two of these strings must belong to the same class. This means that $a^i \simeq a^j$ for some $i \neq j$. But $\simeq$ is right invariant so by concatenating with b^i on the right we have that $a^i b^i \simeq a^j b^i$ for some $i \neq j$. However $a^i b^i \in L$ and since L is the union of classes of $\simeq$ we also have $a^j b^i \in L$ for $i \neq j$, which is absurd, given the definition of L. Therefore L is not regular.

Another useful tool for proving that languages are not regular is the so-called *pumping lemma*.

Lemma 1.12.5 *Given any DFA $D = (Q, \Sigma, \delta, q_0, F)$ where Q has m states, for every $w \in \Sigma^*$, if $w \in L(D)$ and $|w| \geq m$ then there exists a decomposition of w as $w = uxv$, where*

- $x \neq \epsilon$,
- $ux^iv \in L(D)$, *for all* $i \geq 0$, *and*
- $|ux| \leq m$.

Proof. Let $w = w_1 \dots w_n$. Since $|w| \geq m$, we have $n \geq m$. Since $w \in L(D)$, let

$$(q_0, q_1, \dots, q_n),$$

be the sequence of states in the accepting computation of w (where $q_n \in F$). Consider the subsequence

$$(q_0, q_1, \dots, q_m).$$

This sequence contains $m + 1$ states, but there are only m states in Q and thus we have $q_i = q_j$, for some i, j such that $0 \leq i < j \leq m$. Letting $u = w_1 \dots w_i$, $x = w_{i+1} \dots w_j$ and $v = w_{j+1} \dots w_n$ it is clear that the conditions of the lemma hold. □

The reader is urged to use the pumping lemma to prove that $L = \{a^n b^n \mid n \geq 1\}$ is not regular. The usefulness of the condition $|ux| \leq m$ lies in the fact that it reduces the number of legal decompositions uxv of w.

We now consider an equivalence relation associated with a language L.

Exercises

1 Let $D = (Q, \Sigma, \delta, q_0, F)$ be a DFA. Recall that a state $p \in Q$ is *accessible* iff there is some string $w \in \Sigma^*$, such that

$$\delta^*(q_0, w) = p,$$

i.e., there is some path from q_0 to p in D. Consider the following method for computing the set Q_{acc} of accessible states (of D): define the sequence of sets $Q^i_{acc} \subseteq Q$, where

$Q^0_{acc} = \{q_0\}$,

$Q^{i+1}_{acc} = \{q \in Q \mid \exists p \in Q^i_{acc}, \exists a \in \Sigma,\ q = \delta(p, a)\}$.

Let $\bar{Q}^i_{acc}$ denote the set of states reachable from q_0 by paths of length i.
(a) Find a regular expression denoting the set of all strings over $\Sigma = \{a, b\}$ with no more than three a's.
(b) Find a regular expression denoting the set of all strings over $\Sigma = \{a, b\}$ such that the number of a's is a multiple of three (including 0).

2 Let $D = (Q, \Sigma, \delta, q_0, F)$ be a deterministic finite automaton. Define the relations $\simeq$ and $\equiv$ on Σ^* as follows:

$$\begin{gathered} x \simeq y \quad \text{if and only if,} \quad \text{for all} \quad p \in Q, \\ \delta^*(p, x) \in F \quad \text{iff} \quad \delta^*(p, y) \in F, \end{gathered}$$

and

$$x \equiv y \quad \text{if and only if,} \quad \text{for all} \quad p \in Q, \quad \delta^*(p, x) = \delta^*(p, y).$$

(a) Show that $\simeq$ is a left-invariant equivalence relation and that $\equiv$ is an equivalence relation that is both left and right invariant. (A relation R on Σ^* is *left invariant* iff uRv implies that $wuRwv$ for all $w \in \Sigma^*$, and R is *right invariant* iff uRv implies that $uwRvw$ for all $w \in \Sigma^*$.)

(b) Let n be the number of states in Q (the set of states of D). Show that $\simeq$ has at most 2^n equivalence classes and that $\equiv$ has at most n^n equivalence classes.

(c) Given any language $L \subseteq \Sigma^*$, define the relations λ_L and μ_L on Σ^* as follows:

$$u \,\lambda_L\, v \quad \text{iff,} \quad \text{for all} \quad z \in \Sigma^*, \quad zu \in L \quad \text{iff} \quad zv \in L,$$

and

$$u \,\mu_L\, v \quad \text{iff,} \quad \text{for all} \quad x, y \in \Sigma^*, \quad xuy \in L \quad \text{iff} \quad xvy \in L.$$

Prove that λ_L is left-invariant, and that μ_L is left and right-invariant. Prove that if L is regular, then both λ_L and μ_L have a finite number of equivalence classes.

Hint: Show that the number of classes of λ_L is at most the number of classes of $\simeq$, and that the number of classes of μ_L is at most the number of classes of $\equiv$.

3 Let $\Sigma = \{a, b\}$. Classify all languages over Σ whose strings are of the form $a^i b^j$.

1.13 Minimal DFAs

Given any language L (not necessarily regular), we can define an equivalence relation ρ_L which is right-invariant, but not necessarily of finite index. However, when L is regular the relation ρ_L has finite index. In fact, this index is the size of a smallest DFA accepting L. This will lead us to a method for constructing minimal DFA's.

Definition 1.13.1 Given any language L (over Σ), we define the relation ρ_L on Σ^* as follows: for any two strings $u, v \in \Sigma^*$,

$$u \rho_L v \quad \text{iff} \quad \forall w \in \Sigma^* (uw \in L \quad \text{iff} \quad vw \in L).$$

We leave as an easy exercise to prove that ρ_L is an equivalence relation which is right-invariant. It is clear that L is the union of the equivalence classes of strings in L. When L is regular we have the following remarkable result.

Lemma 1.13.2 If L is a regular language and $D = (Q, \Sigma, \delta, q_0, F)$ an (accessible) DFA such that $L = L(D)$ then ρ_L is a right-invariant equivalence relation and we have $\simeq_D \subseteq \rho_L$. Furthermore, if ρ_L has m classes and Q has n states then $m \leq n$.

Proof. By definition, $u \simeq_D v$ iff $\delta^*(q_0, u) = \delta^*(q_0, v)$. Since $w \in L(D)$ iff $\delta^*(q_0, w) \in F$, the fact that $u \rho_L v$ can be expressed as

$$\begin{aligned} \forall w \in \Sigma^* (uw \in L \quad &\text{iff} \quad vw \in L), \qquad \text{iff,} \\ \forall w \in \Sigma^* (\delta^*(q_0, uw) \in F \quad &\text{iff} \quad \delta^*(q_0, vw) \in F), \qquad \text{iff} \\ \forall w \in \Sigma^* (\delta^*(\delta^*(q_0, u), w) \in F \quad &\text{iff} \quad \delta^*(\delta^*(q_0, v), w) \in F), \end{aligned}$$

and if $\delta^*(q_0, u) = \delta^*(q_0, v)$, this shows that $u \rho_L v$. Since the number of classes of $\simeq_D$ is n and $\simeq_D \subseteq \rho_L$ it follows that the equivalence relation ρ_L has fewer classes than $\simeq_D$ and $m \leq n$. □

Lemma 1.13.2 shows that when L is regular the index m of ρ_L is finite and it is a lower bound on the size of all DFA's accepting L. It remains to show that a DFA with m states accepting L exists. However, going back to the proof of lemma 1.12.3 starting with the right-invariant equivalence relation ρ_L of finite index m, if L is the union of the classes $C_{i_1}, \ldots, C_{i_k}$, then the DFA

$$D_{\rho_L} = (\{1, \ldots, m\}, \Sigma, \delta, 1, \{i_1, \ldots, i_k\}),$$

where $\delta(i,a) = j$ iff $C_i a \subseteq C_j$, is such that $L = L(D_{\rho_L})$. Thus D_{ρ_L} is a minimal DFA accepting L.

In the next section we give an algorithm which allows us to find D_{ρ_L} given any DFA D accepting L. This algorithms finds which states of D are equivalent.

Exercises

1 The existence of a minimal DFA follows immediately from the well-ordering of the natural numbers. The point of the previous section is the construction given for a minimal DFA. In fact, this DFA is unique up to isomorphism.

(i) Show that if D is minimal and $L = L(D)$ then $\rho_L = \simeq_D$.
(ii) Use (i) to show that a minimal DFA is unique up to isomorphism.
(iii) Show that if D is minimal and $L(\tilde{D}) = L(D)$ then D is the homomorphic image of $\tilde{D}$.

1.14 State Equivalence and Minimal DFA's

The proof of lemma 1.13.2 suggests the following definition of equivalence between states.

Definition 1.14.1 Given a DFA $D = (Q, \Sigma, \delta, q_0, F)$, the relation $\equiv$ on Q, called *state equivalence*, is defined as follows: for all $p, q \in Q$,

$$p \equiv q \quad \text{iff} \quad \forall w \in \Sigma^*(\delta^*(p,w) \in F \quad \text{iff} \quad \delta^*(q,w) \in F).$$

When $p \equiv q$ we say that *p and q are indistinguishable*.

It is trivial to verify that $\equiv$ is an equivalence relation, and that it satisfies the following property:

$$\text{if } p \equiv q \text{ then } \delta(p,a) \equiv \delta(q,a),$$

for all $a \in \Sigma$. The following lemma shows the relationship between ρ_L and $\equiv$.

Lemma 1.14.2 For any (accessible) DFA $D = (Q, \Sigma, \delta, q_0, F)$ accepting the regular language $L = L(D)$, the function $\varphi\colon \Sigma^* \to Q$ defined such that

$$\varphi(u) = \delta^*(q_0, u)$$

induces a bijection $\widehat{\varphi}\colon \Sigma^*/\rho_L \to Q/\equiv$, defined such that

$$\widehat{\varphi}([u]_{\rho_L}) = [\delta(q_0,u)]_{\equiv}.$$

Furthermore, we have

$$[u]_{\rho_L} a \subseteq [v]_{\rho_L} \quad \text{iff} \quad \delta(\varphi(u), a) \equiv \varphi(v).$$

Proof. Since $\varphi(u) = \delta^*(q_0,u)$ and $\varphi(v) = \delta^*(q_0,v)$, the fact that $\varphi(u) \equiv \varphi(v)$ can be expressed as

$$\begin{aligned} \forall w \in \Sigma^*(\delta^*(\delta^*(q_0,u),w) \in F \quad \text{iff} \quad \delta^*(\delta^*(q_0,v),w) \in F), \qquad \text{iff} \\ \forall w \in \Sigma^*(\delta^*(q_0,uw) \in F \quad \text{iff} \quad \delta^*(q_0,vw) \in F), \end{aligned}$$

which is exactly $u \rho_L v$. Thus, since every state in Q is accessible, we have a bijection $\widehat{\varphi}\colon \Sigma^*/\rho_L \to Q/\equiv$.

Since $\varphi(u) = \delta^*(q_0,u)$, we have

$$\delta(\varphi(u),a) = \delta(\delta^*(q_0,u),a) = \delta^*(q_0,ua) = \varphi(ua),$$

and thus, $\delta(\varphi(u), a) \equiv \varphi(v)$ can be expressed as $\varphi(ua) \equiv \varphi(v)$. By the previous part, this is equivalent to $ua\rho_L v$, which is equivalent to

$$[u]_{\rho_L} a \subseteq [v]_{\rho_L}.$$

□

Lemma 1.14.2 shows that the DFA D_{ρ_L} is isomorphic to the DFA $D/\equiv$ obtained as the quotient of the DFA D modulo the equivalence relation $\equiv$ on Q. More precisely, if

$$D/\equiv = (Q/\equiv, \Sigma, \delta/\equiv, [q_0]_\equiv, F/\equiv),$$

where

$$\delta/\equiv ([p]_\equiv, a) = [\delta(p, a)]_\equiv,$$

then $D/\equiv$ is isomorphic to the minimal DFA D_{ρ_L} accepting L, and thus, it is a minimal DFA accepting L.

The minimal DFA $D/\equiv$ is obtained by merging the states in each block of the partition Π associated with $\equiv$, forming states corresponding to the blocks of Π, and drawing a transition on input a from a block C_i to a block C_j of Π iff there is a transition $q = \delta(p, a)$ from any state $p \in C_i$ to any state $q \in C_j$ on input a. The start state is the block containing q_0, and the final states are the blocks consisting of final states.

Note that if $F = \emptyset$, then $\equiv$ has a single block (Q), and if $F = Q$, then $\equiv$ has a single block (F). In the first case, the minimal DFA is the one state DFA rejecting all strings. In the second case, the minimal DFA is the one state DFA accepting all strings. When $F \neq \emptyset$ and $F \neq Q$, there are at least two states in Q, and $\equiv$ also has at least two blocks, as we shall see shortly. It remains to compute $\equiv$ explicitly. This is done using a sequence of approximations. In view of the previous discussion, we are assuming that $F \neq \emptyset$ and $F \neq Q$, which means that $n \geq 2$, where n is the number of states in Q.

Definition 1.14.3 Given any DFA $D = (Q, \Sigma, \delta, q_0, F)$, for every $i \geq 0$, the relation $\equiv_i$ on Q, called *i-state equivalence*, is defined as follows: for all $p, q \in Q$,

$$p \equiv_i q \quad \text{iff} \quad \forall w \in \Sigma^*,\ |w| \leq i\ (\delta^*(p, w) \in F \quad \text{iff} \quad \delta^*(q, w) \in F).$$

When $p \equiv_i q$, we say that *p and q are i-indistinguishable*.

Since we assumed that $F \neq \emptyset$ and $F \neq Q$, it is immediately verified that $\equiv_0$ has exactly two equivalence classes F and $Q - F$, and that

$$\equiv \subseteq \ldots \subseteq \equiv_{i+1} \subseteq \equiv_i \subseteq \ldots \subseteq \equiv_1 \subseteq \equiv_0 .$$

If this sequence was strictly decreasing, the partition associated with $\equiv_{i+1}$ would contain at least one more block than the partition associated with $\equiv_i$, and since $\equiv$ only has finitely many blocks (at most the number of states n in Q), there is a smallest integer, say i_0, such that

$$\equiv_{i_0+1} = \equiv_{i_0} .$$

Furthermore, we have $i_0 \leq n - 2$, where n is the number of states in Q. Thus, it remains to compute $\equiv_{i+1}$ from $\equiv_i$, which can be done using the following lemma. The lemma will also show that

$$\equiv = \equiv_{i_0} .$$

Lemma 1.14.4 *For any (accessible) DFA $D = (Q, \Sigma, \delta, q_0, F)$, for all $p, q \in Q$, $p \equiv_{i+1} q$ iff $p \equiv_i q$ and $\delta(p, a) \equiv_i \delta(q, a)$, for every $a \in \Sigma$. Furthermore, if $F \neq \emptyset$ and $F \neq Q$, there is a smallest integer $i_0 \leq n - 2$, such that*

$$\equiv_{i_0+1} = \equiv_{i_0} = \equiv .$$

We leave the easy proof as an exercise.

Using lemma 1.14.4, we can compute $\equiv$ inductively, starting from $\equiv_0 = (F, Q - F)$, and computing $\equiv_{i+1}$ from $\equiv_i$, until the sequence of partitions associated with the $\equiv_i$ stabilizes. There are a number of algorithms for computing $\equiv$, or to determine whether $p \equiv q$ for some given $p, q \in Q$.

A simple method to compute $\equiv$ is described in Hopcroft and Ullman. It consists in forming a triangular array corresponding to all unordered pairs (p, q), with $p \neq q$ (the rows and the columns of this triangular array are indexed by the states in Q, where the entries are below the descending diagonal). Initially, the entry (p, q) is marked iff p and q are **not** 0-equivalent, which means that p and q are not both in F. Then, we process every unmarked entry on every row as follows: for any unmarked pair (p, q), we consider of pairs $(\delta(p, a), \delta(q, a))$, for all $a \in \Sigma$. If any pair $(\delta(p, a), \delta(q, a))$ is already marked, this means that $\delta(p, a)$ and $\delta(q, a)$ are inequivalent, and thus p and q are inequivalent, and we mark the pair (p, q). We continue in this fashion, until at the end of a round during which all the rows are processed, nothing has changed. When the algorithm stops, all marked pairs are inequivalent, and all unmarked pairs correspond to equivalent states.

There are ways of improving the efficiency of this algorithm, see Hopcroft and Ullman for such improvements. Fast algorithms for testing whether $p \equiv q$ for some given $p, q \in Q$ also exist. One of these algorithms based on "forward closures" is discussed in the exercises. Such an algorithm is related to a fast unification algorithm.

Exercises

1 The purpose of this problem is to get a fast algorithm for testing state equivalence in a DFA. Let $D = (Q, \Sigma, \delta, q_0, F)$ be a deterministic finite automaton. Recall that *state equivalence* is the equivalence relation $\equiv$ on Q, defined such that,

$$p \equiv q \quad \text{iff} \quad \forall z \in \Sigma^*(\delta^*(p, z) \in F \quad \text{iff} \quad \delta^*(q, z) \in F).$$

and that *i-equivalence* is the equivalence relation $\equiv_i$ on Q, defined such that,

$$p \equiv_i q \quad \text{iff} \quad \forall z \in \Sigma^*,\ |z| \leq i\ (\delta^*(p, z) \in F \quad \text{iff} \quad \delta^*(q, z) \in F).$$

(i) Prove that $p \equiv_{i+1} q$ iff $p \equiv_i q$ and $\delta(p, a) \equiv_i \delta(q, a)$, for every $a \in \Sigma$. Prove that that $\equiv = \equiv_i$, for some $i \leq n - 2$, where n is the number of states in Q.

A relation $S \subseteq Q \times Q$ is a *forward closure* iff it is an equivalence relation and

(1) Whenever $(p, q) \in S$, then $(\delta(p, a), \delta(q, a)) \in S$, for all $a \in \Sigma$.

We say that a forward closure S is *good* iff whenever $(p, q) \in S$, then $good(p, q)$, where $good(p, q)$ holds iff either both $p, q \in F$, or both $p, q \notin F$.

Given any relation $R \subseteq Q \times Q$, recall that the smallest equivalence relation $R_\approx$ containing R is the relation $(R \cup R^{-1})^*$ (where $R^{-1} = \{(q, p) \mid (p, q) \in R\}$, and $(R \cup R^{-1})^*$ is the reflexive and transitive closure of $(R \cup R^{-1})$). We define the sequence of relations $S_i \subseteq Q \times Q$ as follows:

$$\begin{aligned} S_0 &= R_\approx \\ S_{i+1} &= (S_i \cup \{(\delta(p, a), \delta(q, a)) \mid (p, q) \in S_i,\ a \in \Sigma\})_\approx. \end{aligned}$$

(ii) Prove that $S_{i_0+1} = S_{i_0}$ for some least i_0. Prove that S_{i_0} is the smallest forward closure containing R.

(iii) Prove that $p \equiv q$ iff the forward closure of the relation $R = \{(p, q)\}$ is good.

Hint: First, show that $p \equiv q$ iff $good(p, q)$ and $\delta(p, a) \equiv \delta(q, a)$ for all $a \in \Sigma$.

2 A fast algorithm for computing the forward closure of the relation $R = \{(p, q)\}$, or detecting a bad pair of states, can be obtained as follows. An equivalence relation on Q is represented by a partition Π. Each equivalence class C in the partition is represented by a tree structure consisting of nodes and (parent) pointers, with the pointers from the sons of a node to the node itself. The root has a null pointer. Each node also maintains a counter keeping track of the number of nodes in the subtree rooted at that node.

Two functions *union* and $find$ are defined as follows. Given a state p, $find(p, \Pi)$ finds the root of the tree containing p as a node (not necessarily a leaf). Given two root nodes p, q, $union(p, q, \Pi)$ forms a new partition by merging the two trees with roots p and q as follows: if the counter of p is smaller than that of q, then let the root of p point to q, else let the root of q point to p.

In order to speed up the algorithm, we can modify $find$ as follows: during a call $find(p, \Pi)$, as we follow the path from p to the root r of the tree containing p, we redirect the parent pointer of every node q on the path from p (including p itself) to r.

Say that pair $\langle p, q\rangle$ is *bad* iff either both $p \in F$ and $q \notin F$, or both $p \notin F$ and $q \in F$. The function *bad* is such that $bad(\langle p, q\rangle) = true$ if $\langle p, q\rangle$ is bad, and $bad(\langle p, q\rangle) = false$ otherwise.

For details of this implementation of partitions, see *Fundamentals of data structures*, by Horowitz and Sahni, Computer Science press, pp. 248-256.

Then, the algorithm is as follows:

```
function unif[p, q, Π, dd]: flag;
  begin
    trans := left(dd); ff := right(dd); pq; = (p, q); st = (pq); flag = 1;
    k := Length(first(trans));
    while st ≠ () ∧ flag ≠ 0 do
      uv := top(st); uu := left(uv); vv := right(uv);
      pop(st);
      if bad(ff, uv) = 1 then flag := 0
      else
        u := find(uu, Π); v := find(vv, Π);
        if u ≠ v then
          union(u, v, Π);
          for i = 1 to k do
            u1 := delta(trans, uu, k − i + 1); v1 := delta(trans, vv, k − i + 1);
            uv = (u1, v1); push(st, uv)
          endfor
        endif
      endif
    endwhile
  end
```

The algorithm uses a stack *st*. We are assuming that the DFA *dd* is specified as a list of two sublists, the fist list being a representation of the transition function, and the second one of the set of final states. The transition function itself is a list of lists, where the i-th list represents the i-th row of the transition table for *dd*. The function *delta* is such that $delta(trans, i, j)$ returns the j-th state in the i-th row of the transition table of *dd*. For example, we have a DFA

$$dd = (((2,3),(2,4),(2,3),(2,5),(2,3),(7,6),(7,8),(7,9),(7,6)),(5,9)).$$

Implement the above algorithm, and test it at least for the above DFA dd and the pairs of states $(1,6)$ and $(1,7)$.

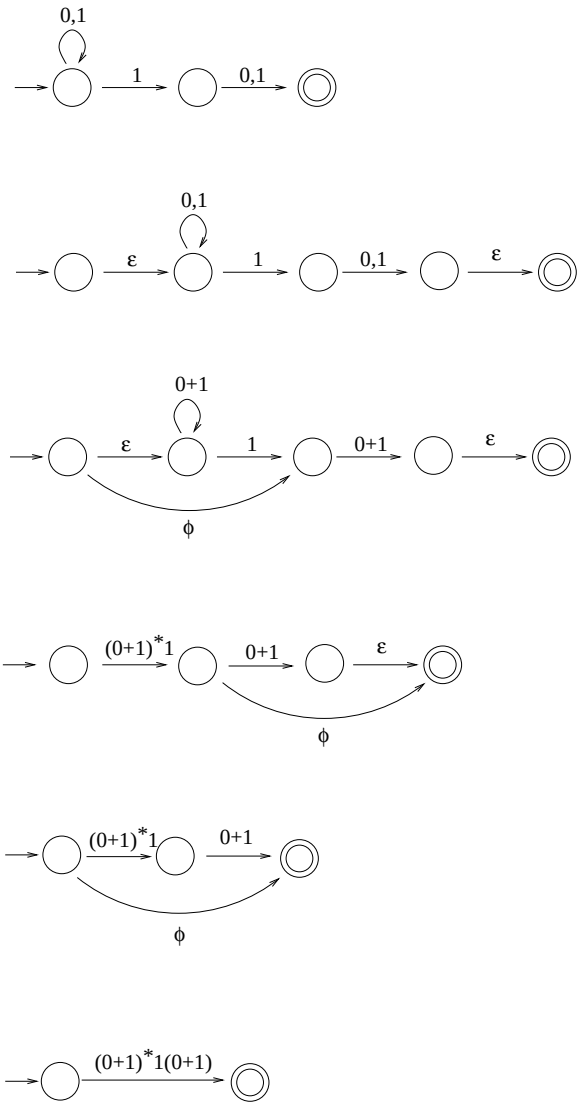
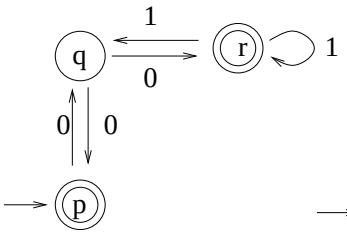
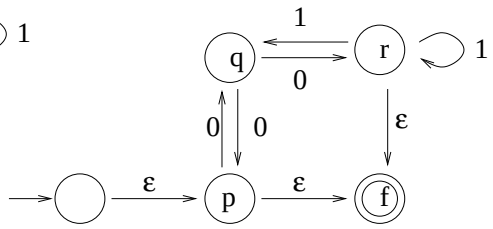


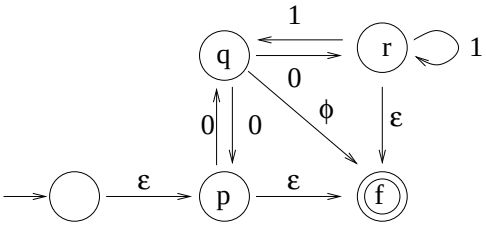
Figure 1.29: An example of node elimination. Keep in mind that if the original NFA lacks an edge between two states, then during the node elimination process we assume that there is an edge there, but it is labeled with the empty set. In most cases we don't want to draw this edge in, since the picture will look like a mess. In many cases these edges can be ignored, so we only draw it if it plays a non-trivial role in the elimination process.



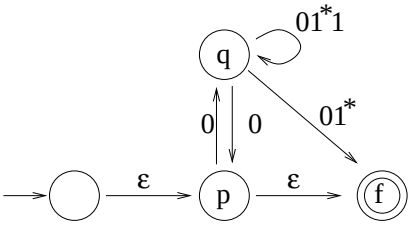
1) The given machine.



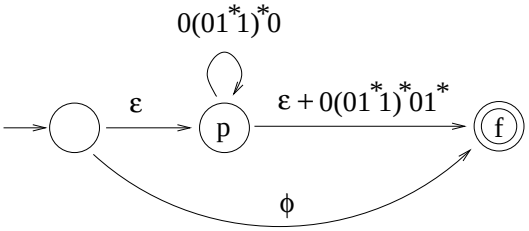
2) The machine preprocessed.



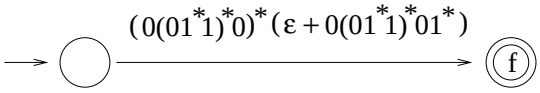
3) Prepare to eliminate the node r.
We must take into account all non-empty edges
going in and out of r, in other words, elimanatie with.
respect to not only the pair (q,f) but the pair (q,q).



4) r has been elimated. We next eliminate q. This must
be done with respect to the pairs (p,p) and (p,f).



5) q has now been eliminated and we prepare the eliminate p.



6) The final machine with the desired regular expression.

Figure 1.30: Another example of node elimination. This example is trickier than the prior one, since some of the nodes have many edges coming and going into them, so several triangles must be simultaneously collapsed.

Chapter 2

Formal Languages

2.1 A Grammar for Parsing English

The study of grammar in computer science is, of course, related to the study of the grammar of languages, and so in some sense it's development overlaps with the development of linguistics, a truly vast subject.
In the "primordial soup" of grammar were the parts of speech.
Once one realizes that that parts of sentences could be labeled to indicate their function, it seems natural that one might notice that there appear to be rules dictating how these parts of speech are used. Also, if they are used incorrectly in a sentence, a listening native speaker will say that the sentence sounds "wrong". As a result of having the notion of the parts of speech, it becomes possible to make a statement grammar. For example,

In English, every sentence contains a verb.
or
A proper sentence of Martian consists of 100 nouns followed by 200 verbs and a participle.

After discovering the parts of speech the next step is to start diagramming, i.e. parsing sentences. The formalization of this process led to the formal notion of a grammar in the 1950's, a process which culminated in Noam Chomsky's classic book *Syntactic structures*. Chomsky is responsible for the definition of a context free grammar, which is a mathematical way of saying how to generate the sentences of a language. A collection of rules are given, often starting in the following way:

$$S \longrightarrow NP + VP$$

NP stands for *noun phrase* and VP stands for verb phase. The idea is that simple sentences consists of two pieces, a noun phrase followed by a verb phrase. But this appears to not be very useful, since we don't know what noun and verb phrases are, and so how does this help to understand English grammar ? We answer this by showing what a noun phrase may be decomposed into:

$$NP \longrightarrow proper\ noun$$

or

$$NP \longrightarrow indefinite\ pronoun$$

or

$$NP \longrightarrow determiner + common\ noun$$

This looks worse! It seems that more and more strange, undefined words are being introduced. Fortunately, these rules eventually terminate in 'recognizable' words. But before we give the full set of rules that demonstrate this, some discussion about the first rule might be helpful.

A noun phrase is what is sounds like, namely, the generalization of what a noun is, and everyone knows that a noun is a person, place or thing. Now admittedly, this last definition is a little shaky, since it is hard to imagine an example of something that is not a 'thing' ! Nevertheless, we ask the reader to bear with us. We would like to consider strings of words (phrases) like

The child
or
Fifteen hundred poison jelly beans
or
Those first three elements
or
Five out of every six planets
or
Somebody
or
The top of the icosahedron

to be considered as nouns, but they have all those funny modifiers in front of what we would normally call the nouns, namely child and jelly beans. So we make up the more general category, the noun phrase. Below we give a collection of rules for generating noun phrases, which are one-half of the rules for simplex sentences, sometimes called minimal clauses. A minimal clause is a noun phrase followed by a verb phrase. But we must emphasize here that these rules will sometimes generate phrases that are not standard American! In fact, this is in some sense the entire problem of syntax in linguistics, i.e. finding a system that describes exactly what the structure of a language is. The authors know of no way of doing this with a system like that described below, which is an example of a context free grammar. So we have to settle for something that will in fact generate many of the sentences that we consider to be standard American, but it will also generate a few more. Nevertheless, such a grammar is still useful in parsing sentences, i.e. given a sentence it can be decomposed into pieces. But it is not even clear that all sentences can be parsed. So we settle for an approximation here, since our point is not to solve the problems of linguistics, but just to motivate our later study of formal languages.

s ⟶ np + vp

np ⟶ proper noun | indefinite pronoun | determiner + common noun
proper-noun ⟶ Δ
indefinite pronoun ⟶ {some-, no-, every-, any-} {-body, -one, -thing}
determiner ⟶ (pre-article) article (locative)
pre-article ⟶ quantifier (out) of (every)
quantifier ⟶ {number } {(a) few, some, much, many, more, most, all}
article ⟶ {definite, non-definite }
definite ⟶ {the, NEAR, FAR }
NEAR ⟶ {this, these }
FAR ⟶ {that, those }
non-definite ⟶ {some (or other), {a, Ø} (certain) }
locative ⟶ [status number dimension]
status ⟶ {past, present, future }
number ⟶ {cardinal, ordinal }
cardinal ⟶ {one, two, three,... }
ordinal ⟶ {first, second, third,..., second from the end, next to the last, last }
dimension ⟶ {top, bottom, right, left, front, back, middle }
common-noun ⟶ {count, non-count }
count ⟶ Δ (plural)
non-count ⟶ Δ

vp ⟶ auxiliary {equational, nonequational (manner) } (adver(ial(s)))
auxiliary ⟶ tense (modal) (participle)
tense ⟶ {pres, past}
modal ⟶ {can, may, shall, will, must, (have to), be to, come to, ought to, do (to)? }
participle ⟶ [perfect/ progressive]
perfect ⟶ have -en/d
progressive ⟶ be -ing
equational ⟶ {compula complement, attributive np(q) }
copula ⟶ {be, become, remain }
complement ⟶ {({preposition, like}) np, adj }
preposition ⟶ {in, out, under,over, through, across, from, to(?), towards, away(from),...}
adj ⟶ (intensifier) Δ
intensifier ⟶ {somewhat, rather, quite, very, awfully, terribly, damned }
attributive ⟶ {have, cost, weigh, measure}
npq ⟶ {two dollars, three square feet, seventeen pounds, an awful lot }
non-equational ⟶ {VI, VT np }
VI ⟶ Δ [participle/complement]
VT ⟶ Δ [participle/complement]
manner ⟶ adj -ly
adverb(ial(s)) ⟶ [place time duration frequency]
place ⟶ preposition np
time ⟶ {yesterday, today, tomorrow, then, now, in the future, three minutes ago, four years ago,... }
duration ⟶ { (for) two hours, (for) three years,...}
frequency ⟶ {twice, three times, three times a minute, often, seldom, never,...}

2.2 Context-Free Grammars

A context-free grammar basically consists of a finite set of grammar rules. In order to define grammar rules, we assume that we have two kinds of symbols: the terminals, which are the symbols of the alphabet underlying the languages under consideration, and the nonterminals, which behave like variables ranging over strings of terminals. A rule is of the form $A \to \alpha$, where A is a single nonterminal, and the right-hand side α is a string of terminal and/or

nonterminal symbols. As usual, first we need to define what the object is (a context-free grammar), and then we need to explain how it is used. Unlike automata, grammars are used to *generate* strings, rather than recognize strings.

Definition 2.2.1 A *context-free grammar (for short, CFG)* is a quadruple $G = (V, \Sigma, P, S)$, where

- V is a finite set of symbols called the *vocabulary (or set of grammar symbols)*;
- $\Sigma \subseteq V$ is the set of *terminal symbols (for short, terminals)*;
- $S \in (V - \Sigma)$ is a designated symbol called the *start symbol*;
- $P \subseteq (V - \Sigma) \times V^*$ is a finite set of *productions (or rewrite rules, or rules)*.

The set $N = V - \Sigma$ is called the set of *nonterminal symbols (for short, nonterminals)*. Thus, $P \subseteq N \times V^*$, and every production $\langle A, \alpha \rangle$ is also denoted as $A \to \alpha$. A production of the form $A \to \epsilon$ is called an *epsilon rule, or null rule*.

Remark: Context-free grammars are sometimes defined as $G = (V_N, V_T, P, S)$. The correspondence with our definition is that $\Sigma = V_T$ and $N = V_N$, so that $V = V_N \cup V_T$. Thus, in this other definition, it is necessary to assume that $V_T \cap V_N = \emptyset$.

Example 1. $G_1 = (\{E, a, b\}, \{a, b\}, E, P)$, where P is the set of rules

$$E \longrightarrow aEb,$$
$$E \longrightarrow ab.$$

As we will see shortly, this grammar generates the language $L_1 = \{a^n b^n \mid n \geq 1\}$, which is not regular.

Example 2. $G_2 = (\{E, +, *, (,), a\}, \{+, *, (,), a\}, E, P)$, where P is the set of rules

$$E \longrightarrow E + E,$$
$$E \longrightarrow E * E,$$
$$E \longrightarrow (E),$$
$$E \longrightarrow a.$$

This grammar generates a set of arithmetic expressions.

2.3 Derivations and Context-Free Languages

The productions of a grammar are used to derive strings. In this process, the productions are used as rewrite rules. Formally, we define the derivation relation associated with a context-free grammar.

Definition 2.3.1 Given a context-free grammar $G = (V, \Sigma, P, S)$, the (one-step) *derivation relation* $\Longrightarrow_G$ *associated with* G is the binary relation $\Longrightarrow_G \subseteq V^* \times V^*$ defined as follows: for all $\alpha, \beta \in V^*$, we have

$$\alpha \Longrightarrow_G \beta$$

iff there exist $\lambda, \rho \in V^*$, and some production $(A \to \gamma) \in P$, such that

$$\alpha = \lambda A \rho \quad \text{and} \quad \beta = \lambda \gamma \rho.$$

The transitive closure of $\Longrightarrow_G$ is denoted as $\overset{+}{\Longrightarrow}_G$ and the reflexive and transitive closure of $\Longrightarrow_G$ is denoted as $\overset{*}{\Longrightarrow}_G$.

When the grammar G is clear from the context, we usually omit the subscript G in $\Longrightarrow_G$, $\overset{+}{\Longrightarrow}_G$, and $\overset{*}{\Longrightarrow}_G$.

A string $\alpha \in V^*$ such that $S \overset{*}{\Longrightarrow} \alpha$ is called a *sentential form*, and a string $w \in \Sigma^*$ such that $S \overset{*}{\Longrightarrow} w$ is called a *sentence*. A derivation $\alpha \overset{*}{\Longrightarrow} \beta$ involving n steps is denoted as $\alpha \overset{n}{\Longrightarrow} \beta$.

Note that a derivation step

$$\alpha \Longrightarrow_G \beta$$

is rather nondeterministic. Indeed, one can chose among various occurrences of nonterminals A in α, and also among various productions $A \to \gamma$ with left-hand side A.

For example, using the grammar $G_1 = (\{E, a, b\}, \{a, b\}, E, P)$, where P is the set of rules

$$E \longrightarrow aEb,$$
$$E \longrightarrow ab,$$

every derivation from E is of the form

$$E \overset{*}{\Longrightarrow} a^n E b^n \Longrightarrow a^n abb^n = a^{n+1}b^{n+1},$$

or

$$E \overset{*}{\Longrightarrow} a^n E b^n \Longrightarrow a^n aEbb^n = a^{n+1}Eb^{n+1},$$

where $n \geq 0$.

Grammar G_1 is very simple: every string $a^n b^n$ has a unique derivation. This is usually not the case. For example, using the grammar $G_2 = (\{E, +, *, (,), a\}, \{+, *, (,), a\}, E, P)$, where P is the set of rules

$$E \longrightarrow E + E,$$
$$E \longrightarrow E * E,$$
$$E \longrightarrow (E),$$
$$E \longrightarrow a,$$

the string $a + a * a$ has the following distinct derivations, where the boldface indicates which occurrence of E is rewritten:

$$\mathbf{E} \Longrightarrow \mathbf{E} * E \Longrightarrow \mathbf{E} + E * E$$
$$\Longrightarrow a + \mathbf{E} * E \Longrightarrow a + a * \mathbf{E} \Longrightarrow a + a * a,$$

and

$$\mathbf{E} \Longrightarrow \mathbf{E} + E \Longrightarrow a + \mathbf{E}$$
$$\Longrightarrow a + \mathbf{E} * E \Longrightarrow a + a * \mathbf{E} \Longrightarrow a + a * a.$$

In the above derivations, the leftmost occurrence of a nonterminal is chosen at each step. Such derivations are called *leftmost derivations*. We could systematically rewrite the rightmost occurrence of a nonterminal, getting *rightmost derivations*. The string $a + a * a$ also has the following two rightmost derivations, where the boldface indicates which occurrence of E is rewritten:

$$\mathbf{E} \Longrightarrow E + \mathbf{E} \Longrightarrow E + E * \mathbf{E}$$
$$\Longrightarrow E + \mathbf{E} * a \Longrightarrow \mathbf{E} + a * a \Longrightarrow a + a * a,$$

and

$$\mathbf{E} \Longrightarrow E * \mathbf{E} \Longrightarrow \mathbf{E} * a$$
$$\Longrightarrow E + \mathbf{E} * a \Longrightarrow \mathbf{E} + a * a \Longrightarrow a + a * a.$$

The language generated by a context-free grammar is defined as follows.

Definition 2.3.2 Given a context-free grammar $G = (V, \Sigma, P, S)$, the *language generated by* G is the set

$$L(G) = \{w \in \Sigma^* \mid S \overset{+}{\Longrightarrow} w\}.$$

A language $L \subseteq \Sigma^*$ is a *context-free language (for short, CFL)* iff $L = L(G)$ for some context-free grammar G.

It is technically very useful to consider derivations in which the leftmost nonterminal is always selected for rewriting, and dually, derivations in which the rightmost nonterminal is always selected for rewriting.

Definition 2.3.3 Given a context-free grammar $G = (V, \Sigma, P, S)$, the (one-step) *leftmost derivation relation* $\underset{lm}{\Longrightarrow}$ *associated with* G is the binary relation $\underset{lm}{\Longrightarrow} \subseteq V^* \times V^*$ defined as follows: for all $\alpha, \beta \in V^*$, we have

$$\alpha \underset{lm}{\Longrightarrow} \beta$$

iff there exist $u \in \Sigma^*$, $\rho \in V^*$, and some production $(A \to \gamma) \in P$, such that

$$\alpha = uA\rho \quad \text{and} \quad \beta = u\gamma\rho.$$

The transitive closure of $\underset{lm}{\Longrightarrow}$ is denoted as $\overset{+}{\underset{lm}{\Longrightarrow}}$ and the reflexive and transitive closure of $\underset{lm}{\Longrightarrow}$ is denoted as $\overset{*}{\underset{lm}{\Longrightarrow}}$. The (one-step) *rightmost derivation relation* $\underset{rm}{\Longrightarrow}$ *associated with* G is the binary relation $\underset{rm}{\Longrightarrow} \subseteq V^* \times V^*$ defined as follows: for all $\alpha, \beta \in V^*$, we have

$$\alpha \underset{rm}{\Longrightarrow} \beta$$

iff there exist $\lambda \in V^*$, $v \in \Sigma^*$, and some production $(A \to \gamma) \in P$, such that

$$\alpha = \lambda Av \quad \text{and} \quad \beta = \lambda\gamma v.$$

The transitive closure of $\underset{rm}{\Longrightarrow}$ is denoted as $\overset{+}{\underset{rm}{\Longrightarrow}}$ and the reflexive and transitive closure of $\underset{rm}{\Longrightarrow}$ is denoted as $\overset{*}{\underset{rm}{\Longrightarrow}}$.

Remarks: It is customary to use the symbols a, b, c, d, e for terminal symbols, and the symbols A, B, C, D, E for nonterminal symbols. The symbols u, v, w, x, y, z denote terminal strings, and the symbols $\alpha, \beta, \gamma, \lambda, \rho, \mu$ denote strings in V^*. The symbols X, Y, Z usually denote symbols in V.

Given a context-free grammar $G = (V, \Sigma, P, S)$, *parsing a string* w consists in finding out whether $w \in L(G)$, and if so, in producing a derivation for w. The following lemma is technically very important. It shows that leftmost and rightmost derivations are "universal". This has some important practical implications for the complexity of parsing algorithms.

Lemma 2.3.4 *Let $G = (V, \Sigma, P, S)$ be a context-free grammar. For every $w \in \Sigma^*$, for every derivation $S \overset{+}{\Longrightarrow} w$, there is a leftmost derivation $S \overset{+}{\underset{lm}{\Longrightarrow}} w$, and there is a rightmost derivation $S \overset{+}{\underset{rm}{\Longrightarrow}} w$.*

Proof. Of course, we have to somehow use induction on derivations, but this is a little tricky, and it is necessary to prove a stronger fact. We treat leftmost derivations, rightmost derivations being handled in a similar way.

Claim: For every $w \in \Sigma^*$, for every $\alpha \in V^+$, if $\alpha \overset{n}{\Longrightarrow} w$, then there is a leftmost derivation $\alpha \overset{n}{\underset{lm}{\Longrightarrow}} w$.

The claim is proved by induction on n.

For $n = 1$, there exist some $\lambda, \rho \in V^*$ and some production $A \to \gamma$, such that $\alpha = \lambda A \rho$ and $w = \lambda \gamma \rho$. Since w is a terminal string, λ, ρ, and γ, are terminal strings. Thus, A is the only nonterminal in α, and the derivation step $\alpha \overset{1}{\Longrightarrow} w$ is a leftmost step (and a rightmost step!).

If $n > 1$, then the derivation $\alpha \overset{n}{\Longrightarrow} w$ is of the form

$$\alpha \Longrightarrow \alpha_1 \overset{n-1}{\Longrightarrow} w.$$

There are two sub-cases.

Case 1. If the derivation step $\alpha \Longrightarrow \alpha_1$ is a leftmost step $\alpha \underset{lm}{\Longrightarrow} \alpha_1$, by the induction hypothesis, there is a leftmost derivation $\alpha_1 \overset{n-1}{\underset{lm}{\Longrightarrow}} w$, and we get the leftmost derivation

$$\alpha \underset{lm}{\Longrightarrow} \alpha_1 \overset{n-1}{\underset{lm}{\Longrightarrow}} w.$$

Case 2. The derivation step $\alpha \Longrightarrow \alpha_1$ is a not a leftmost step. In this case, there must be some $u \in \Sigma^*$, $\mu, \rho \in V^*$, some nonterminals A and B, and some production $B \to \delta$, such that

$$\alpha = uA\mu B\rho \quad \text{and} \quad \alpha_1 = uA\mu\delta\rho,$$

where A is the leftmost nonterminal in α. Since we have a derivation $\alpha_1 \overset{n-1}{\Longrightarrow} w$ of length $n - 1$, by the induction hypothesis, there is a leftmost derivation

$$\alpha_1 \overset{n-1}{\underset{lm}{\Longrightarrow}} w.$$

Since $\alpha_1 = uA\mu\delta\rho$ where A is the leftmost terminal in α_1, the first step in the leftmost derivation $\alpha_1 \overset{n-1}{\underset{lm}{\Longrightarrow}} w$ is of the form

$$uA\mu\delta\rho \underset{lm}{\Longrightarrow} u\gamma\mu\delta\rho,$$

for some production $A \to \gamma$. Thus, we have a derivation of the form

$$\alpha = uA\mu B\rho \Longrightarrow uA\mu\delta\rho \underset{lm}{\Longrightarrow} u\gamma\mu\delta\rho \overset{n-2}{\underset{lm}{\Longrightarrow}} w.$$

We can commute the first two steps involving the productions $B \to \delta$ and $A \to \gamma$, and we get the derivation

$$\alpha = uA\mu B\rho \underset{lm}{\Longrightarrow} u\gamma\mu B\rho \Longrightarrow u\gamma\mu\delta\rho \overset{n-2}{\underset{lm}{\Longrightarrow}} w.$$

This may no longer be a leftmost derivation, but the first step is leftmost, and we are back in case 1. Thus, we conclude by applying the induction hypothesis to the derivation $u\gamma\mu B\rho \overset{n-1}{\Longrightarrow} w$, as in case 1. □

Lemma 2.3.4 implies that

$$L(G) = \{w \in \Sigma^* \mid S \overset{+}{\underset{lm}{\Longrightarrow}} w\} = \{w \in \Sigma^* \mid S \overset{+}{\underset{rm}{\Longrightarrow}} w\}.$$

We observed that if we consider the grammar $G_2 = (\{E, +, *, (,), a\}, \{+, *, (,), a\}, E, P)$, where P is the set of rules

$$\begin{aligned} E &\longrightarrow E + E, \\ E &\longrightarrow E * E, \\ E &\longrightarrow (E), \\ E &\longrightarrow a, \end{aligned}$$

the string $a + a * a$ has the following two distinct leftmost derivations, where the boldface indicates which occurrence of E is rewritten:

$$\mathbf{E} \Longrightarrow \mathbf{E} * E \Longrightarrow \mathbf{E} + E * E$$
$$\Longrightarrow a + \mathbf{E} * E \Longrightarrow a + a * \mathbf{E} \Longrightarrow a + a * a,$$

and

$$\mathbf{E} \Longrightarrow \mathbf{E} + E \Longrightarrow a + \mathbf{E}$$
$$\Longrightarrow a + \mathbf{E} * E \Longrightarrow a + a * \mathbf{E} \Longrightarrow a + a * a.$$

When this happens, we say that we have an ambiguous grammars. In some cases, it is possible to modify a grammar to make it unambiguous. For example, the grammar G_2 can be modified as follows.

Let $G_3 = (\{E, T, F, +, *, (,), a\}, \{+, *, (,), a\}, E, P)$, where P is the set of rules

$$E \longrightarrow E + T,$$
$$E \longrightarrow T,$$
$$T \longrightarrow T * F,$$
$$T \longrightarrow F,$$
$$F \longrightarrow (E),$$
$$F \longrightarrow a.$$

We leave as an exercise to show that $L(G_3) = L(G_2)$, and that every string in $L(G_3)$ has a unique derivation. Unfortunately, it is not always possible to modify a context-free grammar to make it unambiguous. There exist context-free languages that have no unambiguous context-free grammars. For example, it can be shown that

$$L_3 = \{a^m b^m c^n \mid m, n \geq 1\} \cup \{a^m b^n c^n \mid m, n \geq 1\}$$

is context-free, but has no unambiguous grammars. All this motivates the following definition.

Definition 2.3.5 A context-free grammar $G = (V, \Sigma, P, S)$ is *ambiguous* if there is some string $w \in L(G)$ that has two distinct leftmost derivations (or two distinct rightmost derivations). Thus, a grammar G is *unambiguous* if every string $w \in L(G)$ has a unique leftmost derivation (or a unique rightmost derivation). A context-free language L is *inherently ambiguous* if every CFG G for L is ambiguous.

Whether or not a grammar is ambiguous affects the complexity of parsing. Parsing algorithms for unambiguous grammars are more efficient than parsing algorithms for ambiguous grammars.

We now consider various normal forms for context-free grammars.

2.4 Normal Forms for Context-Free Grammars, Chomsky Normal Form

One of the main goals of this section is to show that every CFG G can be converted to an equivalent grammar in *Chomsky Normal Form (for short, CNF)*. A context-free grammar $G = (V, \Sigma, P, S)$ is in Chomsky Normal Form iff its productions are of the form

$$A \to BC,$$
$$A \to a, \quad \text{or}$$
$$S \to \epsilon,$$

where $A, B, C \in N$, $a \in \Sigma$, $S \to \epsilon$ is in G iff $\epsilon \in L(G)$, and S does not occur on the right-hand side of any production.

Note that a grammar in Chomsky Normal Form does not have ϵ-rules, i.e., rules of the form $A \to \epsilon$, except when $\epsilon \in L(G)$, in which case $S \to \epsilon$ is the only ϵ-rule. It also does not have *chain rules*, i.e., rules of the form $A \to B$, where $A, B \in N$. Thus, in order to convert a grammar to Chomsky Normal Form, we need to show how to eliminate ϵ-rules and chain rules. This is not the end of the story, since we may still have rules of the form $A \to \alpha$ where either $|\alpha| \geq 3$ or $|\alpha| \geq 2$ and α contains terminals. However, dealing with such rules is a simple recoding matter, and we first focus on the elimination of ϵ-rules and chain rules. It turns out that ϵ-rules must be eliminated first.

The first step to eliminate ϵ-rules is to compute the set $E(G)$ of *erasable (or nullable) nonterminals*

$$E(G) = \{A \in N \mid A \stackrel{+}{\Longrightarrow} \epsilon\}.$$

The set $E(G)$ is computed using a sequence of approximations E_i defined as follows:

$$\begin{aligned} E_0 &= \{A \in N \mid (A \to \epsilon) \in P\}, \\ E_{i+1} &= E_i \cup \{A \mid \exists (A \to B_1 \dots B_j \dots B_k) \in P,\ B_j \in E_i,\ 1 \leq j \leq k\}. \end{aligned}$$

Clearly, the E_i form an ascending chain

$$E_0 \subseteq E_1 \subseteq \cdots \subseteq E_i \subseteq E_{i+1} \subseteq \cdots \subseteq N,$$

and since N is finite, there is a least i, say i_0, such that $E_{i_0} = E_{i_0+1}$. We claim that $E(G) = E_{i_0}$. Actually, we prove the following lemma.

Lemma 2.4.1 *Given any context-free grammar $G = (V, \Sigma, P, S)$, one can construct a context-free grammar $G' = (V', \Sigma, P', S')$ such that:*

(1) $L(G') = L(G)$;

(2) P' contains no ϵ-rules other than $S' \to \epsilon$, and $S' \to \epsilon \in P'$ iff $\epsilon \in L(G)$;

(3) S' does not occur on the right-hand side of any production in P'.

Proof. We begin by proving that $E(G) = E_{i_0}$. For this, we prove that $E(G) \subseteq E_{i_0}$ and $E_{i_0} \subseteq E(G)$.

To prove that $E_{i_0} \subseteq E(G)$, we proceed by induction on i. Since $E_0 = \{A \in N \mid (A \to \epsilon) \in P\}$, we have $A \stackrel{1}{\Longrightarrow} \epsilon$, and thus $A \in E(G)$. By the induction hypothesis, $E_i \subseteq E(G)$. If $A \in E_{i+1}$, either $A \in E_i$ and then $A \in E(G)$, or there is some production $(A \to B_1 \dots B_j \dots B_k) \in P$, such that $B_j \in E_i$ for all j, $1 \leq j \leq k$. By the induction hypothesis, $B_j \stackrel{+}{\Longrightarrow} \epsilon$ for each j, $1 \leq j \leq k$, and thus

$$A \Longrightarrow B_1 \dots B_j \dots B_k \stackrel{+}{\Longrightarrow} B_2 \dots B_j \dots B_k \stackrel{+}{\Longrightarrow} B_j \dots B_k \stackrel{+}{\Longrightarrow} \epsilon,$$

which shows that $A \in E(G)$.

To prove that $E(G) \subseteq E_{i_0}$, we also proceed by induction, but on the length of a derivation $A \stackrel{+}{\Longrightarrow} \epsilon$. If $A \stackrel{1}{\Longrightarrow} \epsilon$, then $A \to \epsilon \in P$, and thus $A \in E_0$ since $E_0 = \{A \in N \mid (A \to \epsilon) \in P\}$. If $A \stackrel{n+1}{\Longrightarrow} \epsilon$, then

$$A \Longrightarrow \alpha \stackrel{n}{\Longrightarrow} \epsilon,$$

for some production $A \to \alpha \in P$. If α contains terminals of nonterminals not in $E(G)$, it is impossible to derive ϵ from α, and thus, we must have $\alpha = B_1 \dots B_j \dots B_k$, with $B_j \in E(G)$, for all j, $1 \leq j \leq k$. However,

$B_j \overset{n_j}{\Longrightarrow} \epsilon$ where $n_j \leq n$, and by the induction hypothesis, $B_j \in E_{i_0}$. But then, we get $A \in E_{i_0+1} = E_{i_0}$, as desired. □

Having shown that $E(G) = E_{i_0}$, we construct the grammar G'. Its set of production P' is defined as follows. First, we create the production $S' \to S$ where $S' \notin V$, to make sure that S' does not occur on the right-hand side of any rule in P'. Let

$$P_1 = \{A \to \alpha \mid \alpha \in V^+\} \cup \{S' \to S\},$$

and let P_2 be the set of productions

$$P_2 = \{A \to \alpha_1\alpha_2 \dots \alpha_k\alpha_{k+1} \mid \exists \alpha_1 \in V^*, \dots, \exists \alpha_{k+1} \in V^*, \exists B_1 \in E(G), \dots, \exists B_k \in E(G)$$
$$A \to \alpha_1 B_1 \alpha_2 \dots \alpha_k B_k \alpha_{k+1} \in P,\ k \geq 1,\ \alpha_1 \dots \alpha_{k+1} \neq \epsilon\}.$$

Note that $\epsilon \in L(G)$ iff $S \in E(G)$. If $S \notin E(G)$, then let $P' = P_1 \cup P_2$, and if $S \in E(G)$, then let $P' = P_1 \cup P_2 \cup \{S' \to \epsilon\}$. We claim that $L(G') = L(G)$, which is proved by showing that every derivation using G can be simulated by a derivation using G', and vice-versa. All the conditions of the lemma are now met. □

¿From a practical point of view, the construction or lemma 2.4.1 is very costly. For example, given a grammar containing the productions

$$\begin{aligned} S &\to ABCDEF, \\ A &\to \epsilon, \\ B &\to \epsilon, \\ C &\to \epsilon, \\ D &\to \epsilon, \\ E &\to \epsilon, \\ F &\to \epsilon, \\ \dots &\to \dots, \end{aligned}$$

eliminating ϵ-rules will create $2^6 - 1 = 63$ new rules corresponding to the 63 nonempty subsets of the set $\{A, B, C, D, E, F\}$. We now turn to the elimination of chain rules.

It turns out that matters are greatly simplified if we first apply lemma 2.4.1 to the input grammar G, and we explain the construction assuming that $G = (V, \Sigma, P, S)$ satisfies the conditions of lemma 2.4.1. For every nonterminal $A \in N$, we define the set

$$I_A = \{B \in N \mid A \overset{+}{\Longrightarrow} B\}.$$

The sets I_A are computed using approximations $I_{A,i}$ defined as follows:

$$\begin{aligned} I_{A,0} &= \{B \in N \mid (A \to B) \in P\}, \\ I_{A,i+1} &= I_{A,i} \cup \{C \in N \mid \exists (B \to C) \in P, \text{ and } B \in I_{A,i}\}. \end{aligned}$$

Clearly, for every $A \in N$, the $I_{A,i}$ form an ascending chain

$$I_{A,0} \subseteq I_{A,1} \subseteq \cdots \subseteq I_{A,i} \subseteq I_{A,i+1} \subseteq \cdots \subseteq N,$$

and since N is finite, there is a least i, say i_0, such that $I_{A,i_0} = I_{A,i_0+1}$. We claim that $I_A = I_{A,i_0}$. Actually, we prove the following lemma.

Lemma 2.4.2 *Given any context-free grammar $G = (V, \Sigma, P, S)$, one can construct a context-free grammar $G' = (V', \Sigma, P', S')$ such that:*

(1) $L(G') = L(G)$*;*

(2) Every rule in P' *is of the form* $A \to \alpha$ *where* $|\alpha| \geq 2$*, or* $A \to a$ *where* $a \in \Sigma$*, or* $S' \to \epsilon$ *iff* $\epsilon \in L(G)$*;*

(3) S' *does not occur on the right-hand side of any production in* P'*.*

Proof. First, we apply lemma 2.4.1 to the grammar G, obtaining a grammar $G_1 = (V_1, \Sigma, S_1, P_1)$. The proof that $I_A = I_{A,i_0}$ is similar to the proof that $E(G) = E_{i_0}$. First, we prove that $I_{A,i} \subseteq I_A$ by induction on i. This is straightforward. Next, we prove that $I_A \subseteq I_{A,i_0}$ by induction on derivations of the form $A \stackrel{+}{\Longrightarrow} B$. In this part of the proof, we use the fact that G_1 has no ϵ-rules except perhaps $S_1 \to \epsilon$, and that S_1 does not occur on the right-hand side of any rule. This implies that a derivation $A \stackrel{n+1}{\Longrightarrow} C$ is necessarily of the form $A \stackrel{n}{\Longrightarrow} B \Longrightarrow C$ for some $B \in N$. Then, in the induction step, we have $B \in I_{A,i_0}$, and thus $C \in I_{A,i_0+1} = I_{A,i_0}$.

We now define the following sets of rules. Let

$$P_2 = P_1 - \{A \to B \mid A \to B \in P_1\},$$

and let

$$P_3 = \{A \to \alpha \mid B \to \alpha \in P_1,\ \alpha \notin N_1,\ B \in I_A\}.$$

We claim that $G' = (V_1, \Sigma, P_2 \cup P_3, S_1)$ satisfies the conditions of the lemma. For example, S_1 does not appear on the right-hand side of any production, since the productions in P_3 have right-hand sides from P_1, and S_1 does not appear on the right-hand side in P_1. It is also easily shown that $L(G') = L(G_1) = L(G)$. □

Let us apply the method of lemma 2.4.2 to the grammar

$$G_3 = (\{E, T, F, +, *, (,), a\}, \{+, *, (,), a\}, E, P),$$

where P is the set of rules

$$\begin{aligned}
E &\longrightarrow E + T,\\
E &\longrightarrow T,\\
T &\longrightarrow T * F,\\
T &\longrightarrow F,\\
F &\longrightarrow (E),\\
F &\longrightarrow a.
\end{aligned}$$

We get $I_E = \{T, F\}$, $I_T = \{F\}$, and $I_F = \emptyset$. The new grammar G_3' has the set of rules

$$\begin{aligned}
E &\longrightarrow E + T,\\
E &\longrightarrow T * F,\\
E &\longrightarrow (E),\\
E &\longrightarrow a,\\
T &\longrightarrow T * F,\\
T &\longrightarrow (E),\\
T &\longrightarrow a,\\
F &\longrightarrow (E),\\
F &\longrightarrow a.
\end{aligned}$$

At this stage, the grammar obtained in lemma 2.4.2 no longer has ϵ-rules (except perhaps $S' \to \epsilon$ iff $\epsilon \in L(G)$) or chain rules. However, it may contain rules $A \to \alpha$ with $|\alpha| \geq 3$, or with $|\alpha| \geq 2$ and where α contains terminals(s). To obtain the Chomsky Normal Form. we need to eliminate such rules. This is not difficult, but notationally a bit messy.

Lemma 2.4.3 *Given any context-free grammar $G = (V, \Sigma, P, S)$, one can construct a context-free grammar $G' = (V', \Sigma, P', S')$ such that $L(G') = L(G)$ and G' is in Chomsky Normal Form, that is, a grammar whose productions are of the form*

$$\begin{aligned} A &\to BC, \\ A &\to a, \quad \textit{or} \\ S' &\to \epsilon, \end{aligned}$$

where $A, B, C \in N'$, $a \in \Sigma$, $S' \to \epsilon$ is in G' iff $\epsilon \in L(G)$, and S' does not occur on the right-hand side of any production in P'.

Proof. First, we apply lemma 2.4.2, obtaining G_1. Let Σ_r be the set of terminals occurring on the right-hand side of rules $A \to \alpha \in P_1$, with $|\alpha| \geq 2$. For every $a \in \Sigma_r$, let X_a be a new nonterminal not in V_1. Let

$$P_2 = \{X_a \to a \mid a \in \Sigma_r\}.$$

Let $P_{1,r}$ be the set of productions

$$A \to \alpha_1 a_1 \alpha_2 \cdots \alpha_k a_k \alpha_{k+1},$$

where $a_1, \ldots, a_k \in \Sigma_r$ and $\alpha_i \in N_1^*$. For every production

$$A \to \alpha_1 a_1 \alpha_2 \cdots \alpha_k a_k \alpha_{k+1}$$

in $P_{1,r}$, let

$$A \to \alpha_1 X_{a_1} \alpha_2 \cdots \alpha_k X_{a_k} \alpha_{k+1}$$

be a new production, and let P_3 be the set of all such productions. Let $P_4 = (P_1 - P_{1,r}) \cup P_2 \cup P_3$. Now, productions $A \to \alpha$ in P_4 with $|\alpha| \geq 2$ do not contain terminals. However, we may still have productions $A \to \alpha \in P_4$ with $|\alpha| \geq 3$. We can perform some recoding using some new nonterminals. For every production of the form

$$A \to B_1 \cdots B_k,$$

where $k \geq 3$, create the new nonterminals

$$[B_1 \cdots B_{k-1}], [B_1 \cdots B_{k-2}], \cdots, [B_1 B_2 B_3], [B_1 B_2],$$

and the new productions

$$\begin{aligned} A &\to [B_1 \cdots B_{k-1}] B_k, \\ [B_1 \cdots B_{k-1}] &\to [B_1 \cdots B_{k-2}] B_{k-1}, \\ \cdots &\to \cdots, \\ [B_1 B_2 B_3] &\to [B_1 B_2] B_3, \\ [B_1 B_2] &\to B_1 B_2. \end{aligned}$$

All the productions are now in Chomsky Normal Form, and it is clear that the same language is generated. □

Applying the first phase of the method of lemma 2.4.3 to the grammar G_3', we get the rules

$$\begin{aligned} E &\longrightarrow E X_+ T, \\ E &\longrightarrow T X_* F, \\ E &\longrightarrow X_(E X_), \\ E &\longrightarrow a, \end{aligned}$$

$$
\begin{aligned}
T &\longrightarrow TX_*F,\\
T &\longrightarrow X_(EX_),\\
T &\longrightarrow a,\\
F &\longrightarrow X_(EX_),\\
F &\longrightarrow a,\\
X_+ &\longrightarrow +,\\
X_* &\longrightarrow *,\\
X_(&\longrightarrow (,\\
X_) &\longrightarrow).
\end{aligned}
$$

After applying the second phase of the method, we get the following grammar in Chomsky Normal Form:

$$
\begin{aligned}
E &\longrightarrow [EX_+]T,\\
[EX_+] &\longrightarrow EX_+,\\
E &\longrightarrow [TX_*]F,\\
[TX_*] &\longrightarrow TX_*,\\
E &\longrightarrow [X_(E]X_),\\
[X_(E] &\longrightarrow X_(E,\\
E &\longrightarrow a,\\
T &\longrightarrow [TX_*]F,\\
T &\longrightarrow [X_(E]X_),\\
T &\longrightarrow a,\\
F &\longrightarrow [X_(E]X_),\\
F &\longrightarrow a,\\
X_+ &\longrightarrow +,\\
X_* &\longrightarrow *,\\
X_(&\longrightarrow (,\\
X_) &\longrightarrow).
\end{aligned}
$$

For large grammars, it is often convenient to use the abbreviation which consists in grouping productions having a common left-hand side, and listing the right-hand sides separated by the symbol |. Thus, a group of productions

$$
\begin{aligned}
A &\to \alpha_1,\\
A &\to \alpha_2,\\
\cdots &\to \cdots,\\
A &\to \alpha_k,
\end{aligned}
$$

may be abbreviated as

$$A \to \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_k.$$

An interesting corollary of the CNF is the following decidability result. There is an algorithm which, given any context-free grammar G, given any string $w \in \Sigma^*$, decides whether $w \in L(G)$. Indeed, we first convert G to a grammar G' in Chomsky Normal Form. If $w = \epsilon$, we can test whether $\epsilon \in L(G)$, since this is the case iff $S' \to \epsilon \in P'$. If $w \neq \epsilon$, letting $n = |w|$, note that since the rules are of the form $A \to BC$ or $A \to a$, where

$a \in \Sigma$, any derivation for w has $n - 1 + n = 2n - 1$ steps. Thus, we enumerate all (leftmost) derivations of length $2n - 1$.

There are much better parsing algorithms than this naive algorithm. We now show that every regular language is context-free.

2.5 Regular Languages are Context-Free

The regular languages can be characterized in terms of very special kinds of context-free grammars, right-linear (and left-linear) context-free grammars.

Definition 2.5.1 A context-free grammar $G = (V, \Sigma, P, S)$ is *left-linear* iff its productions are of the form

$$A \to Ba,$$
$$A \to a,$$
$$A \to \epsilon.$$

where $A, B \in N$, and $a \in \Sigma$. A context-free grammar $G = (V, \Sigma, P, S)$ is *right-linear* iff its productions are of the form

$$A \to aB,$$
$$A \to a,$$
$$A \to \epsilon.$$

where $A, B \in N$, and $a \in \Sigma$.

The following lemma shows the equivalence between NFA's and right-linear grammars.

Lemma 2.5.2 *A language L is regular if and only if is it generated by some right-linear grammar.*

Proof. Let $L = L(D)$ for some DFA $D = (Q, \Sigma, \delta, q_0, F)$. We construct a right-linear grammar G as follows. Let $V = Q \cup \Sigma$, $S = q_0$, and let P be defined as follows:

$$P = \{p \to aq \mid q = \delta(p, a),\ p, q \in Q,\ a \in \Sigma\} \cup \{p \to \epsilon \mid p \in F\}.$$

It is easily shown by induction on the length of w that

$$p \stackrel{*}{\Longrightarrow} wq \quad \text{iff} \quad q = \delta^*(p, w),$$

and thus, $L(D) = L(G)$.

Conversely, let $G = (V, \Sigma, P, S)$ be a right-linear grammar. First, let $G = (V', \Sigma, P', S)$ be the right-linear grammar obtained from G by adding the new terminal E to N, replacing every rule in P of the form $A \to a$ where $a \in \Sigma$ by the rule $A \to aE$, and adding the rule $E \to \epsilon$. It is immediately verified that $L(G') = L(G)$. Next, we construct the NFA $M = (Q, \Sigma, \delta, q_0, F)$ as follows: $Q = N'$, $q_0 = S$, $F = \{A \in N' \mid A \to \epsilon\}$, and

$$\delta(A, a) = \{B \in N \mid A \to aB \in P'\},$$

for all $A \in N'$ and all $a \in \Sigma$. It is easily shown by induction on the length of w that

$$A \stackrel{*}{\Longrightarrow} wB \quad \text{iff} \quad B \in \delta^*(A, w),$$

and thus, $L(M) = L(G') = L(G)$. □

A similar lemma holds for left-linear grammars. It is also easily shown that the regular languages are exactly the languages generated by context-free grammars whose rules are of the form

$$A \to Bu,$$
$$A \to u,$$

where $A, B \in N$, and $u \in \Sigma^*$.

2.6 Useless Productions in Context-Free Grammars

Given a context-free grammar $G = (V, \Sigma, P, S)$, it may contain rules that are useless for a number of reasons. For example, consider the grammar $G_3 = (\{E, A, a, b\}, \{a, b\}, E, P)$, where P is the set of rules

$$E \longrightarrow aEb,$$
$$E \longrightarrow ab,$$
$$E \longrightarrow A,$$
$$A \longrightarrow bAa.$$

The problem is that the nonterminal A does not derive any terminal strings, and thus, it is useless, as well as the last two productions. Let us now consider the grammar $G_4 = (\{E, A, a, b, c, d\}, \{a, b, c, d\}, E, P)$, where P is the set of rules

$$E \longrightarrow aEb,$$
$$E \longrightarrow ab,$$
$$A \longrightarrow cAd,$$
$$A \longrightarrow cd.$$

This time, the nonterminal A generates strings of the form $c^n d^n$, but there is no derivation $E \overset{+}{\Longrightarrow} \alpha$ from E where A occurs in α. The nonterminal A is not connected to E, and the last two rules are useless. Fortunately, it is possible to find such useless rules, and to eliminate them.

Let $T(G)$ be the set of nonterminals that actually derive some terminal string, i.e.

$$T(G) = \{A \in (V - \Sigma) \mid \exists w \in \Sigma^*,\ A \Longrightarrow^+ w\}.$$

The set $T(G)$ can be defined by stages. We define the sets T_n $(n \geq 1)$ as follows:

$$T_1 = \{A \in (V - \Sigma) \mid \exists (A \longrightarrow w) \in P, \text{ with } w \in \Sigma^*\},$$

and

$$T_{n+1} = T_n \cup \{A \in (V - \Sigma) \mid \exists (A \longrightarrow \beta) \in P, \text{ with } \beta \in (T_n \cup \Sigma)^*\}.$$

It is easy to prove that there is some least n such that $T_{n+1} = T_n$, and that for this n, $T(G) = T_n$.

If $S \notin T(G)$, then $L(G) = \emptyset$, and G is equivalent to the trivial grammar

$$G' = (\{S\}, \Sigma, \emptyset, S).$$

If $S \in T(G)$, then let $U(G)$ be the set of nonterminals that are actually useful, i.e.,

$$U(G) = \{A \in T(G) \mid \exists \alpha, \beta \in (T(G) \cup \Sigma)^*,\ S \Longrightarrow^* \alpha A \beta\}.$$

The set $U(G)$ can also be computed by stages. We define the sets U_n $(n \geq 1)$ as follows:

$$U_1 = \{A \in T(G) \mid \exists (S \longrightarrow \alpha A \beta) \in P, \text{ with } \alpha, \beta \in (T(G) \cup \Sigma)^*\},$$

and

$$U_{n+1} = U_n \cup \{B \in T(G) \mid \exists (A \longrightarrow \alpha B \beta) \in P, \text{ with } A \in U_n,\ \alpha, \beta \in (T(G) \cup \Sigma)^*\}.$$

It is easy to prove that there is some least n such that $U_{n+1} = U_n$, and that for this n, $U(G) = U_n \cup \{S\}$. Then, we can use $U(G)$ to transform G into an equivalent CFG in which every nonterminal is useful (i.e., for which $V - \Sigma = U(G)$). Indeed, simply delete all rules containing symbol not in $U(G)$. The details are left as an exercise.

It should be noted than although dull, the above considerations are important in practice. Certain algorithms for constructing parsers, for example, LR-parsers, may loop if useless rules are not eliminated!

We now consider another normal form for context-free grammars, the Greibach Normal Form.

2.7 The Greibach Normal Form

Every CFG G can also be converted to an equivalent grammar in *Greibach Normal Form (for short, GNF).* A context-free grammar $G = (V, \Sigma, P, S)$ is in Greibach Normal Form iff its productions are of the form

$$\begin{aligned} A &\to aBC, \\ A &\to aB, \\ A &\to a, \quad \text{or} \\ S &\to \epsilon, \end{aligned}$$

where $A, B, C \in N$, $a \in \Sigma$, $S \to \epsilon$ is in G iff $\epsilon \in L(G)$, and S does not occur on the right-hand side of any production.

Note that a grammar in Greibach Normal Form does not have ϵ-rules other than possibly $S \to \epsilon$. More importantly, except for the special rule $S \to \epsilon$, every rule produces some terminal symbol.

An important consequence of the Greibach Normal Form is that every nonterminal is not left recursive. A nonterminal A is *left recursive* iff $A \overset{+}{\Longrightarrow} A\alpha$ for some $\alpha \in V^*$. Left recursive nonterminals cause top-down deterministic parsers to loop. The Greibach Normal Form provides a way of avoiding this problem.

There are no easy proofs that every CFG can be converted to a Greibach Normal Form. A particularly elegant method due to Rosenkrantz using least fixed-points and matrices will be given in section 2.10.

Lemma 2.7.1 *Given any context-free grammar $G = (V, \Sigma, P, S)$, one can construct a context-free grammar $G' = (V', \Sigma, P', S')$ such that $L(G') = L(G)$ and G' is in Greibach Normal Form, that is, a grammar whose productions are of the form*

$$\begin{aligned} A &\to aBC, \\ A &\to aB, \\ A &\to a, \quad \textit{or} \\ S' &\to \epsilon, \end{aligned}$$

where $A, B, C \in N'$, $a \in \Sigma$, $S' \to \epsilon$ is in G' iff $\epsilon \in L(G)$, and S' does not occur on the right-hand side of any production in P'.

2.8 Least Fixed-Points

Context-free languages can also be characterized as least fixed-points of certain functions induced by grammars. This characterization yields a rather quick proof that every context-free grammar can be converted to Greibach Normal Form. This characterization also reveals very clearly the recursive nature of the context-free languages.

We begin by reviewing what we need from the theory of partially ordered sets.

Definition 2.8.1 Given a partially ordered set $\langle A, \leq \rangle$, an *ω-chain* $(a_n)_{n \geq 0}$ is a sequence such that $a_n \leq a_{n+1}$ for all $n \geq 0$. The *least-upper bound* of an ω-chain (a_n) is an element $a \in A$ such that:

(1) $a_n \leq a$, for all $n \geq 0$;

(2) For any $b \in A$, if $a_n \leq b$, for all $n \geq 0$, then $a \leq b$.

A partially ordered set $\langle A, \leq \rangle$ is an *ω-chain complete poset* iff it has a least element $\bot$, and iff every ω-chain has a least upper bound denoted as $\bigsqcup a_n$.

Remark: The ω in ω-chain means that we are considering countable chains (ω is the ordinal associated with the order-type of the set of natural numbers). This notation may seem arcane, but is standard in denotational semantics.

For example, given any set X, the power set 2^X ordered by inclusion is an ω-chain complete poset with least element $\emptyset$. The Cartesian product $\underbrace{2^X \times \cdots \times 2^X}_{n}$ ordered such that

$$(A_1, \ldots, A_n) \leq (B_1, \ldots, B_n)$$

iff $A_i \subseteq B_i$ (where $A_i, B_i \in 2^X$) is an ω-chain complete poset with least element $(\emptyset, \ldots, \emptyset)$.

We are interested in functions between partially ordered sets.

Definition 2.8.2 Given any two partially ordered sets $\langle A_1, \leq_1 \rangle$ and $\langle A_2, \leq_2 \rangle$, a function $f\colon A_1 \to A_2$ is *monotonic* iff for all $x, y \in A_1$,

$$x \leq_1 y \quad \text{implies that} \quad f(x) \leq_2 f(y).$$

If $\langle A_1, \leq_1 \rangle$ and $\langle A_2, \leq_2 \rangle$ are ω-chain complete posets, a function $f\colon A_1 \to A_2$ is ω-continuous iff it is monotonic, and for every ω-chain (a_n),

$$f(\bigsqcup a_n) = \bigsqcup f(a_n).$$

Remark: Note that we are not requiring that an ω-continuous function $f\colon A_1 \to A_2$ preserve least elements, i.e., it is possible that $f(\bot_1) \neq \bot_2$.

We now define the crucial concept of a least fixed-point.

Definition 2.8.3 Let $\langle A, \leq \rangle$ be a partially ordered set, and let $f\colon A \to A$ be a function. A *fixed-point of f* is an element $a \in A$ such that $f(a) = a$. The *least fixed-point of f* is an element $a \in A$ such that $f(a) = a$, and for every $b \in A$ such that $f(b) = b$, then $a \leq b$.

The following lemma gives sufficient conditions for the existence of least fixed-points. It is one of the key lemmas in denotational semantics.

Lemma 2.8.4 *Let $\langle A, \leq \rangle$ be an ω-chain complete poset with least element $\bot$. Every ω-continuous function $f\colon A \to A$ has a unique least fixed-point x_0 given by*

$$x_0 = \bigsqcup f^n(\bot).$$

Furthermore, for any $b \in A$ such that $f(b) \leq b$, then $x_0 \leq b$.

Proof. First, we prove that the sequence

$$\bot\,, f(\bot)\,, f^2(\bot),\ \ldots,\ f^n(\bot),\ \ldots$$

is an ω-chain. This is shown by induction on n. Since $\bot$ is the least element of A, we have $\bot \leq f(\bot)$. Assuming by induction that $f^n(\bot) \leq f^{n+1}(\bot)$, since f is ω-continuous, it is monotonic, and thus we get $f^{n+1}(\bot) \leq f^{n+2}(\bot)$, as desired.

Since A is an ω-chain complete poset, the ω-chain $(f^n(\bot))$ has a least upper bound

$$x_0 = \bigsqcup f^n(\bot).$$

Since f is ω-continuous, we have

$$f(x_0) = f(\bigsqcup f^n(\bot)) = \bigsqcup f(f^n(\bot)) = \bigsqcup f^{n+1}(\bot) = x_0,$$

and x_0 is indeed a fixed-point of f.

Clearly, if $f(b) \leq b$ implies that $x_0 \leq b$, then $f(b) = b$ implies that $x_0 \leq b$. Thus, assume that $f(b) \leq b$ for some $b \in A$. We prove by induction of n that $f^n(\bot) \leq b$. Indeed, $\bot \leq b$, since $\bot$ is the least element of A. Assuming by induction that $f^n(\bot) \leq b$, by monotonicity of f, we get

$$f(f^n(\bot)) \leq f(b),$$

and since $f(b) \leq b$, this yields

$$f^{n+1}(\bot)) \leq b.$$

Since $f^n(\bot) \leq b$ for all $n \geq 0$, we have

$$x_0 = \bigsqcup f^n(\bot) \leq b.$$

□

The second part of lemma 2.8.4 is very useful to prove that functions have the same least fixed-point. For example, under the conditions of lemma 2.8.4, if $g: A \to A$ is another ω-chain continuous function, letting x_0 be the least fixed-point of f and y_0 be the least fixed-point of g, if $f(y_0) \leq y_0$ and $g(x_0) \leq x_0$, we can deduce that $x_0 = y_0$. Indeed, since $f(y_0) \leq y_0$ and x_0 is the least fixed-point of f, we get $x_0 \leq y_0$, and since $g(x_0) \leq x_0$ and y_0 is the least fixed-point of g, we get $y_0 \leq x_0$, and therefore $x_0 = y_0$.

Lemma 2.8.4 also shows that the least fixed-point x_0 of f can be approximated as much as desired, using the sequence $(f^n(\bot))$. We will now apply this fact to context-free grammars. For this, we need to show how a context-free grammar $G = (V, \Sigma, P, S)$ with m nonterminals induces an ω-continuous map

$$\Phi_G: \underbrace{2^{\Sigma^*} \times \cdots \times 2^{\Sigma^*}}_{m} \to \underbrace{2^{\Sigma^*} \times \cdots \times 2^{\Sigma^*}}_{m}.$$

2.9 Context-Free Languages as Least Fixed-Points

Given any context-free grammar $G = (V, \Sigma, P, S)$ with m nonterminals $A_1, \ldots A_m$, grouping all the productions having the same left-hand side, the grammar G can be concisely written as

$$\begin{aligned}
A_1 &\to \alpha_{1,1} + \cdots + \alpha_{1,n_1},\\
\cdots &\to \cdots\\
A_i &\to \alpha_{i,1} + \cdots + \alpha_{i,n_i},\\
\cdots &\to \cdots\\
A_m &\to \alpha_{m,1} + \cdots + \alpha_{m,n_n}.
\end{aligned}$$

Given any set A, let $\mathcal{P}_{fin}(A)$ be the set of finite subsets of A.

Definition 2.9.1 Let $G = (V, \Sigma, P, S)$ be a context-free grammar with m nonterminals A_1, ..., A_m. For any m-tuple $\Lambda = (L_1, \ldots, L_m)$ of languages $L_i \subseteq \Sigma^*$, we define the function

$$\Phi[\Lambda]: \mathcal{P}_{fin}(V^*) \to 2^{\Sigma^*}$$

inductively as follows:

$$\begin{aligned}
\Phi[\Lambda](\emptyset) &= \emptyset,\\
\Phi[\Lambda](\{\epsilon\}) &= \{\epsilon\},\\
\Phi[\Lambda](\{a\}) &= \{a\}, \quad \text{if } a \in \Sigma,\\
\Phi[\Lambda](\{A_i\}) &= L_i, \quad \text{if } A_i \in N,\\
\Phi[\Lambda](\{\alpha X\}) &= \Phi[\Lambda](\{\alpha\})\Phi[\Lambda](\{X\}), \quad \text{if } \alpha \in V^+,\ X \in V,\\
\Phi[\Lambda](S \cup \{\alpha\}) &= \Phi[\Lambda](S) \cup \Phi[\Lambda](\{\alpha\}), \quad \text{if } S \in \mathcal{P}_{fin}(V^*),\ S \neq \emptyset,\ \alpha \in V^*,\ \alpha \notin S.
\end{aligned}$$

Then, writing the grammar G as

$$\begin{aligned}
A_1 &\to \alpha_{1,1} + \cdots + \alpha_{1,n_1},\\
\cdots &\to \cdots\\
A_i &\to \alpha_{i,1} + \cdots + \alpha_{i,n_i},\\
\cdots &\to \cdots\\
A_m &\to \alpha_{m,1} + \cdots + \alpha_{m,n_n},
\end{aligned}$$

we define the map

$$\Phi_G: \underbrace{2^{\Sigma^*} \times \cdots \times 2^{\Sigma^*}}_{m} \to \underbrace{2^{\Sigma^*} \times \cdots \times 2^{\Sigma^*}}_{m}$$

such that

$$\Phi_G(L_1, \ldots L_m) = (\Phi[\Lambda](\{\alpha_{1,1}, \ldots, \alpha_{1,n_1}\}), \ldots, \Phi[\Lambda](\{\alpha_{m,1}, \ldots, \alpha_{m,n_m}\}))$$

for all $\Lambda = (L_1, \ldots, L_m) \in \underbrace{2^{\Sigma^*} \times \cdots \times 2^{\Sigma^*}}_{m}$.

One should verify that the map $\Phi[\Lambda]$ is well defined, but this is easy. The following lemma is easily shown.

Lemma 2.9.2 *Given a context-free grammar $G = (V, \Sigma, P, S)$ with m nonterminals A_1, ..., A_m, the map*

$$\Phi_G: \underbrace{2^{\Sigma^*} \times \cdots \times 2^{\Sigma^*}}_{m} \to \underbrace{2^{\Sigma^*} \times \cdots \times 2^{\Sigma^*}}_{m}$$

is ω-continuous.

Now, $\underbrace{2^{\Sigma^*} \times \cdots \times 2^{\Sigma^*}}_{m}$ is an ω-chain complete poset, and the map Φ_G is ω-continuous. Thus, by lemma 2.8.4, the map Φ_G has a least-fixed point. It turns out that the components of this least fixed-point are precisely the languages generated by the grammars (V, Σ, P, A_i). Before proving this fact, let us give an example illustrating it.

Example. Consider the grammar $G = (\{A, B, a, b\}, \{a, b\}, P, A)$ defined by the rules

$$\begin{aligned}
A &\to BB + ab,\\
B &\to aBb + ab.
\end{aligned}$$

The least fixed-point of Φ_G is the least upper bound of the chain

$$(\Phi_G^n(\emptyset, \emptyset)) = ((\Phi_{G,A}^n(\emptyset, \emptyset), \Phi_{G,B}^n(\emptyset, \emptyset)),$$

where

$$\Phi_{G,A}^0(\emptyset, \emptyset) = \Phi_{G,B}^0(\emptyset, \emptyset) = \emptyset,$$

and

$$\begin{aligned}
\Phi_{G,A}^{n+1}(\emptyset, \emptyset) &= \Phi_{G,B}^n(\emptyset, \emptyset)\Phi_{G,B}^n(\emptyset, \emptyset) \cup \{ab\},\\
\Phi_{G,B}^{n+1}(\emptyset, \emptyset) &= a\Phi_{G,B}^n(\emptyset, \emptyset)b \cup \{ab\}.
\end{aligned}$$

It is easy to verify that

$$\begin{aligned}
&\Phi^1_{G,A}(\emptyset,\emptyset) = \{ab\},\\
&\Phi^1_{G,B}(\emptyset,\emptyset) = \{ab\},\\
&\Phi^2_{G,A}(\emptyset,\emptyset) = \{ab, abab\},\\
&\Phi^2_{G,B}(\emptyset,\emptyset) = \{ab, aabb\},\\
&\Phi^3_{G,A}(\emptyset,\emptyset) = \{ab, aabbab, aabbaabb, abaabb\},\\
&\Phi^3_{G,B}(\emptyset,\emptyset) = \{ab, aabb, aaabbb\}.
\end{aligned}$$

By induction, we can easily prove that the two components of the least fixed-point are the languages

$$L_A = \{a^m b^m a^n b^n \mid m, n \geq 1\} \cup \{ab\} \quad \text{and} \quad L_B = \{a^n b^n \mid n \geq 1\}.$$

Letting $G_A = (\{A, B, a, b\}, \{a, b\}, P, A)$ and $G_B = (\{A, B, a, b\}, \{a, b\}, P, B)$, it is indeed true that $L_A = L(G_A)$ and $L_B = L(G_B)$.

We have the following theorem due to Ginsburg and Rose.

Theorem 2.9.3 *Given any context-free grammar $G = (V, \Sigma, P, S)$ with m nonterminals A_1, ..., A_m, the least fixed-point of the map Φ_G is the m-tuple of languages*

$$(L(G_{A_1}), \ldots, L(G_{A_m})),$$

where $G_{A_i} = (V, \Sigma, P, A_i)$.

Proof. Writing G as

$$\begin{aligned}
A_1 &\to \alpha_{1,1} + \cdots + \alpha_{1,n_1},\\
&\cdots \to \cdots\\
A_i &\to \alpha_{i,1} + \cdots + \alpha_{i,n_i},\\
&\cdots \to \cdots\\
A_m &\to \alpha_{m,1} + \cdots + \alpha_{m,n_n},
\end{aligned}$$

let $M = \max\{|\alpha_{i,j}|\}$ be the maximum length of right-hand sides of rules in P. Let

$$\Phi^n_G(\emptyset, \ldots, \emptyset) = (\Phi^n_{G,1}(\emptyset, \ldots, \emptyset), \ldots, \Phi^n_{G,m}(\emptyset, \ldots, \emptyset)).$$

Then, for any $w \in \Sigma^*$, observe that

$$w \in \Phi^1_{G,i}(\emptyset, \ldots, \emptyset)$$

iff there is some rule $A_i \to \alpha_{i,j}$ with $w = \alpha_{i,j}$, and that

$$w \in \Phi^n_{G,i}(\emptyset, \ldots, \emptyset)$$

for some $n \geq 2$ iff there is some rule $A_i \to \alpha_{i,j}$ with $\alpha_{i,j}$ of the form

$$\alpha_{i,j} = u_1 A_{j_1} u_2 \cdots u_k A_{j_k} u_{k+1},$$

where $u_1, \ldots, u_{k+1} \in \Sigma^*$, $k \geq 1$, and some $w_1, \ldots, w_k \in \Sigma^*$ such that

$$w_h \in \Phi^{n-1}_{G,j_h}(\emptyset, \ldots, \emptyset),$$

and

$$w = u_1 w_1 u_2 \cdots u_k w_k u_{k+1}.$$

We prove the following two claims.

Claim 1: For every $w \in \Sigma^*$, if $A_i \overset{n}{\Longrightarrow} w$, then $w \in \Phi^p_{G,i}(\emptyset, \ldots, \emptyset)$, for some $p \geq 1$.

Claim 2: For every $w \in \Sigma^*$, if $w \in \Phi^n_{G,i}(\emptyset, \ldots, \emptyset)$, with $n \geq 1$, then $A_i \overset{p}{\Longrightarrow} w$ for some $p \leq (M+1)^{n-1}$.

Proof of Claim 1. We proceed by induction on n. If $A_i \overset{1}{\Longrightarrow} w$, then $w = \alpha_{i,j}$ for some rule $A \to \alpha_{i,j}$, and by the remark just before the claim, $w \in \Phi^1_{G,i}(\emptyset, \ldots, \emptyset)$.

If $A_i \overset{n+1}{\Longrightarrow} w$ with $n \geq 1$, then

$$A_i \overset{n}{\Longrightarrow} \alpha_{i,j} \Longrightarrow w$$

for some rule $A_i \to \alpha_{i,j}$. If

$$\alpha_{i,j} = u_1 A_{j_1} u_2 \cdots u_k A_{j_k} u_{k+1},$$

where $u_1, \ldots, u_{k+1} \in \Sigma^*$, $k \geq 1$, then $A_{j_h} \overset{n_h}{\Longrightarrow} w_h$, where $n_h \leq n$, and

$$w = u_1 w_1 u_2 \cdots u_k w_k u_{k+1}$$

for some $w_1, \ldots, w_k \in \Sigma^*$. By the induction hypothesis,

$$w_h \in \Phi^{p_h}_{G,j_h}(\emptyset, \ldots, \emptyset),$$

for some $p_h \geq 1$, for every h, $1 \leq h \leq k$. Letting $p = \max\{p_1, \ldots, p_k\}$, since each sequence $(\Phi^q_{G,i}(\emptyset, \ldots, \emptyset))$ is an ω-chain, we have $w_h \in \Phi^p_{G,j_h}(\emptyset, \ldots, \emptyset)$ for every h, $1 \leq h \leq k$, and by the remark just before the claim, $w \in \Phi^{p+1}_{G,i}(\emptyset, \ldots, \emptyset)$. □

Proof of Claim 2. We proceed by induction on n. If $w \in \Phi^1_{G,i}(\emptyset, \ldots, \emptyset)$, by the remark just before the claim, then $w = \alpha_{i,j}$ for some rule $A \to \alpha_{i,j}$, and $A_i \overset{1}{\Longrightarrow} w$.

If $w \in \Phi^n_{G,i}(\emptyset, \ldots, \emptyset)$ for some $n \geq 2$, then there is some rule $A_i \to \alpha_{i,j}$ with $\alpha_{i,j}$ of the form

$$\alpha_{i,j} = u_1 A_{j_1} u_2 \cdots u_k A_{j_k} u_{k+1},$$

where $u_1, \ldots, u_{k+1} \in \Sigma^*$, $k \geq 1$, and some $w_1, \ldots, w_k \in \Sigma^*$ such that

$$w_h \in \Phi^{n-1}_{G,j_h}(\emptyset, \ldots, \emptyset),$$

and

$$w = u_1 w_1 u_2 \cdots u_k w_k u_{k+1}.$$

By the induction hypothesis, $A_{j_h} \overset{p_h}{\Longrightarrow} w_h$ with $p_h \leq (M+1)^{n-2}$, and thus

$$A_i \Longrightarrow u_1 A_{j_1} u_2 \cdots u_k A_{j_k} u_{k+1} \overset{p_1}{\Longrightarrow} \cdots \overset{p_k}{\Longrightarrow} w,$$

so that $A_i \overset{p}{\Longrightarrow} w$ with

$$p \leq p_1 + \cdots + p_k + 1 \leq M(M+1)^{n-2} + 1 \leq (M+1)^{n-1},$$

since $k \leq M$. □

Combining Claim 1 and Claim 2, we have

$$L(G_{A_i}) = \bigcup_n \Phi^n_{G,i}(\emptyset, \ldots, \emptyset),$$

which proves that the least fixed-point of the map Φ_G is the m-tuple of languages

$$(L(G_{A_1}), \ldots, L(G_{A_m})).$$

□

We now show how theorem 2.9.3 can be used to give a short proof that every context-free grammar can be converted to Greibach Normal Form.

2.10 Least Fixed-Points and the Greibach Normal Form

The hard part in converting a grammar $G = (V, \Sigma, P, S)$ to Greibach Normal Form is to convert it to a grammar in so-called *weak Greibach Normal Form*, where the productions are of the form

$$\begin{aligned} A &\to a\alpha, \quad \text{or} \\ S &\to \epsilon, \end{aligned}$$

where $a \in \Sigma$, $\alpha \in V^*$, and if $S \to \epsilon$ is a rule, then S does not occur on the right-hand side of any rule. Indeed, if we first convert G to Chomsky Normal Form, it turns out that we will get rules of the form $A \to aBC$, $A \to aBC$ or $A \to a$.

Using the algorithm for eliminating ϵ-rules and chain rules, we can first convert the original grammar to a grammar with no chain rules and no ϵ-rules except possibly $S \to \epsilon$, in which case, S does not appear on the right-hand side of rules. Thus, for the purpose of converting to weak Greibach Normal Form, we can assume that we are dealing with grammars without chain rules and without ϵ-rules. Let us also assume that we computed the set $T(G)$ of nonterminals that actually derive some terminal string, and that useless productions involving symbols not in $T(G)$ have been deleted.

Let us explain the idea of the conversion using the following grammar:

$$\begin{aligned} A &\to AaB + BB + b. \\ B &\to Bd + BAa + aA + c. \end{aligned}$$

The first step is to group the right-hand sides α into two categories: those whose leftmost symbol is a terminal ($\alpha \in \Sigma V^*$) and those whose leftmost symbol is a nonterminal ($\alpha \in NV^*$). It is also convenient to adopt a matrix notation, and we can write the above grammar as

$$(A, B) = (A, B) \begin{pmatrix} aB & \emptyset \\ B & \{d, Aa\} \end{pmatrix} + (b, \{aA, c\})$$

Thus, we are dealing with matrices (and row vectors) whose entries are finite subsets of V^*. For notational simplicity, braces around singleton sets are omitted. The finite subsets of V^* form a semiring, where addition is union, and multiplication is concatenation. Addition and multiplication of matrices are as usual, except that the semiring operations are used. We will also consider matrices whose entries are languages over Σ. Again, the languages over Σ form a semiring, where addition is union, and multiplication is concatenation. The identity element for addition is $\emptyset$, and the identity element for multiplication is $\{\epsilon\}$. As above, addition and multiplication of matrices are as usual, except that the semiring operations are used. For example, given any languages $A_{i,j}$ and $B_{i,j}$ over Σ, where $i, j \in \{1, 2\}$, we have

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} = \begin{pmatrix} A_{1,1}B_{1,1} \cup A_{1,2}B_{2,1} & A_{1,1}B_{1,2} \cup A_{1,2}B_{2,2} \\ A_{2,1}B_{1,1} \cup A_{2,2}B_{2,1} & A_{2,1}B_{1,2} \cup A_{2,2}B_{2,2} \end{pmatrix}$$

Letting $X = (A, B)$, $K = (b, \{aA, c\})$, and

$$H = \begin{pmatrix} aB & \emptyset \\ B & \{d, Aa\} \end{pmatrix}$$

the above grammar can be concisely written as

$$X = XH + K.$$

More generally, given any context-free grammar $G = (V, \Sigma, P, S)$ with m nonterminals $A_1, \ldots, A_m$, assuming that there are no chain rules, no ϵ-rules, and that every nonterminal belongs to $T(G)$, letting

$$X = (A_1, \ldots, A_m),$$

we can write G as

$$X = XH + K,$$

for some appropriate $m \times m$ matrix H in in which every entry contains a set (possibly empty) of strings in V^+, and some row vector K in which every entry contains a set (possibly empty) of strings α each beginning with a terminal ($\alpha \in \Sigma V^*$).

Given an $m \times m$ square matrix $A = (A_{i,j})$ of languages over Σ, we can define the matrix A^* whose entry $A^*_{i,j}$ is given by

$$A^*_{i,j} = \bigcup_{n \geq 0} A^n_{i,j},$$

where $A^0 = Id_m$, the identity matrix, and A^n is the n-th power of A. Similarly, we define

$$A^+ = (\bigcup_{n \geq 1} A^n_{i,j}).$$

Given a matrix A where the entries are finite subset of V^*, where $A = \{A_1, \ldots, A_m\}$, for any m-tuple $\Lambda = (L_1, \ldots, L_m)$ of languages over Σ, we let

$$\Phi[\Lambda](A) = (\Phi[\Lambda](A_{i,j})).$$

Given a system $X = XH + K$ where H is an $m \times m$ matrix and X, K are row matrices, if H and K do not contain any nonterminals, we claim that the least fixed-point of the grammar G associated with $X = XH+K$ is KH^*. This is easily seen by computing the approximations $X^n = \Phi^n_G(\emptyset, \ldots, \emptyset)$. Indeed, $X^0 = K$, and

$$X^n = KH^n + KH^{n-1} + \cdots + KH + K = K(H^n + H^{n-1} + \cdots + H + I_m).$$

Similarly, if Y is an $m \times m$ matrix of nonterminals, the least fixed-point of the grammar associated with $Y = HY + H$ is H^+ (provided that H does not contain any nonterminals).

Given any context-free grammar $G = (V, \Sigma, P, S)$ with m nonterminals $A_1, \ldots, A_m$, writing G as $X = XH + K$ as explained earlier, we can form another grammar GH by creating m^2 new nonterminals $Y_{i,j}$, where the rules of this new grammar are defined by the system of two matrix equations

$$X = KY + K,$$
$$Y = HY + H,$$

where $Y = (Y_{i,j})$.

The following lemma is the key to the Greibach Normal Form.

Lemma 2.10.1 *Given any context-free grammar $G = (V, \Sigma, P, S)$ with m nonterminals $A_1, \ldots, A_m$, writing G as*

$$X = XH + K$$

as explained earlier, if GH is the grammar defined by the system of two matrix equations

$$X = KY + K,$$
$$Y = HY + H,$$

as explained above, then the components in X of the least-fixed points of the maps Φ_G and Φ_{GH} are equal.

Proof. Let U be the least-fixed point of Φ_G, and let (V, W) be the least fixed-point of Φ_{GH}. We shall prove that $U = V$. For notational simplicity, let us denote $\Phi[U](H)$ as $H[U]$ and $\Phi[U](K)$ as $K[U]$.

Since U is the least fixed-point of $X = XH + K$, we have

$$U = UH[U] + K[U].$$

Since $H[U]$ and $K[U]$ do not contain any nonterminals, by a previous remark, $K[U]H^*[U]$ is the least-fixed point of $X = XH[U] + K[U]$, and thus,

$$K[U]H^*[U] \leq U.$$

On the other hand, by monotonicity,

$$K[U]H^*[U]H\Big[K[U]H^*[U]\Big] + K\Big[K[U]H^*[U]\Big] \leq K[U]H^*[U]H[U] + K[U] = K[U]H^*[U],$$

and since U is the least fixed-point of $X = XH + K$,

$$U \leq K[U]H^*[U].$$

Therefore, $U = K[U]H^*[U]$. We can prove in a similar manner that $W = H[V]^+$.

Let $Z = H[U]^+$. We have

$$K[U]Z + K[U] = K[U]H[U]^+ + K[U] = K[U]H[U]^* = U,$$

and

$$H[U]Z + H[U] = H[U]H[U]^+ + H[U] = H[U]^+ = Z,$$

and since (V, W) is the least fixed-point of $X = KY + K$ and $Y = HY + H$, we get $V \leq U$ and $W \leq H[U]^+$.

We also have

$$V = K[V]W + K[V] = K[V]H[V]^+ + K[V] = K[V]H[V]^*,$$

and

$$VH[V] + K[V] = K[V]H[V]^*H[V] + K[V] = K[V]H[V]^* = V,$$

and since U is the least fixed-point of $X = XH + K$, we get $U \leq V$. Therefore, $U = V$, as claimed. □

Note that the above lemma actually applies to any grammar. Applying lemma 2.10.1 to our example grammar, we get the following new grammar:

$$(A, B) = (b, \{aA, c\}) \begin{pmatrix} Y_1 & Y_2 \\ Y_3 & Y_4 \end{pmatrix} + (b, \{aA, c\}),$$
$$\begin{pmatrix} Y_1 & Y_2 \\ Y_3 & Y_4 \end{pmatrix} = \begin{pmatrix} aB & \emptyset \\ B & \{d, Aa\} \end{pmatrix} \begin{pmatrix} Y_1 & Y_2 \\ Y_3 & Y_4 \end{pmatrix} + \begin{pmatrix} aB & \emptyset \\ B & \{d, Aa\} \end{pmatrix}$$

There are still some nonterminals appearing as leftmost symbols, but using the equations defining A and B, we can replace A with

$$\{bY_1, aAY_3, cY_3, b\}$$

and B with

$$\{bY_2, aAY_4, cY_4, aA, c\},$$

obtaining a system in weak Greibach Normal Form. This amounts to converting the matrix

$$H = \begin{pmatrix} aB & \emptyset \\ B & \{d, Aa\} \end{pmatrix}$$

to the matrix

$$L = \begin{pmatrix} aB & \emptyset \\ \{bY_2, aAY_4, cY_4, aA, c\} & \{d, bY_1a, aAY_3a, cY_3a, ba\} \end{pmatrix}$$

The weak Greibach Normal Form corresponds to the new system

$$X = KY + K,$$
$$Y = LY + L.$$

This method works in general for any input grammar with no ϵ-rules, no chain rules, and such that every nonterminal belongs to $T(G)$. Under these conditions, the row vector K contains some nonempty entry, all strings in K are in ΣV^*, and all strings in H are in V^+. After obtaining the grammar GH defined by the system

$$X = KY + K,$$
$$Y = HY + H,$$

we use the system $X = KY + K$ to express every nonterminal A_i in terms of expressions containing strings $\alpha_{i,j}$ involving a terminal as the leftmost symbol ($\alpha_{i,j} \in \Sigma V^*$), and we replace all leftmost occurrences of nonterminals in H (occurrences A_i in strings of the form $A_i\beta$, where $\beta \in V^*$) using the above expressions. In this fashion, we obtain a matrix L, and it is immediately shown that the system

$$X = KY + K,$$
$$Y = LY + L,$$

generates the same tuple of languages. Furthermore, this last system corresponds to a weak Greibach Normal Form.

It we start with a grammar in Chomsky Normal Form (with no production $S \to \epsilon$) such that every nonterminal belongs to $T(G)$, we actually get a Greibach Normal Form (the entries in K are terminals, and the entries in H are nonterminals). Thus, we have justified lemma 2.7.1. The method is also quite economical, since it introduces only m^2 new nonterminals. However, the resulting grammar may contain some useless nonterminals.

2.11 Tree Domains and Gorn Trees

Derivation trees play a very important role in parsing theory and in the proof of a strong version of the pumping lemma for the context-free languages known as Ogden's lemma. Thus, it is important to define derivation trees rigorously. We do so using Gorn trees.

Let $\mathbf{N}_+ = \{1, 2, 3, \ldots\}$.

Definition 2.11.1 A *tree domain* D is a nonempty subset of strings in $\mathbf{N}_+^*$ satisfying the conditions:

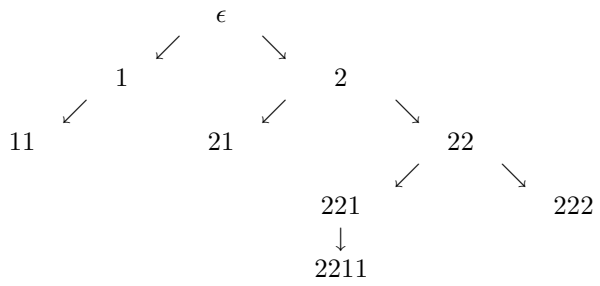
(1) For all $u, v \in D$, if $uv \in D$, then $u \in D$.

(2) For all $u \in D$, for every $i \in \mathbf{N}_+$, if $ui \in D$ then $uj \in D$ for every j, $1 \leq j \leq i$.

The tree domain

$$D = \{\epsilon, 1, 2, 11, 21, 22, 221, 222, 2211\}$$

is represented as follows:



A tree labeled with symbols from a set Δ is defined as follows.

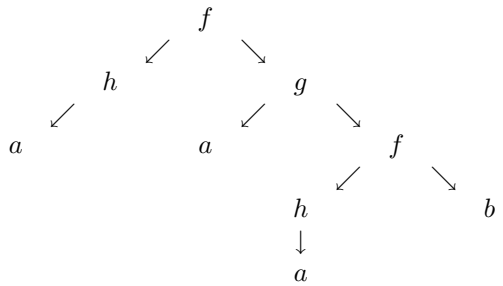
Definition 2.11.2 Given a set Δ of labels, a Δ-tree (for short, a *tree*) is a total function $t : D \to \Delta$, where D is a tree domain.

The domain of a tree t is denoted as $dom(t)$. Every string $u \in dom(t)$ is called a *tree address* or a *node*.

Let $\Delta = \{f, g, h, a, b\}$. The tree $t : D \to \Delta$, where D is the tree domain of the previous example and t is the function whose graph is

$$\{(\epsilon, f), (1, h), (2, g), (11, a), (21, a), (22, f), (221, h), (222, b), (2211, a)\}$$

is represented as follows:



The *outdegree* (sometimes called *ramification*) $r(u)$ of a node u is the cardinality of the set

$$\{i \mid ui \in dom(t)\}.$$

Note that the outdegree of a node can be infinite. Most of the trees that we shall consider will be *finite-branching*, that is, for every node u, $r(u)$ will be an integer, and hence finite. If the oudegree of all nodes in a tree is bounded by n, then we can view the domain of the tree as being defined over $\{1, 2, \ldots, n\}^*$.

A node of outdegree 0 is called a *leaf*. The node whose address is ϵ is called the *root* of the tree. A tree is *finite* if its domain $dom(t)$ is finite. Given a node u in $dom(t)$, every node of the form ui in $dom(t)$ with $i \in \mathbf{N}_+$ is called a *son* (or *immediate successor*) of u.

Tree addresses are totally ordered *lexicographically*: $u \leq v$ if either u is a prefix of v or, there exist strings $x, y, z \in \mathbf{N}_+^*$ and $i, j \in \mathbf{N}_+$, with $i < j$, such that $u = xiy$ and $v = xjz$.

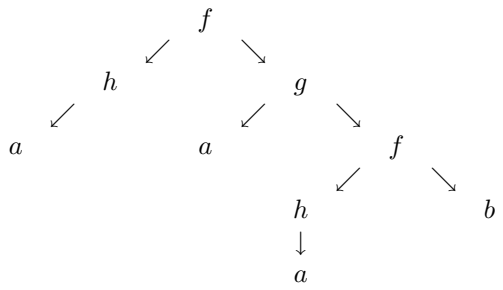
In the first case, we say that u is an *ancestor* (or *predecessor*) of v (or u *dominates* v) and in the second case, that u is *to the left* of v.

If $y = \epsilon$ and $z = \epsilon$, we say that xi is a *left brother* (or *left sibling*) of xj, $(i < j)$. Two tree addresses u and v are *independent* if u is not a prefix of v and v is not a prefix of u.

Given a finite tree t, the *yield* of t is the string

$$t(u_1)t(u_2)\cdots t(u_k),$$

where $u_1, u_2, \ldots, u_k$ is the sequence of leaves of t in lexicographic order.
For example, the yield of the tree below is *aaab*:



Given a finite tree t, the *depth* of t is the integer

$$d(t) = \max\{|u| \mid u \in dom(t)\}.$$

Given a tree t and a node u in $dom(t)$, the *subtree rooted at* u is the tree t/u, whose domain is the set

$$\{v \mid uv \in dom(t)\}$$

and such that $t/u(v) = t(uv)$ for all v in $dom(t/u)$.

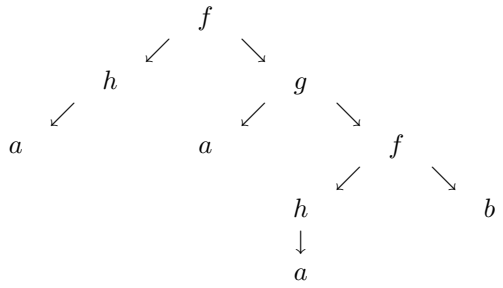
Another important operation is the operation of tree replacement (or tree substitution).

Definition 2.11.3 Given two trees t_1 and t_2 and a tree address u in t_1, the *result of replacing* t_2 at u in t_1, denoted by $t_1[u \leftarrow t_2]$, is the function whose graph is the set of pairs

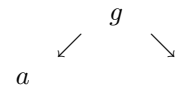
$$\{(v, t_1(v)) \mid u \text{ is not a prefix of } v\} \cup \{(uv, t_2(v)\}.$$

Let t_1 and t_2 be the trees defined by the following diagrams:

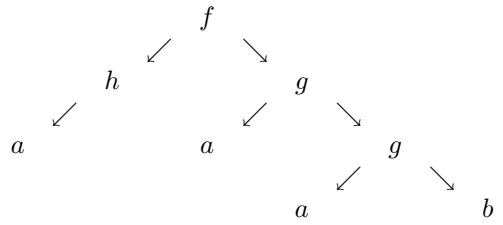
Tree t_1



Tree t_2



The tree $t_1[22 \leftarrow t_2]$ is defined by the following diagram:



We can now define derivation trees and relate derivations to derivation trees.

2.12 Derivations Trees

Definition 2.12.1 Given a context-free grammar $G = (V, \Sigma, P, S)$, for any $A \in N$, an *A-derivation tree for G* is a $(V \cup \{\epsilon\})$-tree t such that:

(1) $t(\epsilon) = A$;

(2) For every nonleaf node $u \in dom(t)$, if $u1, \ldots, uk$ are the successors of u, then either there is a production $A \to X_1 \cdots X_k$ in P such that $t(u) = A$ and $t(ui) = X_i$ for all i, $1 \leq i \leq k$, or $A \to \epsilon \in P$, $t(u) = A$ and $t(u1) = \epsilon$. A *complete derivation (or parse tree)* is an S-tree whose yield belongs to Σ^*.

A derivation tree for the grammar

$$G_3 = (\{E, T, F, +, *, (,), a\}, \{+, *, (,), a\}, E, P),$$

where P is the set of rules

$$E \longrightarrow E + T,$$
$$E \longrightarrow T,$$
$$T \longrightarrow T * F,$$
$$T \longrightarrow F,$$
$$F \longrightarrow (E),$$
$$F \longrightarrow a,$$

is shown below. The yield of the derivation tree is $a + a * a$.

(* Picture of tree goes here *)

Derivations trees are associated to derivations inductively as follows.

Definition 2.12.2 Given a context-free grammar $G = (V, \Sigma, P, S)$, for any $A \in N$, if $\pi : A \overset{n}{\Longrightarrow} \alpha$ is a derivation in G, we construct an *A-derivation tree t_π with yield α* as follows.

(1) If $n = 0$, then t_π is the one-node tree such that $dom(t_\pi) = \{\epsilon\}$ and $t_\pi(\epsilon) = A$.

(2) If $A \overset{n-1}{\Longrightarrow} \lambda B \rho \Longrightarrow \lambda\gamma\rho = \alpha$, then if t_1 is the A-derivation tree with yield $\lambda B\rho$ associated with the derivation $A \overset{n-1}{\Longrightarrow} \lambda B\rho$, and if t_2 is the tree associated with the production $A \to \gamma$ (that is, if

$$\gamma = X_1 \cdots X_k,$$

then $dom(t_2) = \{\epsilon, 1, \ldots, k\}$, $t_2(\epsilon) = A$, and $t_2(i) = X_i$ for all i, $1 \leq i \leq k$, or if $\gamma = \epsilon$, then $dom(t_2) = \{\epsilon, 1\}$, $t_2(\epsilon) = A$, and $t_2(1) = \epsilon$), then

$$t_\pi = t_1[u \leftarrow t_2],$$

where u is the address of the leaf labeled B in t_1. The tree t_π is the *A-derivation tree associated with the derivation $A \overset{n}{\Longrightarrow} \alpha$.*

Given the grammar

$$G_2 = (\{E, +, *, (,), a\}, \{+, *, (,), a\}, E, P),$$

where P is the set of rules

$$E \longrightarrow E + E,$$
$$E \longrightarrow E * E,$$
$$E \longrightarrow (E),$$
$$E \longrightarrow a,$$

the parse trees associated with two derivations of the string $a + a * a$ are shown below.

(* insert picture of trees here *)

The following lemma is easily shown.

Lemma 2.12.3 *Let $G = (V, \Sigma, P, S)$ be a context-free grammar. For any derivation $A \overset{n}{\Longrightarrow} \alpha$, there is a unique A-derivation tree associated with this derivation, with yield α. Conversely, for any A-derivation tree t with yield α, there is a unique leftmost derivation $A \underset{lm}{\overset{*}{\Longrightarrow}} \alpha$ in G having t as its associated derivation tree.*

We will now prove a strong version of the pumping lemma for context-free languages due to Bill Ogden (1968).

2.13 Ogden's Lemma

Ogden's lemma states some combinatorial properties of parse trees that are deep enough. The yield w of such a parse tree can be split into 5 substrings u, v, x, y, z such that

$$w = uvxyz,$$

where u, v, x, y, z satisfy certain conditions. It turns out that we get a more powerful version of the lemma is we allow ouselves to *mark* certain occurrences of symbols in w before invoking the lemma. We can imagine that *marked occurrences* in a nonempty string w are occurrences of symbols in w in boldface, or red, or any given color. For example, given $w = aaababbbaa$, we can mark the symbols of even index as follows:

$$a\mathbf{a}a\mathbf{b}a\mathbf{b}b\mathbf{b}a\mathbf{a}.$$

In view of lemma 2.4.1, we can assume without loss of generality that we are considering grammars without ϵ-rules (except perhaps $S' \to \epsilon$). Ogden's lemma only yields useful information for grammars G generating an infinite language. We could make this hypothesis, but it seems more elegant to use the precondition that the lemma only applies to strings $w \in L(D)$ such that w contains at least K marked occurrences, for a constant K large enough. If K is large enough, $L(G)$ will indeed be infinite.

Lemma 2.13.1 *For every context-free grammar (without ϵ-rules), there is some integer $K > 1$ such that, for every marked string $w \in \Sigma^+$, if $w \in L(G)$ and w contains at least K marked occurrences, then there exists some decomposition of w as $w = uvxyz$, and some $A \in N$, such that the following properties hold:*

(1) There are derivations $S \overset{+}{\Longrightarrow} uAz$, $A \overset{+}{\Longrightarrow} vAy$, and $A \overset{+}{\Longrightarrow} x$, so that

$$uv^n xy^n z \in L(G)$$

for all $n \geq 0$ (the pumping property);

(2) x contains some marked occurrence;

(3) Either (both u and v contain some marked occurrence), or (both y and z contain some marked occurrence);

(4) vxy contains less than K marked occurrences.

Proof. Let t be any parse tree for w. We call a leaf of t a *marked leaf* if its label is a marked occurrence in the marked string w. The general idea is to make sure that K is large enough so that parse trees with yield w contain enough repeated nonterminals along some path from the root to some marked leaf. Let $r = |N|$, and let

$$p = \max\{2, \max\{|\alpha| \mid (A \to \alpha) \in P\}\}.$$

We claim that $K = p^{2r+3}$ does the job.

Is is easily shown by induction on the depth of trees that if the outdegree of a tree t is bounded by $p \geq 2$, then if t has at least p^d leaves, the depth of t is at least d (and the maximum outdegree of nodes in t is at least 2. This is false for $p = 1$). Thus, if $K = p^{2r+3}$, since $p \geq 2$ and we are assuming that the marked input w has at least K marked occurrences, the depth of any parse tree for w is a least $2r + 3$.

The key concept in the proof is the notion of a B-node. Given a parse tree t, a *B-node* is a node with at least two immediate successors u_1, u_2, such that for $i = 1, 2$, either u_i is a marked leaf, or u_i has some marked leaf as a descendant. We consider all paths from the root to some marked leaf maximizing the number of B-nodes. Among such paths, define a path $(s_0, \ldots, s_n)$ from the root to some marked leaf, such that:

(i) s_0 is the root of t;

(ii) If s_j is in the path, s_j is not a leaf, and s_j has a single descendant which is a marked leaf v, let s_{j+1} be the immediate successor of s_j on the unique path from s_j to the marked leaf v;

(iii) If s_j is a B-node in the path, then let s_{j+1} be the leftmost immediate successors of s_j with the maximum number of marked leaves as descendants (assuming that if s_{j+1} is a marked leaf, then it is its own descendant).

(iv) If s_j is a leaf, then it is a marked leaf and $n = j$.

We will show that the path $(s_0, \ldots, s_n)$ contains at least $2r + 3$ B-nodes.

Claim: For every i, $0 \leq i \leq n$, if the path $(s_i, \ldots, s_n)$ contains at most b B-nodes, then s_i has at most p^b marked leaves as descendants.

Proof. We proceed by "backward induction", i.e., by induction on $n - i$. For $i = n$, there are no B-nodes, so that $b = 0$, and there is indeed $p^0 = 1$ marked leaf s_n. Assume that the claim holds for the path $(s_{i+1}, \ldots, s_n)$.

If s_i is not a B-node, then the number b of B-nodes in the path $(s_{i+1}, \ldots, s_n)$ is the same as the number of B-nodes in the path $(s_i, \ldots, s_n)$, and s_{i+1} is the only immediate successor of s_i having a marked leaf as descendant. By the induction hypothesis, s_{i+1} has at most p^b marked leaves as descendants, and this is also an upper bound on the number of marked leaves which are descendants of s_i.

If s_i is a B-node, then if there are b B-nodes in the path $(s_{i+1}, \ldots, s_n)$, there are $b + 1$ B-nodes in the path $(s_i, \ldots, s_n)$. By the induction hypothesis, s_{i+1} has at most p^b marked leaves as descendants. Since s_i is a B-node, s_{i+1} was chosen to be the leftmost immediate successor of s_i having the maximum number of marked leaves as descendants. Thus, since the outdegree of s_i is at most p, and each of its immediate successors has at most p^b marked leaves as descendants, the node s_i has at most $pp^d = p^{d+1}$ marked leaves as descendants, as desired. □

Applying the claim to s_0, since w has at least $K = p^{2r+3}$ marked occurrences, the path $(s_0, \ldots, s_n)$ contains at least $2r + 3$ B-nodes.

Let us now select the lowest $2r+3$ B-nodes in the path, $(s_0, \ldots, s_n)$, and denote them $(b_1, \ldots, b_{2r+3})$. Every B-node b_i has at least two immediate successors $u_i < v_i$ such that u_i or v_i is on the path $(s_0, \ldots, s_n)$. If the path goes through u_i, we say that b_i is a *right B-node* and if the path goes through v_i, we say that b_i is a *left B-node*. Since $2r + 3 = r + 2 + r + 1$, either there are $r + 2$ left B-nodes or there are $r + 2$ right B nodes in the path $(b_1, \ldots, b_{2r+3})$. Let us assume that there are $r + 2$ left B nodes, the other case being similar.

Let $(d_1, \ldots, d_{r+2})$ be the lowest $r + 2$ left B-nodes in the path. Since there are $r + 1$ B-nodes in the sequence $(d_2, \ldots, d_{r+2})$, and there are only r distinct nonterminals, there are two nodes d_i and d_j, with $2 \leq i < j \leq r + 2$, such that $t(d_i) = t(d_j) = A$, for some $A \in N$. We can assume that d_i is an ancestor of d_j, and thus, $d_j = d_i v$, for some $v \neq \epsilon$.

If we prune out the subtree t/d_i rooted at d_i from t, we get an S-derivation tree having a yield of the form uAz, and we have a derivation of the form $S \stackrel{+}{\Longrightarrow} uAz$, since there are at least $r + 2$ left B-nodes on the path, and we are looking at the lowest $r + 1$ left B-nodes. Considering the subtree t/d_i, pruning out the subtree t/d_j rooted at v in t/d_i, we get an A-derivation tree having a yield of the form vAy, and we have a derivation of the form $A \stackrel{+}{\Longrightarrow} vAy$. Finally, the subtree t/d_j is an A-derivation tree with yield x, and we have a derivation $A \stackrel{+}{\Longrightarrow} x$. This proves (1) of the lemma.

Since s_n is a marked leaf and a descendant of d_j, x contains some marked occurrence, proving (2).

Since d_1 is a left B-node, one of its immediate successors to the left of d_i leads to some marked leaf, and thus u contains some marked occurrence. Similarly, since d_i is a left B-node, one of its immediate successors to the left of d_{i+1} leads to some marked leaf, and thus v contains some marked occurrence. This proves (3).

Finally, since $(b_1, \ldots, b_{2r+3})$ are the lowest $2r + 3$ B-nodes in the path, and $(d_1, \ldots, d_{r+2})$ is a subsequence of $(b_1, \ldots, b_{2r+3})$, the sequence $(d_j, \ldots, b_{2r+3})$ has at most $2r + 2$ B-nodes, and by the claim shown earlier, d_j has at most p^{2r+2} marked leaves as descendants. Since $p^{2r+2} < p^{2r+3} = K$, this proves (4). □

Observe that condition (2) implies that $x \neq \epsilon$, and condition (3) implies that either $u \neq \epsilon$ and $v \neq \epsilon$, or $y \neq \epsilon$ and $z \neq \epsilon$. Thus, the pumping condition (1) implies that the set $\{uv^n xy^n z \mid n\}$ is an infinite subset of $L(G)$, and $L(G)$ is indeed infinite, as we mentioned earlier. The "standard pumping lemma" due to Bar-Hillel, Perles, and Shamir, is obtained by letting all occurrences be marked in $w \in L(G)$.

Lemma 2.13.2 *For every context-free grammar (without ϵ-rules), there is some integer $K > 1$ such that, for every string $w \in \Sigma^+$, if $w \in L(G)$ and $|w| \geq K$, then there exists some decomposition of w as $w = uvxyz$, and some $A \in N$, such that the following properties hold:*

(1) There are derivations $S \overset{+}{\Longrightarrow} uAz$, $A \overset{+}{\Longrightarrow} vAy$, and $A \overset{+}{\Longrightarrow} x$, so that

$$uv^n xy^n z \in L(G)$$

for all $n \geq 0$ (the pumping property);

(2) $x \neq \epsilon$;

(3) Either $v \neq \epsilon$ or $y \neq \epsilon$;

(4) $|vxy| \leq K$.

A stronger version could be stated, and we are just following tradition in stating this standard version of the pumping lemma.

The pumping lemma or Ogden's lemma can be used to show that certain languages are not context-free. As an illustration, we show that the language

$$L = \{a^n b^n c^n \mid n \geq 1\}$$

is not context-free. Since L is infinite, we will be able to use the pumping lemma.

The proof proceeds by contradiction. If L was context-free, there would be some context-free grammar G such that $L = L(G)$, and some constant $K > 1$ as in Ogden's lemma. Let $w = a^K b^K c^K$, and choose the $b's$ as marked occurrences. Then by Ogden's lemma, x contains some marked occurrence, and either both u, v or both y, z contain some marked occurrence. Assume that both u and v contain some b. We have the following situation:

$$\underbrace{a \cdots ab \cdots b}_{u}\underbrace{b \cdots b}_{v}\underbrace{b \cdots bc \cdots c}_{xyz}.$$

If we consider the string $uvvxyyz$, the number of a's is still K, but the number of b's is strictly greater than K since v contains at least one b, and thus $uvvxyyz \notin L$, a contradiction.

If both y and z contain some b we will also reach a contradiction because in the string $uvvxyyz$, the number of c's is still K, but the number of b's is strictly greater than K. Having reached a contradiction in all cases, we conclude that L is not context-free.

Let us now show that the language

$$L = \{a^m b^n c^m d^n \mid m, n \geq 1\}$$

is not context-free.

Again, we proceed by contradiction. This time, let

$$w = a^K b^K c^K d^K,$$

where the b's and c's are marked occurrences.

By Ogden's lemma, either both u, v contain some marked occurrence, or both y, z contain some marked occurrence, and x contains some marked occurrence. Let us first consider the case where both u, v contain some marked occurrence.

If v contains some b, since $uvvxyyz \in L$, v must contain only b's, since otherwise we would have a bad string in L, and we have the following situation:

$$\underbrace{a \cdots ab \cdots b}_{u}\underbrace{b \cdots b}_{v}\underbrace{b \cdots bc \cdots cd \cdots d}_{xyz}.$$

Since $uvvxyyz \in L$, the only way to preserve an equal number of b's and d's is to have $y \in d^+$. But then, vxy contains c^K, which contradicts (4) of Ogden's lemma.

If v contains some c, since $uvvxyyz \in L$, v must contain only c's, since otherwise we would have a bad string in L, and we have the following situation:

$$\underbrace{a \cdots ab \cdots bc \cdots c}_{u}\underbrace{c \cdots c}_{v}\underbrace{c \cdots cd \cdots d}_{xyz}.$$

Since $uvvxyyz \in L$ and the number of a's is still K whereas the number of c's is strictly more than K, this case is impossible.

Let us now consider the case where both y, z contain some marked occurrence. Reasoning as before, the only possibility is that $v \in a^+$ and $y \in c^+$:

$$\underbrace{a \cdots a}_{u}\underbrace{a \cdots a}_{v}\underbrace{a \cdots ab \cdots bc \cdots c}_{x}\underbrace{c \cdots c}_{y}\underbrace{c \cdots cd \cdots d}_{z}.$$

But then, vxy contains b^K, which contradicts (4) of Ogden's Lemma. Since a contradiction was obtained in all cases, L is not context-free.

Ogden's lemma can also be used to show that the language

$$\{a^m b^n c^n \mid m, n \geq 1\} \cup \{a^m b^m c^n \mid m, n \geq 1\}$$

is inherently ambiguous. The proof is quite involved.

Another corollary of the pumping lemma is that it is decidable whether a context-free grammar generates an infinite language.

Lemma 2.13.3 *Given any context-free grammar G (without ϵ-rules), there is some $K > 1$ such that $L(G)$ is infinite iff there is some $w \in L(G)$ such that $K \leq |w| < 2K$.*

Proof. If there is some $w \in L(G)$ such that $|w| \geq K$, we already observed that the pumping lemma implies that $L(G)$ contains an infinite subset of the form $\{uv^n xy^n z \mid n \geq 0\}$. Conversely, assume that $L(G)$ is infinite. If $|w| < K$ for all $w \in L(G)$, then $L(G)$ is finite. Thus, there is some $w \in L(G)$ such that $|w| \geq K$. Let $w \in L(G)$ be a minimal string such that $|w| \geq K$. By the pumping lemma, we can write w as $w = uvxyxz$, where $x \neq \epsilon$, $vy \neq \epsilon$, and $|vxy| \leq K$. By the pumping property, $uxz \in L(G)$. If $|w| \geq 2K$, then

$$|uxz| = |uvxyz| - |vy| > |uvxyz| - |vxy| \geq 2K - K = K,$$

and $|uxz| < |uvxyz|$, contradicting the minimality of w. Thus, we must have $|w| < 2K$. □

In particular, if G is in Chomsky Normal Form, it can be shown that we just have to consider derivations of length at most $4K - 3$.

2.14 Pushdown Automata

We have seen that the regular languages are exactly the languages accepted by DFA's or NFA's. The context-free languages are exactly the languages accepted by pushdown automata, for short, PDA's. However, although there are two versions of PDA's, deterministic and nondeterministic, contrary to the fact that every NFA can be converted to a DFA, nondeterministic PDA's are strictly more powerful than deterministic PDA's (DPDA's). Indeed, there are context-free languages that cannot be accepted by DPDA's. Thus, the natural machine model for the context-free languages is nondeterministic, and for this reason, we just use the abbreviation PDA, as opposed to NPDA.

We adopt a definition of a PDA in which the pushdown store, or stack, must not be empty for a move to take place. Other authors allow PDA's to make move when the stack is empty. Novices seem to be confused by such moves, and this is why we do not allow moves with an empty stack.

Intuitively, a PDA consists of an input tape, a nondeterministic finite-state control, and a stack.

Given any set X possibly infinite, let $\mathcal{P}_{fin}(X)$ be the set of all finite subsets of X.

Definition 2.14.1 A *pushdown automaton* is a 7-tuple
$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, where

- Q is a finite set of *states*;
- Σ is a finite *input alphabet*;
- Γ is a finite *pushdown store (or stack) alphabet*;
- $q_0 \in Q$ is the *start state* (or *initial state*);
- $Z_0 \in \Gamma$ is the *initial stack symbol* (or *bottom marker*);
- $F \subseteq Q$ is the set of *final (or accepting) states*;
- $\delta\colon Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \to \mathcal{P}_{fin}(Q \times \Gamma^*)$ is the *transition function.*

A transition is of the form $(q, \gamma) \in \delta(p, a, Z)$, where $p, q \in Q$, $Z \in \Gamma$, and $a \in \Sigma \cup \{\epsilon\}$. A transition of the form $(q, \gamma) \in \delta(p, \epsilon, Z)$ is called an *ϵ-transition (or ϵ-move).*

The way a PDA operates is explained in terms of *Instantaneous Descriptions*, for short ID's. Intuitively, an Instantaneous Description is a snapshot of the PDA. An ID is a triple of the form

$$(p, u, \alpha) \in Q \times \Sigma^* \times \Gamma^*.$$

The idea is that p is the current state, u is the remaining input, and α represents the stack.

It is important to note that we use the convention that the **leftmost** symbol in α represents the topmost stack symbol.

Given a PDA M, we define a relation $\vdash_M$ between pairs of ID's. This is very similar to the derivation relation $\Longrightarrow_G$ associated with a context-free grammar.
Intuitively, a PDA scans the input tape symbol by symbol from left to right, making moves that cause a change of state, an update to the stack (but only at the top), and either advancing the reading head to the next symbol, or not moving the reading head during an ϵ-move.

Definition 2.14.2 Given a PDA

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F),$$

the relation $\vdash_M$ is defined as follows:

(1) For any move $(q, \gamma) \in \delta(p, a, Z)$, where $p, q \in Q$, $Z \in \Gamma$, $a \in \Sigma$, for every ID of the form $(p, au, Z\alpha)$, we have

$$(p, au, Z\alpha) \vdash_M (q, u, \gamma\alpha).$$

(2) For any move $(q, \gamma) \in \delta(p, \epsilon, Z)$, where $p, q \in Q$, $Z \in \Gamma$, for every ID of the form $(p, u, Z\alpha)$, we have

$$(p, u, Z\alpha) \vdash_M (q, u, \gamma\alpha).$$

As usual, $\vdash_M^+$ is the transitive closure of $\vdash_M$, and $\vdash_M^*$ is the reflexive and transitive closure of $\vdash_M$.

A move of the form

$$(p, au, Z\alpha) \vdash_M (q, u, \alpha)$$

where $a \in \Sigma \cup \{\epsilon\}$, is called a *pop move*.

A move on a real input symbol $a \in \Sigma$ causes this input symbol to be consumed, and the reading head advances to the next input symbol. On the other hand, during an ϵ-move, the reading head stays put.

When

$$(p, u, \alpha) \vdash_M^* (q, v, \beta)$$

we say that we have a *computation*.

There are several equivalent ways of defining acceptance by a PDA.

Definition 2.14.3 Given a PDA

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F),$$

the following languages are defined:

(1) $T(M) = \{w \in \Sigma^* \mid (q_0, w, Z_0) \vdash_M^* (f, \epsilon, \alpha)$, where $f \in F$, and $\alpha \in \Gamma^*\}$.

We say that $T(M)$ is the *language accepted by M by final state.*

(2) $N(M) = \{w \in \Sigma^* \mid (q_0, w, Z_0) \vdash_M^* (q, \epsilon, \epsilon)$, where $q \in Q\}$.

We say that $N(M)$ is the *language accepted by M by empty stack.*

(3) $L(M) = \{w \in \Sigma^* \mid (q_0, w, Z_0) \vdash_M^* (f, \epsilon, \epsilon)$, where $f \in F\}$.

We say that $L(M)$ is the *language accepted by M by final state and empty stack.*

In all cases, note that the input w must be consumed entirely.

The following lemma shows that the acceptance mode does not matter for PDA's. As we will see shortly, it does matter for DPDAs.

Lemma 2.14.4 *For any language L, the following facts hold.*

(1) If $L = T(M)$ for some PDA M, then $L = L(M')$ for some PDA M'.

(2) If $L = N(M)$ for some PDA M, then $L = L(M')$ for some PDA M'.

(3) If $L = L(M)$ for some PDA M, then $L = T(M')$ for some PDA M'.

(4) If $L = L(M)$ for some PDA M, then $L = N(M')$ for some PDA M'.

In view of lemma 2.14.4, the three acceptance modes T, N, L are equivalent.
The following PDA accepts the language

$$L = \{a^n b^n \mid n \geq 1\}$$

by empty stack.

$Q = \{1, 2\}$, $\Gamma = \{Z_0, a\}$;

$(1, a) \in \delta(1, a, Z_0)$,

$(1, aa) \in \delta(1, a, a)$,

$(2, \epsilon) \in \delta(1, b, a)$,

$(2, \epsilon) \in \delta(2, b, a)$.

The following PDA accepts the language

$$L = \{a^n b^n \mid n \geq 1\}$$

by final state and empty stack.

$Q = \{1, 2, 3\}$, $\Gamma = \{Z_0, A, a\}$, $F = \{3\}$;

$(1, A) \in \delta(1, a, Z_0)$,

$(1, aA) \in \delta(1, a, A)$,

$(1, aa) \in \delta(1, a, a)$,

$(2, \epsilon) \in \delta(1, b, a)$,

$(2, \epsilon) \in \delta(2, b, a)$,

$(3, \epsilon) \in \delta(1, b, A)$,

$(3, \epsilon) \in \delta(2, b, A)$.
DPDA's are defined as follows.

Definition 2.14.5 A PDA

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

is a *deterministic PDA* (for short, DPDA), iff the following conditions hold for all $(p, Z) \in Q \times \Gamma$: either

(1) $|\delta(p, a, Z)| = 1$ for all $a \in \Sigma$, and $\delta(p, \epsilon, Z) = \emptyset$,

or

(2) $\delta(p, a, Z) = \emptyset$ for all $a \in \Sigma$, and $|\delta(p, \epsilon, Z)| = 1$.

A DPDA *operates in realtime* iff it has no ϵ-transitions.

It turns out that for DPDA's the most general acceptance mode is by final state. Indeed, there are language that can only be accepted deterministically as $T(M)$. The language

$$L = \{a^m b^n \mid m \geq n \geq 1\}$$

is such an example. The problem is that $a^m b$ is a prefix of all strings $a^m b^n$, with $m \geq n \geq 2$.
A language L is a *deterministic context-free language* iff $L = T(M)$ for some DPDA M.

A PDA is *unambiguous* iff for every $w \in \Sigma^*$, there is at most one computation

$$(q_0, w, Z_0) \vdash^* ID_n,$$

where ID_n is an accepting ID.

We now show that every context-free language is accepted by a PDA.

2.15 From Context-Free Grammars To PDA's

We show how a PDA can be easily constructed from a context-free grammar. Although simple, the construction is not practical for parsing purposes, since the resulting PDA is horribly nondeterministic.

Given a context-free grammar $G = (V, \Sigma, P, S)$, we define a one-state PDA M as follows:

$Q = \{q_0\}$; $\Gamma = V$; $q_0 = S$; $Z_0 = S$; $F = \emptyset$;

For every rule $(A \rightarrow \alpha) \in P$, there is a transition

$$(q_0, \alpha) \in \delta(q_0, \epsilon, A).$$

For every $a \in \Sigma$, there is a transition

$$(q_0, \epsilon) \in \delta(q_0, a, a).$$

The intuition is that a computation of M mimics a leftmost derivation in G. One might say that we have a "pop/expand" PDA.

Lemma 2.15.1 *Given any context-free grammar*
$G = (V, \Sigma, P, S)$*, the PDA* M *just described accepts* $L(G)$ *by empty stack, i.e.,* $L(G) = N(M)$.

Proof. The following two claims are proved by induction.

Claim 1:

For all $u, v \in \Sigma^*$ and all $\alpha \in NV^* \cup \{\epsilon\}$, if $S \underset{lm}{\overset{*}{\Longrightarrow}} u\alpha$, then

$$(q_0, uv, S) \vdash^* (q_0, v, \alpha).$$

Claim 2:

For all $u, v \in \Sigma^*$ and all $\alpha \in V^*$, if

$$(q_0, uv, S) \vdash^* (q_0, v, \alpha)$$

then $S \underset{lm}{\overset{*}{\Longrightarrow}} u\alpha$. □

We now show how a PDA can be converted to a context-free grammar

2.16 From PDA's To Context-Free Grammars

The construction of a context-free grammar from a PDA is not really difficult, but it is quite messy. The construction is simplified if we first convert a PDA to an equivalent PDA such that for every move $(q, \gamma) \in \delta(p, a, Z)$ (where $a \in \Sigma \cup \{\epsilon\}$), we have $|\gamma| \leq 2$. In some sense, we form a kind of PDA in Chomsky Normal Form.

Lemma 2.16.1 *Given any PDA*

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F),$$

another PDA

$$M = (Q', \Sigma, \Gamma', \delta', q_0', Z_0', F')$$

can be constructed, such that $L(M) = L(M')$ and the following conditions hold:

(1) There is a one-to-one correspondence between accepting computations of M and M';

(2) If M has no ϵ-moves, then M' has no ϵ-moves; If M is unambiguous, then M' is unambiguous;

(3) For all $p \in Q'$, all $a \in \Sigma \cup \{\epsilon\}$, and all $Z \in \Gamma'$, if $(q, \gamma) \in \delta'(p, a, Z)$, then $q \neq q_0'$ and $|\gamma| \leq 2$.

The crucial point of the construction is that accepting computations of a PDA accepting by empty stack and final state can be decomposed into subcomputations of the form

$$(p, uv, Z\alpha) \vdash^* (q, v, \alpha),$$

where for every intermediate ID (s, w, β), we have $\beta = \gamma\alpha$ for some $\gamma \neq \epsilon$.

The nonterminals of the grammar constructed from the PDA M are triples of the form $[p, Z, q]$ such that

$$(p, u, Z) \vdash^+ (q, \epsilon, \epsilon)$$

for some $u \in \Sigma^*$.

Given a PDA

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

satisfying the conditions of lemma 2.16.1, we construct a context-free grammar $G = (V, \Sigma, P, S)$ as follows:

$$V = \{[p, Z, q] \mid p, q \in Q, Z \in \Gamma\} \cup \Sigma \cup \{S\},$$

where S is a new symbol, and the productions are defined as follows: for all $p, q \in Q$, all $a \in \Sigma \cup \{\epsilon\}$, all $X, Y, Z \in \Gamma$, we have:

(1) $S \to \epsilon \in P$, if $q_0 \in F$;

(2) $S \to a \in P$, if $(f, \epsilon) \in \delta(q_0, a, Z_0)$, and $f \in F$;

(3) $S \to a[p, X, f] \in P$, for every $f \in F$, if $(p, X) \in \delta(q_0, a, Z_0)$;

(4) $S \to a[p, X, s][s, Y, f] \in P$, for every $f \in F$, for every $s \in Q$, if $(p, XY) \in \delta(q_0, a, Z_0)$;

(5) $[p, Z, q] \to a \in P$, if $(q, \epsilon) \in \delta(p, a, Z)$ and $p \neq q_0$;

(6) $[p, Z, s] \to a[q, X, s] \in P$, for every $s \in Q$, if $(q, X) \in \delta(p, a, Z)$ and $p \neq q_0$;

(7) $[p, Z, t] \to a[q, X, s][s, Y, t] \in P$, for every $s, t \in Q$, if $(q, XY) \in \delta(p, a, Z)$ and $p \neq q_0$.

Lemma 2.16.2 *Given any PDA*

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

satisfying the conditions of lemma 2.16.1, the context-free grammar $G = (V, \Sigma, P, S)$ *constructed as above generates* $L(M)$*, i.e.,* $L(G) = L(M)$*. Furthermore,* G *is unambiguous iff* M *is unambiguous.*

Proof. We have to prove that

$$L(G) = \{w \in \Sigma^+ \mid (q_0, w, Z_0) \vdash^+ (f, \epsilon, \epsilon),\ f \in F\}$$
$$\cup \{\epsilon \mid q_0 \in F\}.$$

For this, the following claim is proved by induction.

Claim:

For all $p, q \in Q$, all $Z \in \Gamma$, all $k \geq 1$, and all $w \in \Sigma^*$,

$$[p, Z, q] \underset{lm}{\overset{k}{\Longrightarrow}} w \quad \text{iff} \quad (p, w, Z) \vdash^+ (q, \epsilon, \epsilon).$$

Using the claim, it is possible to prove that $L(G) = L(M)$, □
In view of lemmas 2.16.1 and 2.16.2, the family of context-free languages is exactly the family of languages accepted by PDA's. It is harder to give a grammatical characterization of the deterministic context-free languages. One method is to use Knuth $LR(k)$-grammars.

Another characterization can be given in terms of *strict deterministic grammars* due to Harrison and Havel.

Chapter 3

Computability

The subject of computability is about defining mathematically what computation might be and then using those definitions for exploration of the subject. For example, before we talk about complexity of an algorithm it is probably a good idea to define first what our model of computation we are using.
When did people first start studying computability ? This is a fuzzy thing, since the subject of computability is intermixed with many different things. But what is clear is that the subject as we present it here came in to being in the 1930's due to the work of Church, Godel, Kleene, Post and Turing. At least one influence was that real computers were being developed. But the subject was also of interest to other types of mathematician of the time. In particular, a great deal of work was initiated by Hilbert in his 1900 address at the international congress of mathematicians. There Hilbert gave a list of 21 problems that he thought were important. The so-called 10th, was to find an algorithm that would check if any given set of polynomial equations (i.e. a Diophantine system) had solutions. This problem was not settled until 1971, when Matiyasevich, building on the work of Davis, Putnam and ??? showed that no such algorithm could exist. But the point is, that in 1900 there was no good definition of algorithm. Hilbert's 10th problem may be considered just one of the pressures that were building to have such a definition.
As for the 'fuzzy' aspect mentioned above, there are a number of things that can be said. As was mentioned above, in 1900 no one was sure what was really meant precisely by an algorithm or procedure. Despite this, in the 19th century when non-constructive methods were first introduced (noticed?), they were not 'accepted' by all mathematicians. Intuitively, by a non-constructive proof we mean a proof that doesn't give an algorithm that produces the required object, but just abstractly shows the existence of the object. This led to something of an argument among mathematicians, notably between Hilbert and Brouwer as to whether the non-constructive proofs were 'right'. This eventually led to a branch of mathematics known as constructive analysis. Thus, in constructive analysis, for example, if a real-valued function is to be shown to have a root, then an algorithm for generating the root must be given.
Looking further back, a notion that is analogous to computation is the greek idea of a constructible number. Here we are given a few tools, usually a ruler and compass, (i.e. presumably a means of drawing) and use them to draw figures. We then can ask what lengths may be constructed with these tools. Thus a number is be constructible if there was a finite sequence of operations with these tools resulting in a segment whose length was that number. With the ruler and compass it turns out that only real algebraic numbers that are expressible with nested square roots of rational numbers may be constructed. We mean this analogy of a construction of a number with straight edge and compass with the computation of a function as only a rough analogy to help guide the reader with familiar examples. But it brings up an interesting point. For two thousand years it was an open problem as to whether an angle could be tri-sected with a ruler and compass. But it was known to Archimedes that if other tools (quadric surfaces, essentially) were allowed, then the angle could be tri-sected. This sort of phenomenon was troublesome when in the 1920's people started making attempts at defining what computable should mean. Namely, what operations should be allowed? A reasonable seeming choice leads to the definition of the *primitive recursive functions.* Unfortunately, this

class of functions turns out to be a little too small for the study of the subject. Interestingly though most of the programs that are written today are essentially primitive recursive. One really has to look hard to find a computable function that is not primitive recursive.
Clearly, early mechanical devices for doing arithmetic, such as the one designed and built by Pascal in the 17th century, and the sophisticated machines of Babbage in the early 19th century may be considered as ancestors of the modern computer. But one should be aware that these machines, like modern computers, are finite state machines. The models that we will present for computation below are not finite state, i.e. they have an unbounded amount of memory. For studying the theoretical aspects of computation this is appropriate, but often leads to results that are impractical or irrelevant in real life computing. Nevertheless, it is a suitable foundation and a first step towards making a mathematical model of computing. If we want to worry about space and time limitations then we can impose them on our model and engage in what is know as complexity theory.
In order to define what we mean for something to be computable, it is good to first settle on what the 'something' should be. A reasonable seeming place would be functions from the natural numbers to the natural numbers. But clearly we would like to be able to compute things with several inputs, so we first settle on functions from n-tuples of natural numbers to the natural numbers.
We now define the Turing machine model, which computes functions

$$f: \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{m} \to \Sigma^*,$$

where $\Sigma = \{a_1, \ldots, a_N\}$ is some input alphabet. We only consider deterministic Turing machines.

A Turing machine also uses a *tape alphabet* Γ such that $\Sigma \subseteq \Gamma$. The tape alphabet contains a distinguished symbol $B \notin \Sigma$,called the *blank*. In the model that we will be considering, a Turing machine uses a single 'tape' to store information. This tape can be viewed as a string over Γ. The tape is used both as an input tape and a storage mechanism. Symbols on the tape can be overwritten and the tape can grow in either the left or right direction. There is a read/write head that is, any any given instant during the computation, pointing to a symbol on the tape. Unlike Pushdown automata or NFA's, the read/write head can move to the left or the right. We now give the formal definition:

Definition 3.0.3 A (deterministic) *Turing machine* (or *TM*) M is a sextuple $M = (Q, \Sigma, \Gamma, \{L, R\}, \delta, q_0)$, where

- Q is a finite set of *states*;
- Σ is a finite *input alphabet*;
- Γ is a finite *tape alphabet*, s.t. $\Sigma \subseteq \Gamma$, $Q \cap \Gamma = \emptyset$, and with blank $B \notin \Sigma$;
- $q_0 \in Q$ is the *start state* (or *initial state*);
- δ is the *transition function*, a (finite) set of quintuples

$$\delta \subseteq Q \times \Gamma \times \Gamma \times \{L, R\} \times K,$$

such that for all $(p, a) \in Q \times \Gamma$, there is at most one triple $(b, m, q) \in \Gamma \times \{L, R\} \times Q$ such that $(p, a, b, m, q) \in \delta$.

A quintuple $(p, a, b, m, q) \in \delta$ is called an *instruction*. It is also denoted as

$$p, a \to b, m, q.$$

The effect of an instruction is to switch from state p to state q, overwrite the symbol currently scanned a with b, and move the read/write head either left or right, according to m. There is a convenient way to represent an instruction with a diagram. If the instruction is $p, a \to b, m, q.$ as above then we draw the diagram
A Turing machine can be represented by drawing a circle for each state and then connecting the states as described above.

Example 3.0.4 *Consider the Turing machine*

3.1 Computations of Turing Machines

To explain how a Turing machine works, we describe its action on *Instantaneous descriptions*. We take advantage of the fact that $K \cap \Gamma = \emptyset$ to define instantaneous descriptions.

Definition 3.1.1 Given a Turing machine

$$M = (K, \Sigma, \Gamma, \{L, R\}, \delta, q_0),$$

an *instantaneous description* (for short an *ID*) is a (nonempty) string in $\Gamma^* K \Gamma^+$, that is, a string of the form

$$upav,$$

where $u, v \in \Gamma^*$, $p \in K$, and $a \in \Gamma$.

The intuition is that an ID $upav$ describes a snapshot of a TM in the current state p, whose tape contains the string uav, and with the read/write head pointing to the symbol a.

Thus, in $upav$, the state p is just to the left of the symbol presently scanned by the read/write head.

We explain how a TM works by showing how it acts on ID's.

Definition 3.1.2 Given a Turing machine

$$M = (K, \Sigma, \Gamma, \{L, R\}, \delta, q_0),$$

the *yield relation (or compute relation)* $\vdash$ is a binary relation defined on the set of ID's as follows. For any two ID's ID_1 and ID_2, we have $ID_1 \vdash ID_2$ iff either

1. $(p, a, b, R, q) \in \delta$, and either

(a) $ID_1 = upacv$, $c \in \Gamma$, and $ID_2 = ubqcv$, or

(b) $ID_1 = upa$ and $ID_2 = ubqB$;

or

2. $(p, a, b, L, q) \in \delta$, and either

(a) $ID_1 = ucpav$, $c \in \Gamma$, and $ID_2 = uqcbv$, or

(b) $ID_1 = pav$ and $ID_2 = qBbv$.

Note how the tape is extended by one blank after the rightmost symbol in case (1)(b), and by one blank before the leftmost symbol in case (2)(b).

As usual, we let $\vdash^+$ denote the transitive closure of $\vdash$, and we let $\vdash^*$ denote the reflexive and transitive closure of $\vdash$.

We can now explain how a Turing function computes a partial function

$$f\colon \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{m} \to \Sigma^*.$$

Since we allow functions taking $m \geq 1$ input strings, we assume that Γ contains the special delimiter **,** not in Σ, used to separate the various input strings.

It is convenient to assume that a Turing machine "cleans up" its tape when it halts, before returning its output. For this, we will define proper ID's. We also define starting and blocking ID's.

Definition 3.1.3 Given a Turing machine

$$M = (K, \Sigma, \Gamma, \{L, R\}, \delta, q_0),$$

where Γ contains some delimiter **,** not in Σ in addition to the blank B, a *starting ID* is of the form

$$q_0 w_1{,}w_2, \ldots, w_m$$

where $w_1, \ldots, w_m \in \Sigma^*$ and $m \geq 2$, or $q_0 w$ with $w \in \Sigma^+$, or $q_0 B$.

A *blocking (or halting) ID* is an ID $upav$ such that there are no instructions $(p, a, b, m, q) \in \delta$ for any $(b, m, q) \in \Gamma \times \{L, R\} \times K$.

A *proper ID* is a halting ID of the form

$$B^k p w B^l,$$

where $w \in \Sigma^*$, and $k, l \geq 0$ (with $l \geq 1$ when $w = \epsilon$).

Computation sequences are defined as follows.

Definition 3.1.4 Given a Turing machine

$$M = (K, \Sigma, \Gamma, \{L, R\}, \delta, q_0),$$

a *computation sequence (or computation)* is a finite or infinite sequence of ID's

$$ID_0, ID_1, \ldots, ID_i, ID_{i+1}, \ldots,$$

such that $ID_i \vdash ID_{i+1}$ for all $i \geq 0$.

A computation sequence *halts* iff it is a finite sequence of ID's, so that

$$ID_0 \vdash^* ID_n,$$

and ID_n is a halting ID.

A computation sequence *diverges* if it is an infinite sequence of ID's.

We now explain how a Turing machine computes a partial function.

Definition 3.1.5 A Turing machine

$$M = (K, \Sigma, \Gamma, \{L, R\}, \delta, q_0)$$

computes the partial function

$$f{:}\underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{m} \to \Sigma^*$$

iff the following conditions hold:

1. For every $w_1, \ldots, w_m \in \Sigma^*$, given the starting ID

$$ID_0 = q_0 w_1{,}w_2, \ldots, w_m$$

or $q_0 w$ with $w \in \Sigma^+$, or $q_0 B$, the computation sequence of M from ID_0 halts in a proper ID iff $f(w_1, \ldots, w_m)$ is defined.

2. If $f(w_1, \ldots, w_m)$ is defined, then M halts in a proper ID of the form

$$ID_n = B^k p f(w_1, \ldots, w_m) B^h,$$

which means that it computes the right value.

A function f (over Σ^*) is *Turing computable* iff it is computed by some Turing machine M.

Note that by (1), the TM M may halt in an improper ID, in which case $f(w_1, \ldots, w_m)$ must be undefined. This corresponds to the fact that we only accept to retrieve the output of a computation if the TM has cleaned up its tape, i.e., produced a proper ID. In particular, intermediate calculations have to be erased before halting.
Example.

$K = \{q_0, q_1, q_2, q_3\}$;
$\Sigma = \{a, b\}$;
$\Gamma = \{a, b, B\}$;
The instructions in δ are:

$$
\begin{aligned}
q_0, B &\to B, R, q_3, \\
q_0, a &\to b, R, q_1, \\
q_0, b &\to a, R, q_1, \\
q_1, a &\to b, R, q_1, \\
q_1, b &\to a, R, q_1, \\
q_1, B &\to B, L, q_2, \\
q_2, a &\to a, L, q_2, \\
q_2, b &\to b, L, q_2, \\
q_2, B &\to B, R, q_3.
\end{aligned}
$$

The reader can easily verify that this machine exchanges the a's and b's in a string. For example, on input $w = aaababb$, the output is $bbbabaa$.

It turns out that the class of Turing computable functions coincides with the class of *partial recursive functions*. First, we define the *primitive recursive functions*.

3.2 The Primitive Recursive Functions

The primitive recursive functions were first formally defined by Goedel in 1929, although they were known to Hilbert. Like finite state automata, they are a simplified model of computation. But unlike regular languages, most functions (languages) that we encounter are primitive recursive. One has to look around a bit to find a function that is computable but not primitive recursive. And once such a function is found, it is usually not an easy task to show that it is not primitive recursive and it will be a very hard function to compute.
It seems reasonable that if two functions are computable then their composition should also be. Likewise for recursion. Also there are some 'primitive' functions that we definitely want to consider computable : projection functions, the zero function, and the successor function. These are called the *initial* or *base functions*. Given these, we may use the above two operations to generate a class of functions closed under those operations. This class is known as the class of *primitive recursive functions.* To continue with the ruler and compass analogy, the rational lengths are like the initial functions and the operations of composition and recursion are like the ruler and compass. We now begin the formal definitions:

Definition 3.2.1 Let $\Sigma = \{a_1, \ldots, a_N\}$. The *base functions* over Σ are the following functions:

1. The *erase function* E, defined such that $E(w) = \epsilon$, for all $w \in \Sigma^*$;
2. For every j, $1 \leq j \leq N$, the *j-successor function* S_j, defined such that $S_j(w) = wa_j$, for all $w \in \Sigma^*$;

3. The *projection functions* P_i^n, defined such that

$$P_i^n(w_1, \ldots, w_n) = w_i,$$

for every $n \geq 1$, every i, $1 \leq i \leq n$, and for all $w_1, \ldots, w_n \in \Sigma^*$.

Note that P_1^1 is the identity function on Σ^*. Projection functions can be used to permute the arguments of another function.

A crucial closure operation is (extended) composition.

Definition 3.2.2 Let $\Sigma = \{a_1, \ldots, a_N\}$. For any function

$$g: \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{m} \to \Sigma^*,$$

and any m functions

$$h_i: \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{n} \to \Sigma^*,$$

the *composition of g and the h_i* is the function

$$f: \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{n} \to \Sigma^*,$$

denoted as $g \circ (h_1, \ldots, h_m)$, such that

$$f(w_1, \ldots, w_n) = g(h_1(w_1, \ldots, w_n), \ldots, h_m(w_1, \ldots, w_n)),$$

for all $w_1, \ldots, w_n \in \Sigma^*$.

As an example, $f = g \circ (P_2^2, P_1^2)$ is such that

$$f(w_1, w_2) = g(w_2, w_1).$$

Another crucial closure operation is primitive recursion.

Definition 3.2.3 Let $\Sigma = \{a_1, \ldots, a_N\}$. For any function

$$g: \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{m-1} \to \Sigma^*,$$

where $m \geq 2$, and any N functions

$$h_i: \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{m+1} \to \Sigma^*,$$

the function

$$f: \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{m} \to \Sigma^*,$$

is defined by *primitive recursion from g and $h_1, \ldots, h_N$*, if

$$\begin{aligned}
f(\epsilon, w_2, \ldots, w_m) &= g(w_2, \ldots, w_m), \\
f(ua_1, w_2, \ldots, w_m) &= h_1(u, f(u, w_2, \ldots, w_m), w_2, \ldots, w_m), \\
\ldots &= \ldots \\
f(ua_N, w_2, \ldots, w_m) &= h_N(u, f(u, w_2, \ldots, w_m), w_2, \ldots, w_m),
\end{aligned}$$

for all $u, w_2, \ldots, w_m \in \Sigma^*$. When $m = 1$, for some fixed $w \in \Sigma^*$, we have

$$\begin{aligned} f(\epsilon) &= w, \\ f(ua_1) &= h_1(u, f(u)), \\ \ldots &= \ldots \\ f(ua_N) &= h_N(u, f(u)), \end{aligned}$$

for all $u \in \Sigma^*$.

As an example, the following function $g\colon \Sigma^* \times \Sigma^* \to \Sigma^*$, is defined by primitive recursion:

$$\begin{aligned} g(\epsilon, v) &= P_1^1(v), \\ g(ua_i, v) &= S_i(P_2^3(u, g(u, v), v)), \end{aligned}$$

where $1 \leq i \leq N$. It is easily verified that $g(u, v) = vu$. Then,

$$f = g \circ (P_2^2, P_1^2)$$

computes the concatenation function, i.e. $f(u, v) = uv$.

Definition 3.2.4 Let $\Sigma = \{a_1, \ldots, a_N\}$. The class of *primitive recursive functions* is the smallest class of functions (over Σ^*) which contains the base functions and is closed under composition and primitive recursion.

We leave as an exercise to show that every primitive function is a total function. The class of primitive recursive functions may not seem very big, but it contains just about any total function that anyone would ever want to compute.

Although it is rather tedious to prove, the following theorem can be shown.

Theorem 3.2.5 *For an alphabet $\Sigma = \{a_1, \ldots, a_N\}$, every primitive recursive function is Turing computable.*

The best way to prove the above theorem is probably to introduce yet another model of computation, *RAM programs*. Then, it can be shown that every Turing machine can simulate a RAM program. It is also rather easy to show that the RAM-computable functions contain the primitive recursive functions.
It is very instructive to consider some examples of the primitive recursive functions over the natural numbers, the way they were originally defined. To do this within the definitions given above we simply let $\Sigma = \{a\}$. Then we identify the sting a^n with the natural number n. Thus the erase function becomes the zero function

$$E(x) = 0$$

the successor becomes addition by 1

$$S(x) = x + 1,$$

and the projections become the usual projections on the natural numbers

$$P_i^n(x_1, ..., x_n) = x_i.$$

This doesn't look like much to work with. But the amazing thing is that very complex functions can be made out of them.

Example 3.2.6 As a first example let's work backwards. Consider the factorial function, $f(n) = n!$, which is given by the equations

$$f(0) = 0,$$
$$f(n+1) = nf(n).$$

How do we show that this function is primitive recursive ? We must demonstrate that we can construct it within the above formalism. Beware - it is not automatic that this function is primitive recursive. For example, the defining equations contain a multiplication. But we did show that concatenation was primitive recursive. What does this translate into over the natural numbers ? Addition, i.e. we now know that the function

$$+(x, y) = x + y$$

is primitive recursive. We want to show that the multiplication function, m(x, y) =xy, is primitive recursive. We may define m by the equations

$$m(x, 0) = x,$$
$$m(x+1, y) = +(m(x, y), y).$$

This is still technically not enough, since we need to exhibit two the functions g and h used in the definition of primitive recursive. But it is not hard to see that we may take $g = P_1^2$ and $h = + \circ (P_2^3, P_3^3)$.

So far, the primitive recursive functions do not yield all the Turing-computable functions. In order to get a larger class of functions, we need the closure operation known as minimization.

3.3 The Partial Recursive Functions

The operation of minimization (sometimes called minimalization) is defined as follows.

Definition 3.3.1 Let $\Sigma = \{a_1, \ldots, a_N\}$. For any function

$$g: \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{m+1} \to \Sigma^*,$$

where $m \geq 0$, for every j, $1 \leq j \leq N$, the function

$$f: \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{m} \to \Sigma^*,$$

is defined by *minimization over* $\{a_j\}^*$ *from* g, if the following conditions hold for all $w_1, \ldots, w_m \in \Sigma^*$:
1. $f(w_1, \ldots, w_m)$ is defined iff there is some $n \geq 0$ such that
$g(a_j^p, w_1, \ldots, w_m)$ is defined for all p, $0 \leq p \leq n$, and

$$g(a_j^n, w_1, \ldots, w_m) = \epsilon.$$

2. When $f(w_1, \ldots, w_m)$ is defined,

$$f(w_1, \ldots, w_m) = a_j^n,$$

where n is such that

$$g(a_j^n, w_1, \ldots, w_m) = \epsilon$$

and

$$g(a_j^p, w_1, \ldots, w_m) \neq \epsilon$$

for every p, $0 \leq p \leq n-1$. The value $f(w_1, \ldots, w_m)$ is also denoted as

$$min_j u[g(u, w_1, \ldots, w_m) = \epsilon].$$

Note: When $f(w_1, \ldots, w_m)$ is defined,

$$f(w_1, \ldots, w_m) = a_j^n,$$

where n is the smallest integer such that condition (1) holds. It is very important to require that all the values $g(a_j^p, w_1, \ldots, w_m)$ be defined for all p, $0 \leq p \leq n$, when defining $f(w_1, \ldots, w_m)$. Failure to do so allows non-computable functions.

The class of partial computable functions is defined as follows.

Definition 3.3.2 Let $\Sigma = \{a_1, \ldots, a_N\}$. The class of *partial recursive functions* is the smallest class of functions (over Σ^*) which contains the base functions and is closed under composition, primitive recursion, and minimization. The class of *recursive functions* is the subset of the class of partial recursive functions consisting of functions defined for every input.

One of the major results of computability theory is the following theorem.

Theorem 3.3.3 *For an alphabet $\Sigma = \{a_1, \ldots, a_N\}$, every partial recursive function is Turing-computable. Conversely, every Turing-computable function is a partial recursive function. Similarly, the class of recursive functions is equal to the class of Turing-computable functions that halt in a proper ID for every input.*

To prove that every partial recursive function is indeed Turing-computable, the simplest thing to do is to show that every partial recursive function is RAM-computable. For the converse, one can show that given a Turing machine, there is a primitive recursive function describing how to go from one ID to the next. Then, minimization is used to guess whether a computation halts. The proof shows that every partial recursive function needs minimization at most once. The characterization of the recursive functions in terms of TM's follows easily.

We can also deal with languages.

3.4 Recursively Enumerable Languages and Recursive Languages

The idea behind a set being recursively enumerable can be expressed by the following example. Suppose that we are seeking a integer solution to the equation

$$x^2 + y^2 = z^2 + z^4.$$

We may try to find a solution to this equation by enumerating all integer triples and then plugging them into the equation one by one, each time checking to see if the triple is a solution. As we go along, if we get a triple that is a solution then we are done, otherwise we forget about it and move on to the next triple. This is fine except that, if there is no solution we will go on forever plugging in triples to the equation, and hence be caught in an infinite loop. But we may get lucky and find that there is a solution.
Now let's generalize a little. Suppose that we wanted to write a program that would take as input a system of polynomial equations in several variables as input and determine if the system had a solution in the integers. Applying the above scheme to each equation seems like a good idea. But there is a problem: if the system of equations given to the program has no solution we get caught in an infinite loop. So if our program returns an answer, it will be 'yes, the system has a solution.' But otherwise we have to sit and wait, and there is no way of telling if the wait is going to be forever!
This is one way of looking at recursively enumerable sets. The other way, which is equivalent, is that a recursively enumerable set is the simply the range of a recursive function. That is where the name comes from.
In the definition that follows we assume that the TM's under consideration have a tape alphabet. containing the special symbols 0 and 1.

Definition 3.4.1 Let $\Sigma = \{a_1, \ldots, a_N\}$. A language $L \subseteq \Sigma^*$ is *recursively enumerable (for short, an r.e. set)* iff there is some TM M such that for every $w \in L$, M halts in a proper ID with the output 1, and for every $w \notin L$, either M halts in a proper ID with the output 0, or it runs forever. A language $L \subseteq \Sigma^*$ is *recursive* iff there is some TM M such that for every $w \in L$, M halts in a proper ID with the output 1, and for every $w \notin L$, M halts in a proper ID with the output 0.

Thus, given a recursively enumerable language L, for some $w \notin L$, it is possible that a TM accepting L runs forever on input w. On the other hand, for a recursive language L, a TM accepting L always halts in a proper ID.
When dealing with languages, it is often useful to consider *nondeterministic Turing machines*. Such machines are defined just like deterministic Turing machines, except that their transition function δ is just a (finite) set of quintuples

$$\delta \subseteq K \times \Gamma \times \Gamma \times \{L, R\} \times K,$$

with no particular extra condition.

It can be shown that every nondeterministic Turing machine can be simulated by a deterministic Turing machine, and thus, nondeterministic Turing machines also accept the class of r.e. sets.

It can be shown that a recursively enumerable language is the range of some recursive function. It can also be shown that a language L is recursive iff both L and its complement are recursively enumerable. There are recursively enumerable languages that are not recursive.
Turing machines were invented by Turing around 1935. The primitive recursive functions were known to Hilbert circa 1890. Gödel formalized their definition in 1929. The partial recursive functions were defined by Kleene around 1934. Church also introduced the λ-calculus as a model of computation around 1934. Other models: Post systems, Markov systems. The equivalence of the various models of computation was shown around 1935/36. RAM programs were only defined around 1963.

A further study of the partial recursive functions requires the notions of pairing functions and of universal functions (or universal Turing machines).

3.5 Phrase-Structure Grammars

Context-free grammars can be generalized in various ways. The most general grammars generate exactly the recursively enumerable languages. Between the context-free languages and the recursively enumerable languages, there is a natural class of languages, the context-sensitive languages. The context-sensitive languages also have a Turing-machine characterization. We begin with phrase-structure grammars.

Definition 3.5.1 A *phrase-structure grammar* is a
quadruple $G = (V, \Sigma, P, S)$, where

- V is a finite set of symbols called the *vocabulary (or set of grammar symbols)*;
- $\Sigma \subseteq V$ is the set of *terminal symbols (for short, terminals)*;
- $S \in (V - \Sigma)$ is a designated symbol called the *start symbol*;

The set $N = V - \Sigma$ is called the set of *nonterminal symbols (for short, nonterminals)*.

- $P \subseteq V^*NV^* \times V^*$ is a finite set of *productions (or rewrite rules, or rules)*. Every production $\langle \alpha, \beta \rangle$ is also denoted as $\alpha \rightarrow \beta$. A production of the form $\alpha \rightarrow \epsilon$ is called an *epsilon rule, or null rule*.

Example 1.

$$G_1 = (\{S, A, B, C, D, E, a, b\}, \{a, b\}, S, P),$$

where P is the set of rules

$$\begin{aligned}
S &\longrightarrow ABC,\\
AB &\longrightarrow aAD,\\
AB &\longrightarrow bAE,\\
DC &\longrightarrow BaC,\\
EC &\longrightarrow BbC,\\
Da &\longrightarrow aD,\\
Db &\longrightarrow bD,\\
Ea &\longrightarrow aE,\\
Eb &\longrightarrow bE,\\
AB &\longrightarrow \epsilon,\\
C &\longrightarrow \epsilon,\\
aB &\longrightarrow Ba,\\
bB &\longrightarrow Bb.
\end{aligned}$$

It can be shown that this grammar generates the language

$$L = \{ww \mid w \in \{a, b\}^*\},$$

which is not context-free.

3.6 Derivations and Type-0 Languages

The productions of a grammar are used to derive strings. In this process, the productions are used as rewrite rules.

Definition 3.6.1 Given a phrase-structure grammar $G = (V, \Sigma, P, S)$, the (one-step) *derivation relation* $\Longrightarrow_G$ *associated with* G is the binary relation $\Longrightarrow_G \subseteq V^* \times V^*$ defined as follows: for all $\alpha, \beta \in V^*$, we have

$$\alpha \Longrightarrow_G \beta$$

iff there exist $\lambda, \rho \in V^*$, and some production $(\gamma \to \delta) \in P$, such that

$$\alpha = \lambda\gamma\rho \quad \text{and} \quad \beta = \lambda\delta\rho.$$

The transitive closure of $\Longrightarrow_G$ is denoted as $\overset{+}{\Longrightarrow}_G$ and the reflexive and transitive closure of $\Longrightarrow_G$ is denoted as $\overset{*}{\Longrightarrow}_G$.

When the grammar G is clear from the context, we usually omit the subscript G in $\Longrightarrow_G$, $\overset{+}{\Longrightarrow}_G$, and $\overset{*}{\Longrightarrow}_G$. The language generated by a phrase-structure grammar is defined as follows.

Definition 3.6.2 Given a phrase-structure grammar $G = (V, \Sigma, P, S)$, the *language generated by* G is the set

$$L(G) = \{w \in \Sigma^* \mid S \overset{+}{\Longrightarrow} w\}.$$

A language $L \subseteq \Sigma^*$ is a *type*-0 *language* iff $L = L(G)$ for some phrase-structure grammar G.

The following lemma can be shown.

Lemma 3.6.3 *A language L is recursively enumerable iff it generated by some phrase-structure grammar G.*

In one direction, we can construct a nondeterministic Turing machine simulating the derivations of the grammar G. In the other direction, we construct a grammar simulating the computations of a Turing machine.

We now consider some variants of the phrase-structure grammars.

3.7 Type-0 Grammars and Context-Sensitive Grammars

We begin with type-0 grammars. At first glance, it may appear that they are more restrictive than phrase-structure grammars, but this is not so.

Definition 3.7.1 A *type-0 grammar* is a phrase-structure grammar $G = (V, \Sigma, P, S)$, such that the productions are of the form

$$\alpha \to \beta,$$

where $\alpha \in N^+$. A production of the form $\alpha \to \epsilon$ is called an *epsilon rule, or null rule.*

Lemma 3.7.2 *A language L is generated by a phrase-structure grammar iff it is generated by some type-0 grammar.*

We now place additional restrictions on productions, obtaining context-sensitive grammars.

Definition 3.7.3 A *context-sensitive grammar*
(for short, csg) is a phrase-structure grammar $G = (V, \Sigma, P, S)$, such that the productions are of the form

$$\alpha A \beta \to \alpha \gamma \beta,$$

with $A \in N$, $\gamma \in V^+$, $\alpha, \beta \in V^*$, or

$$S \to \epsilon,$$

and if $S \to \epsilon \in P$, then S does not appear on the right-hand side of any production.

The notion of derivation is defined as before. A language L is *context-sensitive* iff it is generated by some context-sensitive grammar.

We can also define monotonic grammars.

Definition 3.7.4 A *monotonic grammar* is a phrase-
structure grammar $G = (V, \Sigma, P, S)$, such that the productions are of the form

$$\alpha \to \beta$$

with $\alpha, \beta \in V^+$ and $|\alpha| \leq |\beta|$, or

$$S \to \epsilon,$$

and if $S \to \epsilon \in P$, then S does not appear on the right-hand side of any production.

Example 2.

$$G_2 = (\{S, A, B, C, a, b, c\}, \{a, b, c\}, S, P),$$

where P is the set of rules

$$\begin{aligned} S &\longrightarrow ABC, \\ S &\longrightarrow ABCS, \\ AB &\longrightarrow BA, \\ AC &\longrightarrow CA, \\ BC &\longrightarrow CB, \\ BA &\longrightarrow AB, \\ CA &\longrightarrow AC, \\ CB &\longrightarrow BC, \\ A &\longrightarrow a, \\ B &\longrightarrow b, \\ C &\longrightarrow c. \end{aligned}$$

It can be shown that this grammar generates the language

$$L = \{w \in \{a, b, c\}^+ \mid \#(a) = \#(b) = \#(c)\},$$

which is not context-free.

Lemma 3.7.5 *A language L is generated by a context-sensitive grammar iff it is generated by some monotonic grammar.*

The context-sensitive languages are accepted by space-bounded Turing machines, defined as follows.

Definition 3.7.6 A *linear-bounded automaton*
(for short, lba) is a nondeterministic Turing machine such that for every input $w \in \Sigma^*$, there is some accepting computation in which at most $|w| + 1$ tape symbols are scanned.

Lemma 3.7.7 *A language L is generated by a context-sensitive grammar iff it is accepted by a linear-bounded automaton.*

The class of context-sensitive languages is very large. The main problem is that no practical methods for constructing parsers from csg's are known.

3.8 The Halting Problem

3.9 A Univeral Machine

3.10 The Parameter Theorem

3.11 Recursively Enumerable Languages

3.12 Hilbert's Tenth Problem

Chapter 4

Current Topics