Basic Formal Language Theory

James Rogers
jrogers@cs.ucf.edu

PART I

Basic Concepts

Part I
Basic Concepts

1 Computation and Languages

This tutorial is intended to refresh your understanding of the topics covered in a typical undergraduate level Formal Languages class (at UCF, COT4210). While the presentation occasionally assumes some familiarity with these topics, we have attempted to explicitly define every notion we use; hence, it should be accessible even for those without that background. For the most part, we follow the notation and conventions of Hopcroft and Ullman (on reserve in the library). The first six chapters of that text provide a slightly different perspective on the material as well as a wealth of additional exercises.

This tutorial covers the basic aspects of *Formal Language Theory*—the study of the mathematical properties of *abstract languages*, in which the elements of the language are sequences of arbitrary symbols rather than the sequences of words, letters, or characters that make up most (written) natural languages. Nonetheless, our focus will be directed to the *Theory of Computation*—the study of whether problems can be solved algorithmically and how difficult it is to do so. The first question we have to address, then, is how these two things are related.

When we study computability we are studying *problems* in an abstract sense. For example, *addition* is the problem of, having been given two numbers, returning a third number that is their sum. Two problems of particular interest in Computer Science, which you have probably encountered previously, are the *Traveling Salesperson Problem* (TSP) and the *Halting Problem*. In TSP one is given a list of distances between some number of cities and is asked to find the shortest route that visits each city once and returns to the start. In the Halting Problem, one is given a program and some appropriate input and asked to decide whether the program, when run on that input, loops forever or halts. Note that, in each of the cases the statement of the problem doesn't give us the actual values we need to provide the result for, but rather just tells us what kind of objects they are. A set of actual values for a problem is called an *instance* of the problem. (So, in this terminology, all the homework problems you did throughout school were not problems but were, rather, instances of problems.)

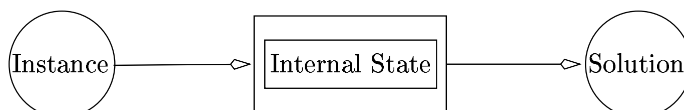
A problem, then, specifies what an instance is, i.e., what the input is, and how the solution, or output, must be related to the that input. There are a number of things one might seek to know about a problem, among them:

- Can it be solved *algorithmically*; is there a definite procedure that solves any instance of the problem in a finite amount of time? In other words, is it *computable*. Not all problems are computable; the halting problem is the classic example of one that is not.

- How hard is it to solve? What kind of resources are needed and how much of those resources is required? Again, some problems are harder than others. TSP is an example of a frustrating class of problems that have no known efficient solution, but which have never been proven to be necessarily hard.

When we say "solved algorithmically" we are not asking about a specific programming language, in fact one of the theorems in computability is that essentially all reasonable programming languages are equivalent in their power. Rather, we want to know if there is an algorithm for solving it that can be expressed in any rigorous way at all. Similarly, we are not asking about whether the problem can be solved on any particular computer, but whether it can be solved by any computing mechanism, including a human using a pencil and paper (even a limitless supply of paper).

What we need is an *abstract model* of computation that we can treat in a rigorous mathematical way. We'll start with the obvious model:



Here a computer receives some input (an instance of a problem), has some computing mechanism, and produces some output (the solution of that instance). We will refer to the configuration of the computing mechanism at a given point in it's processing as its *internal state*. Note that in this model the computer is not a general purpose device: it solves some specific problem. Rather, we consider a general purpose computer and a program to both be part of a single machine. The program, in essence, specializes the computer to solve a particular problem.

We can simplify this somewhat by eliminating the output if a couple of mild assumptions hold:

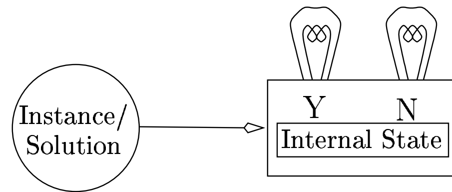
1. Every instance of the problem has a solution.
2. The solution to any instance of the problem is unique.
3. We can list all instances of the problem along with its possible solutions in a systematic way.

Note that there are likely to be infinitely many instances of a problem and infinitely many possible solutions for each of them. What we mean by listing them in a systematic way is that any given instance and possible solution will eventually be listed. Note also that every possible solution includes all the incorrect ones along with the correct one. (Otherwise we would essentially be assuming that the problem was computable.)

These assumptions hold for addition, for instance. Every instance of addition has a unique solution. Each instance is a pair of numbers and the possible solutions include any third number. We can systematically list all instances along with all possible solutions by systematically listing all triples of numbers. This is not completely trivial—we can't, for instance, list all triples starting with 0 and then all triples starting with 1, etc. Since there are infinitely many triples starting with zero, we would never get around to listing any starting with one. Suppose, though, that we are only concerned with the *Natural Numbers*, $\{0, 1, \ldots\}$. If we first list all triples that *sum* to zero (i.e., just the triple $\langle 0, 0, 0 \rangle$) and then all triples that sum to one (i.e., $\langle 1, 0, 0 \rangle$, $\langle 0, 1, 0 \rangle$, $\langle 0, 0, 1 \rangle$), etc., we are guaranteed that we will eventually list any given triple.

With the exception of the assumption that the solution is unique (which can be fudged in a variety of ways) these assumptions are pretty nearly minimal. We can't even consider solving a problem algorithmically unless every instance has a solution. An algorithm must produce some answer for every instance. If there is no answer for some instance, then whatever answer it produces will necessarily be wrong. (Note that if we modify the problem to require that we return "No Solution" in the case that none exists, we will have converted it into a problem that has a solution for every instance—albeit one that sometimes has the solution "No Solution".) The third assumption is true of every reasonable problem. In fact, it takes a fair amount of the theory of computation to even get to the point where we can argue that problems that don't satisfy the assumption might exist.

Under these assumptions we can reduce our model to a machine for checking the correctness of solutions:



This machine takes an instance of a problem along with a possible solution as its input and lights one lamp (Y) if the solution is correct and the other (N) if it is not. We will refer to an algorithm for the original model, in which we are given an instance and must produce a solution as an algorithm for *solving* the problem and we will refer to an algorithm for the second model as an algorithm for *checking* the problem.

This leads us to our first theorem.

Claim 1 *Under the assumptions above, if there is an algorithm for checking a problem then there is an algorithm for solving the problem.*

Before going on, you should think a bit about how to do this. For this claim the assumption that the solution of each instance is unique is not necessary; but both of the others are. If you had a program that checks whether a proposed solution to an instance of a problem is correct and another that systematically generates every instance of the problem along with every possible solution, how could you use them (as subroutines) to build a program that, when given an instance, was guaranteed to find a correct solution to that problem under the assumption that such a solution always exists?

Proof (Claim 1)**:** Suppose we had a subroutine that, each time we call it, returns the next instance/solution in the list. We can simply put this in a loop that gets the next instance/solution and checks to see if the instance listed matches the instance we are to solve. If it does, we use the checking program as a subroutine to tell us if the solution listed along with that instance is correct. If it is we are done, otherwise we reiterate the loop.

This gives us an effective procedure that we claim is an algorithm for solving the problem. There are two things we need to prove to verify that it is. First we must confirm that, for any input, it always produces a solution in a finite amount of time (in other words, the procedure always terminates). Then we must show that the solution it produces is, in fact, correct (in other words, the procedure solves the problem). The second part is nearly immediate. The procedure only produces a solution if the checking algorithm says it is correct. The correctness of the checking algorithm ensures the correctness of our solution. The first part is nearly as simple. Since there is a correct solution to every instance and every given combination of an instance and a possible solution will be listed eventually, we are certain that our given instance and its correct solution will show up in a finite amount of time. At that point, the checking algorithm says "Yes" and our procedure halts. $\dashv$

The converse of the claim is also true.

Claim 2 *Under the assumptions above, if there is an algorithm for solving a problem then there is an algorithm for checking it.*

The proof of this is your first exercise.

1. Prove Claim 2. Show how, using an algorithm to solve the problem as a subroutine, one can construct an algorithm for checking the problem. This claim depends only on the assumption that there is a unique solution for every instance (1 & 2), not on the assumption that instances and solutions can be enumerated (3).

Putting the claims together we get a theorem.

Theorem 1 *Under our assumptions, a problem can be solved algorithmically iff it can be checked algorithmically.*

In other words, the two models are equivalent computationally. It is possible to compute solutions if and only if it is possible to check them. Given this, we

will usually restrict attention to the simpler model, that is, we will usually be concerned with identifying *solved instances* of a problem.

Without loss of generality we can assume that instances and solutions are encoded as strings of symbols. Our task is to distinguish those strings that encode solved instances of the problem among all strings drawn from the same set of symbols. Such a distinguished set of strings is, of course, a language in the formal sense. One approach, then, to studying computability in general is to study how to define and recognize formal languages.

2 Formal Languages

The purpose of this section is to give precise mathematical meaning to the notions of string and language and the operations on them.

2.1 Alphabets, Strings and Languages

An *alphabet* is any finite set of symbols:

$$\text{e.g., } \Sigma = \{a, b, c\}.$$

We will generally use Σ (the Greek capital letter *sigma*) to denote our alphabet. When we need to distinguish two alphabets, the second will usually be denoted by Γ (the Greek capital letter *gamma*). Lower-case Greek letters will be used to denote arbitrary symbols in an alphabet (e.g., $\sigma \in \Sigma$, $\gamma \in \Gamma$.)

The set of all *strings* over Σ, denoted Σ^*, is the set of all sequences of symbols drawn from Σ.

$$\text{e.g., } w = abbac \in \{a, b, c\}^*.$$

We will usually employ lower case variables near w, v and u, sometimes primed (w') and sometimes with subscripts (w_1), to denote strings. Included in Σ^* is the *empty string* (the sequence of length zero), denoted here as ε. Other ways of denoting the empty string that you may encounter include λ and Λ (lower and upper case Greek letter *lambda*). We will always use ε.

We can formalize this in the following way:

Definition 1 (Strings over an alphabet Σ) *Given any alphabet Σ:*

- *The empty sequence is a string over Σ (i.e., $\varepsilon \in \Sigma^*$).*
- *If v is a string over Σ, σ is a symbol in Σ and $w = v \cdot \sigma$, then w is a string over Σ (i.e., $v \in \Sigma^*$, $\sigma \in \Sigma$ and $w = v \cdot \sigma$ implies $w \in \Sigma^*$.*
- *Nothing else is a string over Σ.*

This is an example of an *inductive definition* of a set: we supply a *basis*—the simplest cases of members of the set (here just ε), one or more *inductive clauses*—ways of building new members of the set out of simpler ones, and a *closure clause* a statement limiting the members of the set to those constructible from the basis by the operations of the inductive clauses. Every

string in Σ^*, then, can be understood to have been built from the empty string by finitely many concatenations of single symbols.

Note that '$\cdot$' denotes the operation of concatenation, just as '+' denotes the operation of addition. The result of the concatenation is just the juxtaposition of the two operands: $aba \cdot a = abaa$, just as $2 + 7 = 9$. While it is a slight abuse of notation, we will follow this even when the one or both of the strings are represented by variables. Thus the string denoted by $w \cdot \sigma$ will usually be expressed as $w\sigma$.

Inductive definitions of sets support *recursive* definitions of operations on those sets. The *length* of a string $w \in \Sigma^*$ is denoted $|w|$. We define this recursively as follows:

Definition 2 (Length of a string) *For all* $w \in \Sigma^*$*:*

$$|w| = \begin{cases} 0 & \textit{if } w = \varepsilon, \\ |v| + 1 & \textit{if } w = v\sigma. \end{cases}$$

Note that, given the definition above, every string is either ε or is constructed by concatenating some $\sigma \in \Sigma$ onto some $v \in \Sigma^*$ and so the definition need only treat these two cases. In the latter case, the definition *recurs*—the operation is defined in terms of the result of applying the operation to v. The fact that w is obtained from v by concatenation of σ and that all strings, by definition, are obtained from ε by a finite series of such concatenations, insures that the recursion is *well-founded*—all w will be reduced to ε in finitely many steps—thus the recursion is guaranteed to terminate.

In a similar way, we can extend the operation of concatenation to apply to pairs of strings (not just a string and a symbol).

Definition 3 (Concatenation of strings) *For all* $w, v \in \Sigma^*$*:*

$$w \cdot v = \begin{cases} w & \textit{if } v = \varepsilon, \\ (w \cdot u) \cdot \sigma & \textit{if } v = u\sigma. \end{cases}$$

A string consisting of a single symbol is a *unit string*. It is conventional to fail to distinguish a symbol from the unit string containing just that symbol. Thus when we say $a \cdot a = aa$, we do not need to worry about which, if either, 'a' is a symbol or a string.

Finally, we will denote the *reversal* of a string w as w^{R}. This is just w with the order of its symbols reversed, i.e.,

$$(aabac)^{\mathrm{R}} = cabaa \qquad (radar)^{\mathrm{R}} = radar \qquad a^{\mathrm{R}} = a$$

Definition 4 (Reversal of a string.) *For all $w \in \Sigma^*$:*

$$w^{\mathrm{R}} = \begin{cases} \varepsilon & \text{if } w = \varepsilon, \\ \sigma \cdot v^{\mathrm{R}} & \text{if } w = v\sigma. \end{cases}$$

Definition 5 (Language over an alphabet) *A* language*, L, over an alphabet, Σ, is any (not necessarily proper) subset of Σ^*: $L \subseteq \Sigma^*$.*

Note that among the subsets of Σ^* is the empty set $\emptyset$ which is just the language over Σ^* that contains no strings, usually referred to as the *empty language*. It is important not to confuse the empty string ε with the empty language $\emptyset$. They differ in type: the empty string is the sequence of no symbols, the empty language is the set of no strings.

2.2 Operations on Languages

As they are just sets, all of the usual operations on sets apply to languages. Pairwise *union* ($\cup$) and *intersection* ($\cap$) should be familiar.

Definition 6 (Union and intersection) *For $L_1, L_2 \subseteq \Sigma^*$:*

$$\begin{aligned} L_1 \cup L_2 &\stackrel{\text{def}}{=} \{w \mid w \in L_1 \text{ or } w \in L_2\} \\ L_1 \cap L_2 &\stackrel{\text{def}}{=} \{w \mid w \in L_1 \text{ and } w \in L_2\} \end{aligned}$$

The *relative complement*, or *set difference*, of two sets over the same alphabet is:

Definition 7 (Relative complement) *For $L_1, L_2 \subseteq \Sigma^*$:*

$$L_1 \setminus L_2 \stackrel{\text{def}}{=} \{w \mid w \in L_1 \text{ and } w \notin L_2\}$$

The *complement* of a language L over Σ^* (denoted $\overline{L}$) is just all the strings over Σ that are not in L.

Definition 8 (Complement of a language) *If L is a language over an alphabet Σ, then*

$$\overline{L} \stackrel{\text{def}}{=} \Sigma^* \setminus L.$$

The *cardinality* of a set is its size:

Definition 9 (Cardinality of a set) *If A is any* finite *set then* $\mathbf{card}(A)$ *is the number of elements in A. If A is not finite the we will say* $\mathbf{card}(A)$ *is infinite.*

(We will not distinguish between countably infinite sets and the larger infinite sets.)

The *Cartesian product* of two sets is the set of pairs drawn from them:

Definition 10 (Cartesian product) *If A and B are arbitrary sets:*

$$A \times B \stackrel{\text{def}}{=} \{\langle a, b\rangle \mid a \in A, b \in B\}.$$

The *powerset* of a set is the set of all its subsets:

Definition 11 (Powerset) *If A is any set:*

$$\mathcal{P}(A) \stackrel{\text{def}}{=} \{B \mid B \subseteq A\}.$$

The *characteristic function* of s set is a function that tests for membership in that set:

Definition 12 (Characteristic function of a set) *If A is any set and $B \subseteq A$, then $\chi_B : A \to \{0, 1\}$ is:*

$$\chi_B(x) \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } x \in B \\ 0 & \text{otherwise} \end{cases}.$$

We generalize pairwise union and intersection to allow union and intersection of possibly infinite *families* of languages. If $L_1, L_2, \ldots$ is such a family of languages, then their *infinite union* is denoted $\bigcup_i[L_i]$ and their *infinite intersection* is denoted $\bigcap_i[L_i]$. These are just the set of strings that are in any (union) or all (intersection) of the L_i.

Definition 13 (Infinite union, intersection) *If $L_1, L_2, \ldots$ is an infinite sequence of languages then*

$$\bigcup_{i \in \mathbb{N}}[L_i] \stackrel{\text{def}}{=} L_1 \cup L_2 \cup \cdots$$
$$\bigcap_{i \in \mathbb{N}}[L_i] \stackrel{\text{def}}{=} L_1 \cap L_2 \cap \cdots$$

Just as we extended concatenation to pairs of strings, we can extend it to pairs of languages:

Definition 14 (Concatenation of languages) *For any languages L_1 and L_2, (not necessarily over the same alphabet):*

$$L_1 \cdot L_2 = \{w \cdot v \mid w \in L_1 \text{ and } v \in L_2\}.$$

Which is to say, the concatenation of two languages is just the concatenation of all pairs of strings drawn from them.

If w is a string over some alphabet Σ (i.e., $w \in \Sigma^*$) and i is a natural number (i.e., $i \in \mathbb{N}$ where $\mathbb{N} = \{0, 1, 2, \ldots\}$), we will use the notation w^i to represent i copies of w concatenated together. So

$$\begin{aligned} (aaba)^3 &= aabaaabaaaba \\ a^7 &= aaaaaaa \end{aligned}$$

Definition 15 (Iteration of a Language) *If L is a language and $i \in \mathbb{N}$ then:*

$$L^i = \begin{cases} \{\varepsilon\} & \text{if } i = 0, \\ L^j \cdot L & \text{if } i = j + 1. \end{cases}$$

Definition 16 (Closure of a Language) *If L is a language L^* is its* Kleene closure*:*

$$L^* = \bigcup_{i \geq 0} [L^i].$$

L^+ is the positive closure *of L:*

$$L^+ = \bigcup_{i \geq 1} [L^i].$$

This leads the the second exercise.

2. Is L^* ever empty? What about L^+? Under what circumstances does L^+ contain ε? (Consider the cases: $L = \emptyset$ and $L = \{\varepsilon\}$.)

3 An Informal Preview

We are going to focus on two related issues: how to define languages and how to specify algorithms for *recognizing* them—for identifying the strings in the language. These are actually just different perspectives on the same problem, since an algorithm for recognizing a language can be understood as a definition of that language. In the case of finite languages this is easy. We can define a finite language simply by listing all of its strings, and such a list can be "hard-coded" into a recognition algorithm (or "hard-wired" into a machine). It is the infinite case that is more interesting. Since our descriptions and algorithms must, necessarily, be finite, we will be studying, in essence, finite ways of defining infinite sets. These will fall into three broad categories:

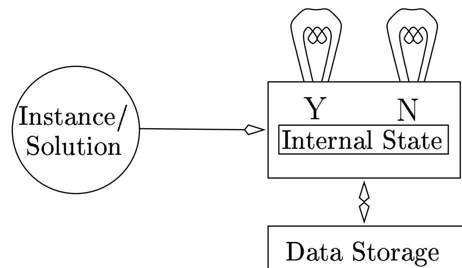
- Algebraic definitions—in which languages are defined by expressions specifying how they are built up from a finite set of simple languages using various operations for combining languages.

- Grammars—which can be understood as algorithms for *generating* languages—for listing all and only the strings in the language.

- Automata—which are simply abstract models of computers specialized to recognize particular languages.

Remarkably, while we can limit the power of these mechanisms in a variety of independent ways it will frequently turn out that the languages definable using a restricted mechanism of one sort will be exactly the languages definable using another mechanism restricted in a completely independent way.

We can get an intuitive idea of what is involved by considering a range of variations in what can be stored in the internal state of our second model of computation.



Here we have limited the internal state to include only a program counter and perhaps some status information, all of which is finitely bounded. We can then vary the power of the model by augmenting this with varying types of data storage. Our objective is to get you thinking about the problem of recognizing strings given various restrictions to your model of computation. We will work with whatever representation of an algorithm you are comfortable with (C or Pascal or, perhaps, some form of pseudo-code—just make sure it is unambiguous). In all of the problems we will assume the same basic machine:

- The program is read-only (it can't be modified, you might even think of it as being hard-wired).

- For the sake of uniformity, let's assume the following methods for accessing the input:

  - `input()`, a function that returns the current input character. You can use this in forms like
    `i:=input()`, or
    `if (input() == 'a' ) then` ..., or
    `push(input())`.
    This does not consume the character; any subsequent calls to `input()` prior to a call to `next()` will return the same character. You may assume that `input()` returns a unique value `EOF` if all of the input has been consumed.

  - `next()`, a function that bumps to the next position in the input. This discards the previous character which cannot be re-read. You can either assume that it returns nothing or that it returns `TRUE` in the case the input is not at `EOF` and `FALSE` otherwise.

- There are no internal data registers. The way we are varying the power of these models is by changing the type of data storage that is available. You cannot save any data in any way except explicitly in the structure you are given. In particular, there are no temporary variables, no `malloc()`s, etc.

- The program returns TRUE if the input is a string in the language and FALSE otherwise.

There are a number of parts to this exercise, but none of them are intended to be very difficult. Where you are told to sketch an algorithm, the algorithm

should be pretty simple—the languages you are asked to recognize are more or less tailored to the computational model you are using. Don't be obsessive about the format or your code—declarations, etc.—only include what is needed to make the algorithm clear. The questions asking whether you can or can't recognize a language within a given model will take a little more thought, but we are not looking for a rigorous proof, just an indication of what makes it hard or even impossible for that model to do it.

3. First model: Computer has a fixed number of bits of storage. You will model this by limiting your program to a single fixed-precision unsigned integer variable, e.g., a single one-byte variable (which, of course, can store only values in the range $[0, \ldots, 255]$), etc. Limit yourself, further, to a single call to input() which occurs in the argument of a `case` (or `switch`) statement. The reason for this will become clear in the last part of this question.

   (a) Sketch an algorithm to recognize the language: $\{(ab)^i \mid i \geq 0\}$ (that is, the set of strings in $\{a, b\}^*$ consisting of zero or more repetitions of ab: $\{\varepsilon, ab, abab, ababab, \ldots\}$).

   (b) How many bits do you need for this (how much precision do you need)? Can you do it with a single bit integer?

   (c) Sketch an algorithm to recognize the language: $\{(abbba)^i \mid i \geq 0\}$.

   (d) How many bits do you need for this?

   (e) Suppose we relax the last limitation and allow any (finite) number of calls to `input` occurring anywhere in the program. Sketch an algorithm for recognizing the language of part (a) using (apparently) *no* data storage. Argue that any algorithm for recognizing this language must store at least one bit of information. Where does your program store it?

4. Second model: Computer has a single unbounded precision counter which you can only increment, decrement and test for zero. (You may assume that it is initially zero or you may include an explicit instruction to clear.) Limit your program to a single unsigned integer variable, and limit your methods of accessing it to something like `inc(i)`, `dec(i)` and a predicate `zero?(i)` which returns true iff $i = 0$. This integer has unbounded precision—it can range over the entire set of natural numbers—so you never have to worry about your counter overflowing.

It is, however, restricted to only the natural numbers—it cannot go negative, so you cannot decrement past zero. You may call `input()` as many times as you like.

(a) Sketch an algorithm to recognize the language: $\{a^i b^i \mid i \geq 0\}$. This is the set of strings consisting of zero or more 'a's followed by *exactly the same number* of 'b's.

(b) Can you do this within the first model of computation? Either sketch an algorithm to do it, or make an informal argument that it can't be done.

(c) Give an informal argument that one can't recognize the language: $\{a^i b^i c^i \mid i \geq 0\}$ within this second model of computation (i.e, with a single counter).

5. Third model: Computer has a single LIFO stack containing fixed precision unsigned integers (so each integer is subject to overflow problems) but which has unbounded depth (so the stack itself never overflows). In your program you should limit yourself to accessing this with methods like `push(i)`, `top()`, `pop()`, and a predicate like `empty?()`. These will push a value into the stack, return the value stored in the top of the stack (the most recent value pushed), discard the top of stack, and test the stack for empty, respectively. Don't forget that you have no storage outside of the stack, so you need to work directly with the values in the stack—you can't pull a value out of the stack and assign it to some other variable. Similarly, while you can, again, call `input()` as often as you wish, the only way to store the value of the input is to push it directly into the stack (e.g., `push(input())`).

   (a) Sketch an algorithm to recognize the language: $\{wcw^{\mathrm{R}} \mid w \in \{a, b\}^*\}$. This is the set of strings made up of any sequence of 'a's and 'b's followed by a 'c' and then exactly the same sequence of 'a's and 'b's in reverse order, so these are all *palindromes* over the alphabet $\{a, b, c\}$ in which 'c' occurs only as the middle symbol. It includes strings like:

   $abbacabba$ $\quad aca \quad$ $abaabcbaaba$ $\quad c$ (which, of course, equals $\varepsilon c \varepsilon$).

   (b) What is your intuition about recognizing this language within the second model (i.e., using just a single counter)?

(c) What is your intuition about the possibility of recognizing the language $\{wcw \mid w \in \{a,b\}^*\}$? This is the set of strings made up of any sequence of 'a's and 'b's followed by a 'c' and then exactly the same sequence of 'a's and 'b's in exactly the same order; it's referred to as the *(deterministic) copy language*.

6. Fourth and final model: Computer has a single FIFO queue of fixed precision unsigned integers with the length of the queue unbounded. You can use access methods similar to those in the third model. In this model you will have something like `front()` that will return the value in the front of the queue (the eldest item) rather than `top()`.

   (a) Sketch an algorithm to recognize the copy language ($\{wcw \mid w \in \{a,b\}^*\}$).

   (b) What is your intuition about the possibility of recognizing the palindrome language of Question 5a ($\{wcw^{\mathrm{R}} \mid w \in \{a,b\}^*\}$)?

4 Inductive Proof

Inductive proofs exploit the structure of inductively defined sets. Since every member of such a set is either one of the basis elements or is built from them using finitely many applications of the inductive clauses, we can prove that every member of the set has a certain property by proving that each of the basis elements has that property and that the inductive clauses preserve it.

4.1 Simple Induction on the Natural Numbers

The most familiar application of induction is in proving properties of the natural numbers. Here we prove that the property holds for zero and that if it holds for an arbitrary natural number n then it holds as well for $n+1$. As an example, consider the proposition:

$$\boldsymbol{P(n):} \qquad \sum_{0 \leq i \leq n} [i] = \frac{n(n+1)}{2}.$$

Proof:

(Basis)

$\sum_{0 \leq i \leq 0} [i] = 0 = 0(0+1)/2.$

(Inductive Hypothesis)

Suppose $\sum_{0 \leq i \leq n} [i] = n(n+1)/2.$

(Induction)

To show that $\sum_{0 \leq i \leq (n+1)} [i] = (n+1)((n+1)+1)/2$:

$$\begin{array}{lcll} \sum_{0 \leq i \leq (n+1)} [i] & = & \sum_{0 \leq i \leq n} [i] + (n+1) & \\ & = & n(n+1)/2 + (n+1) & \text{by IH} \\ & = & (n(n+1) + 2(n+1))/2 & \\ & = & (n+2)(n+1)/2 & \text{distributive law} \\ & = & (n+1)((n+1)+1)/2. & \end{array}$$

⊣

It is worth considering why this works. The only case we explicitly prove is the base case ($n = 0$). To see that the proof implies that the proposition holds for all natural numbers, let k be an arbitrary natural number. Now

either $k = 0$ and we are done, or k is $n+1$ for some n, namely $k-1$, and we can apply the inductive step. But this actually proves a hypothetical: *If* the property is true of $k-1$ then it is also true of k. By itself this does not prove the k case; all it does is replace the need to prove the k case with a need to prove the $k-1$ case. If $k-1$ happened to be 0 we would, again, be done, since this case is proved. On the other hand, if $k-1 \neq 0$ then the inductive step applies again, reducing the $k-1$ case to the $k-2$ case; the inductive step applies uniformly to all cases other than the base case. So, in a sense, we are building a chain of hypotheticals:

$$P(k) \Leftarrow P(k-1) \Leftarrow P(k-2) \Leftarrow \cdots$$

(The '$\Leftarrow$' should be read "is implied by" or "if".) The key is that for any actual value of n we are guaranteed to arrive at the base case after some finite number of applications of the inductive step. So, invariably, the chain of hypotheticals has the form:

$$P(k) \Leftarrow P(k-1) \Leftarrow P(k-2) \Leftarrow \cdots \Leftarrow P(1) \Leftarrow P(0).$$

Since we have proved $P(0)$, the implication proves $P(1)$ which in turn proves $P(2)$, etc. Each of the propositions in the chain is true. In particular, $P(k)$ is true and, since the choice of k was arbitrary, $P(n)$ is true for all $n \in \mathbb{N}$.

4.2 Simple Induction on Inductively Defined Sets

In general induction is valid for any *well-ordered* set, that is, for any set with a well-defined notion of predecessor for which there are no infinite sequences of elements related by predecessor. What this means is that, while any given point may have many immediate predecessors, if one follows any chain of predecessors from that point one will invariably arrive at a *minimal* element—one with no predecessors. The minimal elements form the basis of the set.

Sets that are defined inductively are always well-ordered. The basis of the definition is the set of minimal elements. The predecessors of an element introduced by an inductive clause are the elements from which it is constructed. Since every element is either a basis element or is constructed by finitely many applications of the inductive clauses, it is certain that every chain of predecessors will reach minimal elements in finitely many steps.

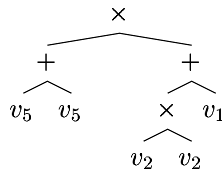
As an example, consider the following set EXPR of simple algebraic expressions over a set of variables $V = \{v_1, v_2, \ldots\}$:

- Every $v_i \in V$ is an expression in EXPR.
- If x and y are expressions in EXPR then
  - $(x + y)$ is an expression in EXPR,
  - $(x \times y)$ is an expression in EXPR.
- Nothing else is an expression in EXPR.

Here we have a set of fully parenthesized expressions including, for instance,

$$v_2 \qquad (v_1 + v_2) \qquad ((v_5 + v_5) \times ((v_2 \times v_2) + v_1)).$$

The basis elements are just the v_i. The predecessors of $(v_1 + v_2)$ are v_1 and v_2, and the predecessors of $((v_5 + v_5) \times ((v_2 \times v_2) + v_1))$ are $(v_5 + v_5)$ and $((v_2 \times v_2) + v_1)$. It is often helpful to analyze elements of sets like this in terms of their *syntax tree*. For $((v_5 + v_5) \times ((v_2 \times v_2) + v_1))$ this is:



The leaves of the syntax tree are labeled with the basis elements from which the expression is built; the interior nodes are labeled with the operations of the inductive clauses that build it.

The steps of an inductive proof of a property of such a set are modeled directly on the steps of its definition. First one proves that the property holds for each of the basis elements. Then one proves that if the property holds for each of the predecessors of an element then it holds for that element as well.

For example, to prove that every expression in EXPR has properly balanced parentheses it suffices to prove that every right parenthesis has a matching left parenthesis and *v.v.*. It is not hard to see that this comes down to the property that every expression has equal numbers of left and right parentheses and every prefix of the expression has at least as many left parentheses as right. We can prove that every expression in EXPR has this property as follows:

Proof:

(Basis)

Every $v_i \in V$ is an expression with neither left nor right parenthesis and therefore satisfies the property.

(Inductive Hypothesis)

Suppose that x is a non-minimal expression in EXPR and that each of the predecessors of x satisfy the property.

(Induction)

By the definition of EXPR x is either $(y+z)$ or $(y \times z)$, where y and z are the predecessors of x. Suppose, for the sake of the argument, that x is $(y+z)$; a similar argument applies to $(y \times z)$. By the induction hypothesis, both y and z have equal numbers of left and right parentheses, and every prefix either of y or of z has at least as many left as right parentheses. Since x adds one left and one right parenthesis, these occur in equal numbers in x as well. Note that every prefix of x either is empty, is '(' followed by a (possibly empty, not necessarily proper) prefix of y, is '$(y+$' followed by a similar prefix of z, or is '$(y+z)$'. It is easy to verify that under each of these conditions the fact that the prefixes of y and z have at least as many left as right parentheses implies that the prefixes of x do as well. $\dashv$

The key thing to note about all of this is that the proof follows the definition exactly. This is one compelling reason to define sets inductively, and we shall generally endeavor to do so whenever we reasonably can. While one can frequently find simpler (read cleverer) proofs, if a proposition over an inductively defined set is valid the inductive proof will always go through. Furthermore, it can be applied nearly automatically. When in doubt, use induction.

If, following Peano, we define the Natural Numbers inductively in terms of zero and the successor function $s(n)$ then induction over $\mathbb{N}$ becomes an instance of this schema:

Definition 17 (Natural Numbers (Peano))

- $0 \in \mathbb{N}$.
- *If* $n \in \mathbb{N}$ *then* $s(n) \in \mathbb{N}$.
- *Nothing else.*

Here the number we usually denote with the numeral '3' (or '11', working in binary, '10' in ternary, etc.) is defined as $s(s(s(0)))$—the successor of the successor of the successor of zero—a much more satisfactory definition than "the number we usually denote with the numeral '3'". The predecessor of $s(n)$ is just n, or, in the earlier terms, the predecessor of $n+1$ is n.

4.3 Complete Induction

It often occurs that the way one needs to decompose a case in an inductive proof does not reduce it to immediate predecessors. Under these circumstances, one can use the superficially stronger form known as *complete induction* in which one shows that if the proposition is satisfied by *every* element "smaller" than the case to be proved (in the ordering induced by predecessor) then it is satisfied by that case as well. This is sometimes referred to as *strong induction* but it, in fact, is no stronger than simple induction (at least over countable sets). It is, however, often more convenient than simple induction and is no harder to use.

Consider, as an example, the proposition:

$$F_n = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right) - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right),$$

where F_n is the nth Fibonacci number:

$$F_0 \stackrel{\text{def}}{=} 0, \qquad F_1 \stackrel{\text{def}}{=} 1, \qquad F_{m+2} \stackrel{\text{def}}{=} F_{m+1} + F_m, \qquad m \geq 0.$$

This is not a simple inductive definition and, as a result, the most direct way to approach the proof is to base the induction on n. But this leads to a difficulty for simple induction because in order to prove the proposition for the $m+2$ case we will need to assume the proposition not only for $m+1$ but for m as well. Thus we will need the IH to cover both the predecessor and the predecessor of the predecessor. Complete induction gives us both of these and everything else down to the base cases.

Proof: Let

$$a = \frac{1+\sqrt{5}}{2} \quad \text{and } b = \frac{1-\sqrt{5}}{2}.$$

The claim, then, is

$$F_n = \frac{1}{\sqrt{5}}(a^n - b^n).$$

The proof contains one possibly non-obvious algebraic trick:

$$a^2 = \left(\frac{1+\sqrt{5}}{2}\right)^2 = \frac{1+2\sqrt{5}+5}{4} = 1 + \frac{1+\sqrt{5}}{2} = a+1,$$

and, similarly,

$$b^2 = \left(\frac{1-\sqrt{5}}{2}\right)^2 = \frac{1-2\sqrt{5}+5}{4} = 1 + \frac{1-\sqrt{5}}{2} = b+1.$$

(Basis)

$$\begin{array}{rcl} F_0 & = & \frac{1}{\sqrt{5}}(a^0 - b^0) = 0. \\ F_1 & = & \frac{1}{\sqrt{5}}(a-b) \\ & = & \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}-(1-\sqrt{5})}{2}\right) \\ & = & \frac{1}{\sqrt{5}}\sqrt{5} = 1. \end{array}$$

(Inductive Hypothesis)
Suppose that the proposition is true for all F_n where $n < m+2$.

(Induction)
To show that it is true for F_{m+2} as well:

$$\begin{array}{rcll} F_{m+2} & = & F_{m+1} + F_m & \\ & = & \frac{1}{\sqrt{5}}(a^{m+1} - b^{m+1} + a^m - b^m) & \text{by IH} \\ & = & \frac{1}{\sqrt{5}}(a^m(a+1) - b^m(b+1)) & \\ & = & \frac{1}{\sqrt{5}}(a^{m+2} - b^{m+2}) & \text{by result above.} \end{array}$$

$\dashv$

4.4 Induction Loading

Note that all that differs between simple and complete induction is the induction hypothesis: the hypothesis used in complete induction (that the proposition holds for all smaller cases) is *stronger* than that used in simple induction (that it holds only for the immediate predecessors). It is just as valid, since

the same kinds of sequences of hypotheticals work for every element of the set. But being stronger, it makes it easier to prove the inductive case.

This idea of strengthening the hypothesis to simplify the proof often works not only when we assume the proposition is true of more elements, but also when we strengthen the proposition itself, that is, when using induction, it is often easier to prove a stronger property than the result you are trying to obtain. This is because the induction hypothesis is formed from that property; the stronger it is, the stronger the hypothesis one has to work with. While there is more to prove, one has more ammunition with which to prove it. Consider, for instance, the following set of strings $L \subseteq \{a, b\}^*$:

- $a \in L, \qquad ab \in L.$
- If $w \in L$ and $v \in L$ then $w \cdot v \in L$.
- Nothing else.

We would like to prove the proposition: 'bb' does not occur as a substring of any string in L. Intuitively, this should be fairly obvious. The only stumbling block to proving this by induction (on the structure of the string) is proving that concatenating two strings in L never juxtaposes 'b's at the boundary between them. This is simplified if we strengthen the proposition to: If $w \in L$ then 'bb' does not occur as a substring of w and $w = aw'$ for some $w' \in \{a, b\}^*$.

Proof:

(Basis)

Clearly, the strengthened proposition is satisfied by both a and ab.

(Inductive Hypothesis)

Suppose $w = v \cdot u$ for some $v, u \in L$ and that the strengthened proposition is true of both u and v.

(Induction)

By the induction hypothesis 'bb' does not occur in either of v or u. Moreover, the initial symbol in u is an 'a', so 'bb' does not occur at the boundary between the two either. Finally, by the IH again, the initial symbol of v is also 'a' and, thus, so is the initial symbol of w. $\dashv$

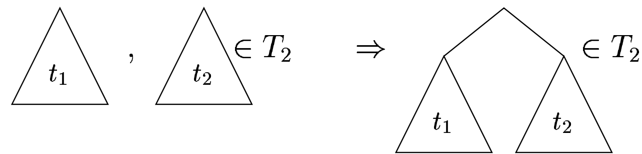
Now for a couple of exercises.

7. Prove for all $w, v \in \Sigma^*$ for any alphabet Σ, that $(wv)^{\mathrm{R}} = v^{\mathrm{R}} \cdot w^{\mathrm{R}}$.

8. Let T_2 be the set of all binary-branching trees, where

   - The trivial tree (consisting of a single node) is in T_2.
   - If t_1 and t_2 are trees in T_2, then the tree formed by taking t_1 as the left subtree, t_2 as the right subtree, and a new node as a root is also a tree in T_2.
   - Nothing else.

   Graphically, the inductive clause says:



   For every tree $t \in T_2$ let $d(t)$ be the *depth* of t defined:

$$d(t) = \begin{cases} 0 & \text{if } t \text{ is trivial,} \\ 1 + \max(d(t_1), d(t_2)) & \text{if } t \text{ is constructed from } t_1 \text{ and } t_2. \end{cases}$$

   The *leaves* in a tree $t \in T_2$ are its trivial subtrees—the nodes which have no descendents. Let $l(t)$ denote the number of leaves in the tree t. Prove for all $t \in T_2$ that $d(t) + 1 \leq l(t) \leq 2^{d(t)}$.