CHAPTER 3

The Big Picture: A Language Hierarchy

Our goal, in the rest of this book, is to build a framework that lets us examine a new problem and be able to say something about how intrinsically difficult it is. In order to do this, we need to be able to compare problems that appear, at first examination, to be wildly different. Apples and oranges come to mind. So the first thing we need to do is to define a single framework into which any computational problem can be cast. Then we will be in a position to compare problems and to distinguish between those that are relatively easy to solve and those that are not.

3.1 Defining the Task: Language Recognition

The unifying framework that we will use is language recognition. Assume that we are given:

- The definition of a language L. (We will consider about half a dozen different techniques for providing this definition.)
- A string w.

Then we must answer the question: "Is w in L?" This question is an instance of a more general class that we will call decision problems. A ***decision problem*** is simply a problem that requires a yes or no answer.

In the rest of this book, we will discuss programs to solve decision problems specifically of the form, "Is w in L?" We will see that, for some languages, a very simple program suffices. For others, a more complex one is required. For still others, we will prove that no program can exist.

3.2 The Power of Encoding

The question that we are going to ask, "Is w in L?" may seem, at first glance, way too limited to be useful. What about problems like multiplying numbers, sorting lists, and retrieving values from a database? And what about real problems like air traffic control or inventory management? Can our theory tell us anything interesting about them?

The answer is yes and the key is encoding. With an appropriate encoding, other kinds of problems can be recast as the problem of deciding whether a string is in a language. We will show some examples to illustrate this idea. We will divide the examples into two categories:

- Problems that are already stated as decision problems. For these, all we need to do is to encode the inputs as strings and then define a language that contains exactly the set of inputs for which the desired answer is yes.
- Problems that are not already stated as decision problems. These problems may require results of any type. For these, we must first reformulate the problem as a decision problem and then encode it as a language recognition task.

3.2.1 Everything is a String

Our stated goal is to build a theory of computation. What we are actually about to build is a theory specifically of languages and strings. Of course, in a computer's memory, everything is a (binary) string. So, at that level, it is obvious that restricting our attention to strings does not limit the scope of our theory. Often, however, we will find it easier to work with languages with larger alphabets.

Each time we consider a new problem, our first task will be to describe it in terms of strings. In the examples that follow, and throughout the book, we will use the notation $<X>$ to mean a string encoding of some object X. We'll use the notation $<X, Y>$ to mean the encoding, into a single string, of the two objects X and Y.

The first three examples we'll consider are of problems that are naturally described in terms of strings. Then we'll look at examples where we must begin by constructing an appropriate string encoding.

EXAMPLE 3.1 Pattern Matching on the Web

- Problem: Given a search string w and a web document d, do they match? In other words, should a search engine, on input w, consider returning d?
- The language to be decided: $\{<w, d> : d \text{ is a candidate match for the query } w\}$.

EXAMPLE 3.2 Question-Answering on the Web

- Problem: Given an English question q and a web document d (which may be in English or Chinese), does d contain the answer to q?
- The language to be decided: $\{<q, d> : d \text{ contains the answer to } q\}$.

The techniques that we will describe in the rest of this book are widely used in the construction of systems that work with natural language (e. g., English or Spanish or Chinese) text and speech inputs. (Appendix L)

EXAMPLE 3.3 Does a Program Always Halt?

- Problem: Given a program p, written in some standard programming language, is p guaranteed to halt on all inputs?
- The language to be decided: $HP_{ALL} = \{p : p \text{ halts on all inputs}\}$.

A procedure that could decide whether or not a string is in HP_{ALL} could be an important part of a larger system that proves the correctness of a program. Unfortunately, as we will see in Theorem 21.3, no such procedure can exist.

EXAMPLE 3.4 Primality Testing

- Problem: Given a nonnegative integer n, is it prime? In other words, does it have at least one positive integer factor other than itself and 1?
- An instance of the problem: Is 9 prime?
- Encoding of the problem: We need a way to encode each instance. We will encode each nonnegative integer as a binary string.
- The language to be decided: PRIMES = $\{w : w \text{ is the binary encoding of a prime number}\}$.

Prime numbers play an important role in modern cryptography systems. (J.3) We'll discuss the complexity of PRIMES in Section 28.1.7 and again in Section 30.2.4.

EXAMPLE 3.5 Verifying Addition

- Problem: Verify the correctness of the addition of two numbers.
- Encoding of the problem: We encode each of the numbers as a string of decimal digits. Each instance of the problem is a string of the form:

$$<integer_1> + <integer_2> = <integer_3>.$$

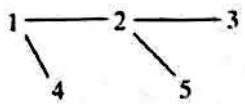
EXAMPLE 3.5 (*Continued*)

- The language to be decided:

  INTEGERSUM = {w of the form: <$integer_1$>+ <$integer_2$> = <$integer_3$> : each of the substrings <$integer_1$>.<$integer_2$> and <$integer_3$> is an element of {0, 1, 2, 3, 4, 5, 6, 8, 9} and $integer_3$ is the sum of $integer_1$ and $integer_2$}.
- Examples of strings in L: `2 + 4 = 6` `23 + 47 = 70.`
- Examples of strings not in L: `2 + 4 = 10` `2 + 4.`

EXAMPLE 3.6 Graph Connectivity

- Problem: Given an undirected graph G, is it connected? In other words, given any two distinct vertices x and y in G, is there a path from x to y?
- Instance of the problem: Is the following graph connected?



- Encoding of the problem: Let V be a set of binary numbers, one for each vertex in G. Then we construct <G> as follows:
  - Write $|V|$ as a binary number.
  - Write a list of edges, each of which is represented by a pair of binary numbers corresponding to the vertices that the edge connects.
  - Separate all such binary numbers by the symbol /.

For example, the graph shown above would be encoded by the following string, which begins with an encoding of 5 (the number of vertices) and is followed by four pairs corresponding to the four edges:

`101/1/10/10/11/1/100/10/101.`

- The language to be decided:

  CONNECTED = {$w \in \{0, 1, /\}^* : w = n_1/n_2/\ldots n_i$, where each n_i is a binary string and w encodes a connected graph, as described above}.

EXAMPLE 3.7 Protein Sequence Alignment

- Problem: Given a protein fragment f and a complete protein molecule p, could f be a fragment from p?

- Encoding of the problem: Represent each protein molecule or fragment as a sequence of amino acid residues. Assign a letter to each of the 20 possible amino acids. So a protein fragment might be represented as AGHTYWDNR.
- The language to be decided: $\{<f, p> f$ could be a fragment from $p\}$.

> The techniques that we will describe in the rest of this book are widely used in computational biology. (Appendix K)

In each of these examples, we have chosen an encoding that is expressive enough to make it possible to describe all of the instances of the problem we are interested in. But have we chosen a good encoding? Might there be another one? The answer to this second question is yes. And it will turn out that the encoding we choose may have a significant impact on what we can say about the difficulty of solving the original problem. To see an example of this, we need look no farther than the addition problem that we just considered. Suppose that we want to write a program to examine a string in the addition language that we proposed above. Suppose further that we impose the constraint that our program reads the string one character at a time, left to right. It has only a finite (bounded in advance, independent of the length of the input string) amount of memory. These restrictions correspond to the notion of a finite state machine, as we will see in Chapter 5. It turns out that no machine of this sort can decide the language that we have described. We'll see how to prove results such as this in Chapter 8.

But now consider a different encoding of the addition problem. This time we encode each of the numbers as a binary string, and we write the digits, from lowest order to highest order, left to right (i.e., backwards from the usual way). Furthermore, we imagine the three numbers aligned in the way they often are when we draw an addition problem. So we might encode 10 + 4 = 14 as:

```
 0101    writing 1010 backwards
+0010    writing 0100 backwards
 0111    writing 1110 backwards
```

We now encode each column of that sum as a single character. Since each column is a sequence of three binary digits, it may take on any one of 8 possible values. We can use the symbols a, b, c, d, e, f, g, and h to correspond to 000, 001, 010, 011, 100, 101, 110, and 111, respectively. So we could encode the 10 + 4 = 14 example as afdf.

It is easy to design a program that reads such a string, left to right, and decides, as each character is considered, whether the sum so far is correct. For example, if the first character of a string is c, then the sum is wrong, since 0 + 1 cannot be 0 (although it could be later if there were a carry bit from the previous column).

> This idea is the basis for the design of binary adders, as well as larger circuits, like multipliers, that exploit them. (P3)

In Part V of this book we will be concerned with the efficiency (stated in terms of either time or space) of the programs that we write. We will describe both time and space requirements as functions of the length of the program's input. When we do that, it may matter what encoding scheme we have picked since some encodings produce longer strings than others do. For example, consider the integer 25. It can be encoded:

- In decimal as: 25,
- In binary as: 11001, or
- In unary as: 1111111111111111111111111.

We'll return to this issue in Section 27.3.1.

3.2.2 Casting Problems as Decision Questions

Problems that are not already stated as decision questions can be transformed into decision questions. More specifically, they can be reformulated so that they become language recognition problems. The idea is to encode, into a single string, both the inputs and the outputs of the original problem P. So, for example, if P takes two inputs and produces one result, we could construct strings of the form $i_1; i_2; r$. Then a string $s = x; y; z$ is in the language L that corresponds to P iff z is the result that P produces given the inputs x and y.

EXAMPLE 3.8 Casting Addition as Decision

- Problem: Given two nonnegative integers, compute their sum.
- Encoding of the problem: We transform the problem of adding two numbers into the problem of checking to see whether a third number is the sum of the first two. We can use the same encoding that we used in Example 3.5.
- The language to be decided:
  INTEGERSUM = {w of the form: <*integer*$_1$>+<*integer*$_2$>=<*integer*$_3$>, where each of the substrings <*integer*$_1$>, <*integer*$_2$>, and <*integer*$_3$> is an element of $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}^+$ and *integer*$_3$ is the sum of *integer*$_1$ and *integer*$_2$}.

EXAMPLE 3.9 Casting Sorting as Decision

- Problem: Given a list of integers, sort it.
- Encoding of the problem: We transform the problem of sorting a list into the problem of examining a pair of lists and deciding whether the second corresponds to the sorted version of the first.

- The language to be decided:

$$L = \{w_1 \# w_2 : \exists n \geq 1 \ (w_1 \text{ is of the form } int_1, int_2, \ldots int_n, \\ w_2 \text{ is of the form } int_1, int_2, \ldots int_n, \text{ and} \\ w_2 \text{ contains the same objects as } w_1 \text{ and } w_2 \text{ is sorted})\}.$$

- Example of a string in L: `1,5,3,9,6#1,3,5,6,9.`
- Example of a string not in L: `1,5,3,9,6#1,2,3,4,5,6,7.`

EXAMPLE 3.10 Casting Database Querying as Decision

- Problem: Given a database and a query, execute the query against the database.
- Encoding of the problem: We transform the task of executing the query into the problem of evaluating a reply to see if it is correct.
- The language to be decided:

$$L = \{d \# q \# a : d \text{ is an encoding of a database,} \\ q \text{ is a string representing a query, and} \\ a \text{ is the correct result of applying } q \text{ to } d\}.$$

- Example of a string in L:

```
(name, age, phone), (John, 23, 567-1234) (Mary, 24, 234-9876 )#
(select name age=23) #
(John).
```

Given each of the problems that we have just considered, there is an important sense in which the encoding of the problem as a decision question is equivalent to the original formulation of the problem: Each can be reduced to the other. We'll have a lot more to say about the idea of reduction in Chapter 21. But, for now, what we mean by ***reduction*** of one problem to another is that, if we have a program to solve the second, we can use it to build a program to solve the first. For example, suppose that we have a program P that adds a pair of integers. Then the following program decides the language INTEGERSUM, which we described in Example 3.8:

Given a string of the form $<integer_1>+<integer_2>=<integer_3>$ do:

1. Let x = *convert-to-integer* ($<integer_1>$).
2. Let y = *convert-to-integer* ($<integer_2>$).
3. Let $z = P(x, y)$.
4. If z = *convert-to-integer* ($<integer_3>$) then accept. Else reject.

Alternatively, if we have a program T that decides INTEGERSUM, then the following program computes the sum of two integers x and y:

1. Lexicographically enumerate the strings that represent decimal encodings of nonnegative integers.
2. Each time a string s is generated, create the new string $<x>+<y>=s$.
3. Feed that string to T.
4. If T accepts $<x>+<y>=s$, halt and return *convert-to-integer*(s).

3.3 A Machine-Based Hierarchy of Language Classes

In Parts II, III, and IV, we will define a hierarchy of computational models, each more powerful than the last. The first model is simple: Programs written for it are generally easy to understand, they run in linear time, and algorithms exist to answer almost any question we might wish to ask about such programs. The second model is more powerful, but still limited. The last model is powerful enough to describe anything that can be computed by any sort of real computer. All of these models will allow us to write programs whose job is to accept some language L. In this section, we sketch this machine hierarchy and provide a short introduction to the language hierarchy that goes along with it.

3.3.1 The Regular Languages

The first model we will consider is the ***finite state machine*** or ***FSM***. Figure 3.1 shows a simple FSM that accepts strings of a's and b's, where all a's come before all b's.

The input to an FSM is a string, which is fed to it one character at a time, left to right. The FSM has a start state, shown in the diagram with an unlabelled arrow leading to it, and some number (zero or more) of accepting states, which will be shown in our diagrams with double circles. The FSM starts in its start state. As each character is read, the FSM changes state based on the transitions shown in the figure. If an FSM M is in an accepting state after reading the last character of some input string s, then M accepts s. Otherwise it rejects it. Our example FSM stays in state 1 as long as it is reading a's. When it sees a b, it moves to state 2, where it stays as long as it continues seeing b's. Both state 1 and state 2 are accepting states. But if, in state 2, it sees an a, it goes to state 3, a nonaccepting state, where it stays until it runs out of input. So, for example, this machine will accept aab, aabbb, and bb. It will reject ba.

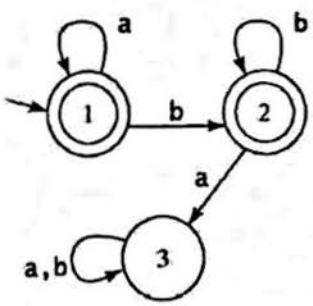


FIGURE 3.1 A simple FSM.

We will call the class of languages that can be accepted by some FSM ***regular***. As we will see in Part II, many useful languages are regular, including binary strings with even parity, syntactically well-formed floating point numbers, and sequences of coins that are sufficient to buy a soda.

3.3.2 The Context-Free Languages

But there are useful simple languages that are not regular. Consider, for example, Bal, the language of balanced parentheses. Bal contains strings like (()) and ()(); it does not contain strings like ()))(. Because it's hard to read strings of parentheses, let's consider instead the related language $A^nB^n = \{a^nb^n : n \geq 0\}$. In any string in A^nB^n, all the a's come first and the number of a's equals the number of b's. We could try to build an FSM to accept A^nB^n. But the problem is, "How shall we count the a's so that we can compare them to the b's?" The only memory in an FSM is in the states and we must choose a fixed number of states when we build our machine. But there is no bound on the number of a's we might need to count. We will prove in Chapter 8 that it is not possible to build an FSM to accept A^nB^n.

But languages like Bal and A^nB^n are important. For example, almost every programming language and query language allows parentheses, so any front end for such a language must be able to check to see that the parentheses are balanced. Can we augment the FSM in a simple way and thus be able to solve this problem? The answer is yes. Suppose that we add one thing, a single stack. We will call any machine that consists of an FSM, plus a single stack, a ***pushdown automaton*** or ***PDA***.

We can easily build a PDA M to accept A^nB^n. The idea is that, each time it sees an a, M will push it onto the stack. Then, each time it sees a b, it will pop an a from the stack. If it runs out of input and stack at the same time and it is in an accepting state, it will accept. Otherwise, it will reject. M will use the same state structure that we used in our FSM example above to guarantee that all the a's come before all the b's. In diagrams of PDAs, read an arc label of the form $x/y/z$ to mean, "if the input is an x, and it is possible to pop y off the stack, then take the transition, do the pop of y, and push z". If the middle argument is ε, then don't bother to check the stack. If the third argument is ε, then don't push anything. Using those conventions, the PDA shown in Figure 3.2 accepts A^nB^n.

Using a very similar sort of PDA, we can build a machine to accept Bal and other languages whose strings are composed of properly nested substrings. For example, a ***palindrome*** is a string that reads the same right-to-left as it does left-to right. We can easily build a PDA to accept the language PalEven = $\{ww^R : w \in \{a, b\}^*\}$, the

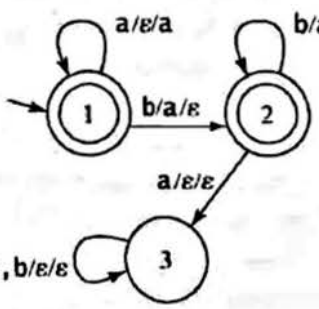


FIGURE 3.2 A simple PDA that accepts A^nB^n.

language of even-length palindromes of a's and b's. The PDA for PalEven simply pushes all the characters in the first half of its input string onto the stack, guesses where the middle is, and then starts popping one character for each remaining input character. If there is a guess that causes the pushed string (which will be popped off in reverse order) to match the remaining input string, then the input string is in PalEven.

But we should note some simple limitations to the power of the PDA. Consider the language WW = $\{ww : w \in \{a, b\}^*\}$, which is just like PalEven except that the second half of each of its strings is an exact copy of the first half (rather than the reverse of it). Now, as we'll prove in Chapter 13, it is not possible to build an accepting PDA (although it would be possible to build an accepting machine if we could augment the finite state controller with a first-in, first-out queue rather than a stack).

We will call the class of languages that can be accepted by some PDA ***context-free***. As we will see in Part III, many useful languages are context-free, including most programming languages, query languages, and markup languages.

3.3.3 The Decidable and Semidecidable Languages

But there are useful straightforward languages that are not context-free. Consider, for example, the language of English sentences in which some word occurs more than once. As an even simpler (although probably less useful) example, consider another language to which we will give a name. Let $A^nB^nC^n = \{a^nb^nc^n : n \geq 0\}$, i.e., the language composed of all strings of a's, b's, and c's such that all the a's come first, followed by all the b's, then all the c's, and the number of a's equals the number of b's equals the number of c's. We could try to build a PDA to accept $A^nB^nC^n$. We could use the stack to count the a's, just as we did for A^nB^n. We could pop the stack as the b's come in and compare them to the a's. But then what shall we do about the c's? We have lost all information about the a's and the b's since, if they matched, the stack will be empty. We will prove in Chapter 13 that it is not possible to build a PDA to accept $A^nB^nC^n$.

But it is easy to write a program to accept $A^nB^nC^n$. So, if we want a class of machines that can capture everything we can write programs to compute, we need a model that is stronger than the PDA. To meet this need, we will introduce a third kind of machine. We will get rid of the stack and replace it with an infinite tape. The tape will have a single read/write head. Only the tape square under the read/write head can be accessed (for reading or for writing). The read/write head can be moved one square in either direction on each move. The resulting machine is called a ***Turing machine***. We will also change the way that input is given to the machine. Instead of streaming it, one character at a time, the way we did for FSMs and PDAs, we will simply write the input string onto the tape and then start the machine with the read/write head just to the left of the first input character. We show the structure of a Turing machine in Figure 3.3. The arrow under the tape indicates the location of the read/write head.

At each step, a Turing machine M considers its current state and the character that is on the tape directly under its read/write head. Based on those two things, it chooses its next state, chooses a character to write on the tape under the read/write head, and chooses whether to move the read/write head one square to the right or one square to the left. A finite segment of M's tape contains the input string. The rest is blank, but M may move the read/write head off the input string and write on the blank squares of the tape.

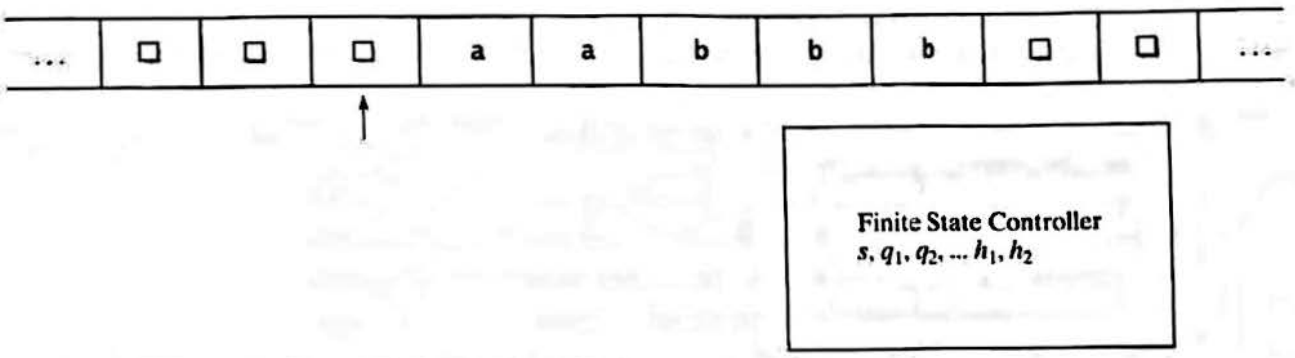


FIGURE 3.3 The structure of a Turing machine.

There exists a simple Turing machine that accepts $A^nB^nC^n$. It marks off the leftmost a, scans to the right to find a b, marks it off, continues scanning to the right, finds a c, and marks it off. Then it goes back to the left, marks off the next a, and so forth. When it runs out of a's, it makes one final pass to the right to make sure that there are no extra b's or c's. If that check succeeds, the machine accepts. If it fails, or if at any point the machine failed to find a required b or c, it rejects. For the details of how this machine operates, see Example 17.8.

Finite state machines and pushdown automata (with one technical exception that we can ignore for now) are guaranteed to halt. They must do so when they run out of input. Turing machines, on the other hand, carry no such guarantee. The input simply sits on the tape. A Turing machine may (and generally does) move back and forth across its input many times. It may move back and forth forever. Or it may simply move in one direction, off the input onto the blank tape, and keep going forever. Because of its flexibility in using its tape to record its computation, the Turing machine is a more powerful model than either the FSM or the PDA. In fact, we will see in Chapter 18 that any computation that can be written in any programming language or run on any modern computer can be described as a Turing machine. However, when we work with Turing machines, we must be aware of the fact that they cannot be guaranteed to halt. And, unfortunately we can prove (as we will do in Chapter 19) that there exists no algorithm that can examine a Turing machine and tell whether or not it will halt (on any one input or on all inputs). This fundamental result about the limits of computation is known as the undecidability of the halting problem.

We will use the Turing machine to define two new classes of languages:

- A language L is ***decidable*** iff there exists a Turing machine M that halts on all inputs, accepts all strings that are in L, and rejects all strings that are not in L. In other words, M can always say yes or no, as appropriate.
- A language L is ***semidecidable*** iff there exists a Turing machine M that accepts all strings that are in L and fails to accept every string that is not in L. Given a string that is not in L, M may reject or it may loop forever. In other words, M can recognize a solution and then say yes, but it may not know when it should give up looking for a solution and say no.

Bal, A^nB^n, PalEven, WW, and $A^nB^nC^n$ are all decidable languages. Every decidable language is also semidecidable (since the requirement for semidecidability is strictly weaker than the requirement for decidability). But there are languages that are semidecidable yet not decidable. As an example, consider $L = \{<p, w>: p$ is a Java program that halts on input $w\}$. L is semidecidable by a Turing machine that simulates p running on w. If the simulation halts, the semidecider can halt and accept. But, if the simulation does not halt, the semidecider will not be able to recognize that it isn't going to. So it has no way to halt and reject. Just as there exists no algorithm that can examine a Turing machine and decide whether or not it will halt, there is no algorithm to examine a Java program (without having to run it) and make that determination. So L is semidecidable but not decidable.

3.3.4 The Computational Hierarchy and Why It Is Important

We have now defined four language classes:

1. Regular languages, which can be accepted by some finite state machine.
2. Context-free languages, which can be accepted by some pushdown automaton.
3. Decidable (or simply D) languages, which can decided by some Turing machine that always halts.
4. Semidecidable (or SD) languages, which can be semidecided by some Turing machine that halts on all strings in the language.

Each of these classes is a proper subset of the next class, as illustrated in the diagram shown in Figure 3.4.

As we move outward in the language hierarchy, we have access to tools with greater and greater expressive power. So, for example, we can define A^nB^n as a context-free language but not as a regular one. We can define $A^nB^nC^n$ as a decidable language but not as a context-free or a regular one. This matters because expressiveness generally comes at a price. The price may be:

- ***Computational efficiency***: Finite state machines run in time that is linear in the length of the input string. A general context-free parser based on the idea of a pushdown automaton requires time that grows as the cube of the length of the input string. A Turing machine may require time that grows exponentially (or faster) with the length of the input string.
- ***Decidability***: There exist procedures to answer many useful questions about finite state machines. For example, does an FSM accept some particular string? Is an FSM minimal (i.e., is it the simplest machine that does the job it does)? Are two FSMs identical? A subset of those questions can be answered for pushdown automata. None of them can be answered for Turing machines.
- ***Clarity***: There exist tools that enable designers to draw and analyze finite state machines. Every regular language can also be described using the (often very convenient) regular expression pattern language that we will define in Chapter 6. Every context-free language, in addition to being recognizable by some pushdown

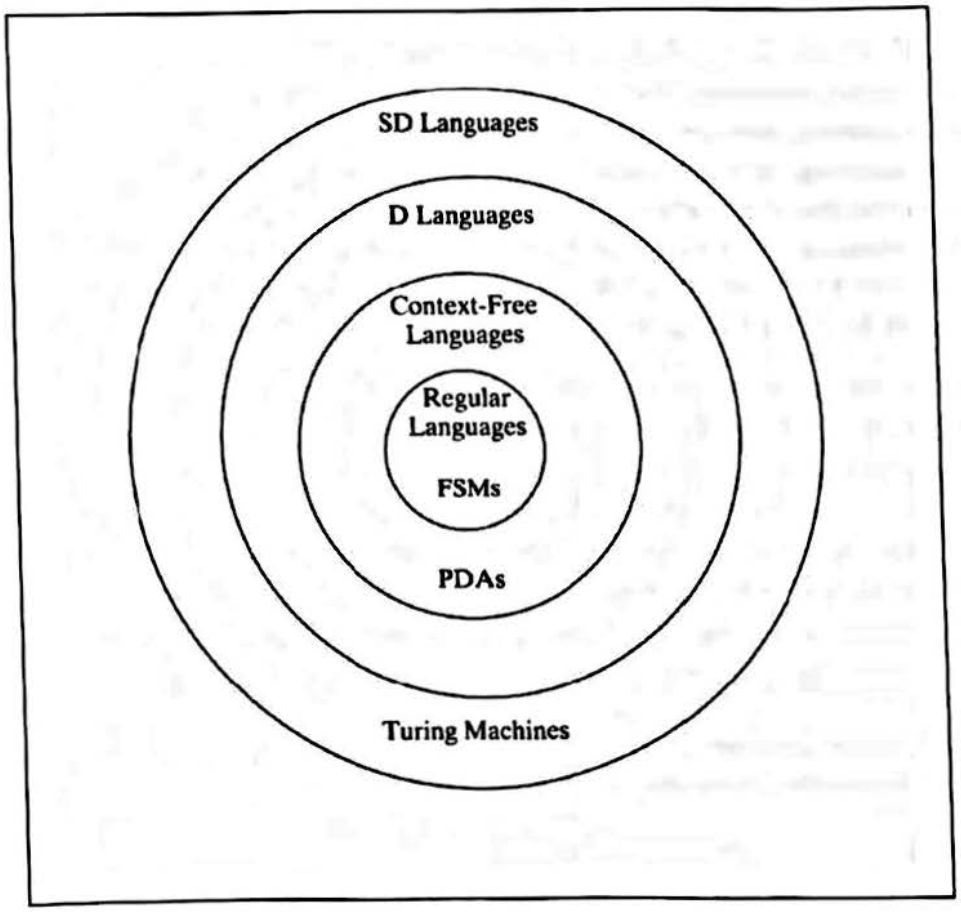


FIGURE 3.4 A hierarchy of language classes.

automaton, can (as we will see in Chapter 11) be described with a context-free grammar. For many important kinds of languages, context-free grammars are sufficiently natural that they are commonly used as documentation tools. No corresponding tools exist for the broader classes of decidable and semidecidable languages.

So, as a practical as well as a theoretical matter, it makes sense, given a particular problem, to describe it using the simplest (i.e., expressively weakest) formalism that is adequate to the job.

> ***The Rule of Least Power***[5]: "Use the least powerful language suitable for expressing information, constraints or programs on the World Wide Web."

Although stated in the context of the World Wide Web, the Rule of Least Power applies far more broadly. We're appealing to a generalization of it here. We'll return to a discussion of it in the specific context of the Semantic Web in I.3.

In Parts II, III, and IV of this book, we explore the language hierarchy that we have just defined. We will start with the smallest class, the regular languages, and move outwards.

[5]Quoted from [Berners-Lee and Mendelsohn 2006].

3.4 A Tractability Hierarchy of Language Classes

The decidable languages, as defined above, are those that can, *in principle*, be decided. Unfortunately, in the case of some of them, any procedure that can decide whether or not a string is in the language may require, on reasonably large inputs, more time steps than have elapsed since the Big Bang. So it makes sense to take another look at the class of decidable languages, this time from the perspective of the resources (time, space, or both) that may be required by the best decision procedures we can construct.

We will do that in Part V. So, for example, we will define the classes:

- ***P***, which contains those languages that can be decided in time that grows as some polynomial function of the length of the input,
- ***NP***, which contains those languages that can be decided by a nondeterministic machine (one that can conduct a search by guessing which move to make) with the property that the amount of time required to explore one sequence of guesses (one path) grows as some polynomial function of the length of the input, and
- ***PSPACE***, which contains those languages that can be decided by a machine whose space requirement grows as some polynomial function of the length of the input.

These classes, like the ones that we defined in terms of particular kinds of machines, can be arranged in a hierarchy. For example, it is the case that:

$$P \subseteq NP \subseteq PSPACE$$

Unfortunately, as we will see, less is known about the structure of this hierarchy than about the structure of the hierarchy we drew in the last section. For example, perhaps the biggest open question of theoretical computer science is whether P = NP. It is possible, although generally thought to be very unlikely, that every language that is in NP is also in P. For this reason, we won't draw a picture here. Any picture we could draw might suggest a situation that will eventually turn out not to be true.

Exercises

1. Consider the following problem: Given a digital circuit C, does C output 1 on all inputs? Describe this problem as a language to be decided.
2. Using the technique we used in Example 3.8 to describe addition, describe square root as a language recognition problem.
3. Consider the problem of encrypting a password, given an encryption key. Formulate this problem as a language recognition problem.
4. Consider the optical character recognition (OCR) problem: Given an array of black and white pixels and a set of characters, determine which character best matches the pixel array. Formulate this problem as a language recognition problem.
5. Consider the language $A^nB^nC^n = \{a^nb^nc^n : n \geq 0\}$, discussed in Section 3.3.3. We might consider the following design for a PDA to accept $A^nB^nC^n$: As each a

is read, push two a's onto the stack. Then pop one a for each b and one a for each c. If the input and the stack come out even, accept. Otherwise reject. Why doesn't this work?

6. Define a PDA-2 to be a PDA with two stacks (instead of one). Assume that the stacks can be manipulated independently and that the machine accepts iff it is in an accepting state and both stacks are empty when it runs out of input. Describe the operation of a PDA-2 that accepts $A^nB^nC^n = \{a^nb^nc^n : n \geq 0\}$. (*Note*: We will see, in Section 17.5.2, that the PDA-2 is equivalent to the Turing machine in the sense that any language that can be accepted by one can be accepted by the other.)