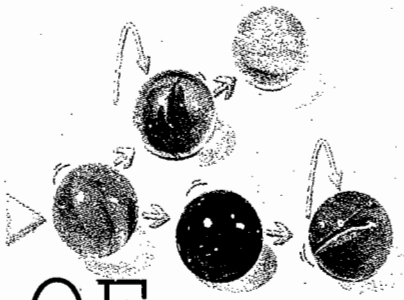


THEORY OF COMPUTATION

A Problem-Solving Approach

Dr Kavi Mahesh
Professor, Centre for Ontological Engineering,
Department of Computer Science, PES Institute of Technology, Bangalore
and
Principal Consultant, Knowledge Management Group, Infosys Ltd.

WILEY-
INDIA

Acknowledgments

The idea of writing a book on this subject by taking a problem-solving approach was conceived over four years ago. This book would not have been possible without constant encouragement and support of many good friends, colleagues and members of my family over these years. I have benefited greatly from valuable reviews and suggestions for the book provided by a number of people, in particular Prof. H.P. Khincha, Prof. H.S. Jamadagni, Prof. K.N. Balasubramanya Murthy, Prof. Ramamoorthy Srinath, Prof. N.S. Kumar, Dr. J.K. Suresh and Mr. D. Krupesha. I would like to thank in particular Prof. Srinath and Prof. Kumar for their detailed reviews of several chapters. Endless discussions over the years with Professors Nitin V. Pujari, R. Srinath and N.S. Kumar have helped me fine tune my understanding of many subtle points.

I would like to express my sincere thanks to the principal and management of PES Institute of Technology for creating the right academic environment and for providing constant encouragement and support that made this book possible. In particular I would like to thank my students and teaching assistants, in successive years, who have helped me in compiling class notes and designing examples and exercises: Tejas Dinkar, Harshita Agarwal, Shreegiri Hiremath and Shilpa Pai have been of great help.

I would like to thank the highly efficient and professional team at Wiley India. It has been a real pleasure to work with them, in particular with Praveen Settigere, Meenakshi Sehrawat and her editorial team.

The developers of the JFLAP open-source software package at Duke University have made a wonderful contribution to the study of automata, formal languages and grammars. I would like to thank them for enabling me to create non-trivial examples and diagrams accurately and efficiently.

Finally, I would like to thank Vani and the kids for putting up with me through the last year when I spent so much time with my Netbook that I hardly ever saw them.

I would like to sincerely request readers to send information about any errors in the book or suggestions for making it better to DrKaviMahesh@gmail.com.

Kavi Mahesh
Bangalore, India
November 2011

Contents

RegEx	Regular expression
TM	Turing machine
UTM	Universal Turing machine
VLSI	Very large scale integration
XML	eXtensible Markup Language

Chapter One

Computers and The Science of Computing

Learning Objectives

After completing this chapter, you will be able to:

- Learn to distinguish between computing machines and other kinds of machines.
- Learn what it means to compute.
- Learn the relationships between computing machines, problems, languages and grammars.
- Learn to define computer science.
- Appreciate the relationships between the science of computing and a core set of subject areas in computer science.
- Appreciate the historical development of the science of computing.
- Get an introduction to computability, intractability and intelligence.
- Understand the reasons for studying the science of computing.

This chapter introduces the science of computing. It begins by exploring what is and what is not a computer and what the very idea of computing is. The chapter also provides a preview of the rest of the book by introducing, albeit informally, key ideas of computing such as computability, decidability and intractability. The chapter also includes the story of the birth of computer science. More importantly, the chapter tells the student why he or she must study the present subject.

1.1 What is (Not) a Computer

The word computer brings to mind a digital electronic machine with a display, a keyboard and a mouse. However, there is no need for a computer to be like a modern digital electronic device. Interestingly, a computer need not have any moving parts, electrical power, display unit, arithmetic logic unit or even memory. It is just the case that most present-day computers have such parts. Moreover, most smart phones, video game players, GPS devices, digital tablets, point-of-sale machines and even some car entertainment units have more computing power than the best computers that were available 10–15 years ago! Yet, we don't often refer to such devices as computers.

What then is a computer? What is the most generic form of a computing machine? Most people when asked this question say that a computer is a machine that accepts input, processes it and produces output. There is a problem, however, with such a definition: juice mixers, kitchen grinders and washing machines also accept inputs, process them and produce outputs. Are they computers?

Some people enhance this definition by saying that a computer has memory, that is, a storage unit. However, dishwashers, vacuum cleaners and wet grinders also have storage units! Are they computers?

One big difference between computers and these electrical appliances is that the latter do not make any significant decision. For example, if we try to wash bricks instead of clothes or try to grind stones instead of rice and lentils, the washing machine or the grinder tries to wash or grind them anyway. A computer, on the other hand, makes significant decisions about its inputs; it can even reject an input. As such, *a computer is a machine that accepts inputs, processes them to produce outputs, and is capable of storing and making decisions about its inputs and outputs.*

In this subject, we study models of abstract computing machines rather than the architecture and design of a modern digital computer. Our objective is to understand the essential nature of a computer and of computing. What are the essential parts, operations and properties of a computing machine? In abstract terms, we can define a car as a machine that takes passengers from one location to another by moving on roads while consuming a fuel and being controlled by a human driver. Similarly, a computer is a machine that *computes* an output for a given input while being controlled by a *program* or *table of instructions.* Just as a typical car has so many parts and functions in addition to its core essentials, so has a typical computer many other parts such as a display, a keyboard, a mouse, a memory unit, a hard disk and so on, and additional functions such as printing and network connections. In this book, we study the essential core of a computing machine known as an *automaton* (*automata,* in plural).

1.2 *The Idea of* Computing

The process of transforming the input of a computing machine to its output is called *computing.* It usually involves storing some of the input, output or their other intermediate forms in memory. It also involves making decisions during processing.

The input to a computing machine is assumed to be provided in the form of a single string of symbols. The symbols are taken from a set of all possible input symbols known as the input *alphabet.* For example, the ASCII character set is an alphabet. To keep matters simple, we often consider much smaller sets as alphabets in the examples in this book. For example, $\{0, 1\}$ and $\{a, b\}$ are the most common alphabets used to illustrate the concepts in this book.

A string is a sequence of zero or more symbols from the alphabet. A string can be of any length. The string of length zero is a special string and is denoted by the symbol λ. The computing machine is expected to process the input in a single left-to-right pass, that is, it starts with the first symbol in the input string and processes the input string one symbol at a time until the entire input string is processed. It is assumed that the machine is not allowed either to look ahead and see what symbols are coming later or to go back and re-examine or re-process symbols which are already processed once. We will make an exception to this constraint only in Chapter 10. When the machine examines a symbol and processes it, we say that the input symbol is consumed by the machine. Figure 1.1 shows an abstract computing machine computing its output from its input.

What about the output? Normally, we are interested in only a binary, yes–no, or Boolean output from the computing machine. It either accepts the input string or rejects it. As such, a typical automaton does not produce any real output.

A variant of the regular automaton, known as a *transducer*, produces real output. In a transducer, the output is a string of symbols similar to the input string. We do not consider transducers in this book except in Section 3.6 in Chapter 3 and later in some sections of Chapter 12. Even when we consider examples such as addition or multiplication, we are usually not talking about transducers that actually produce the sum or the product as output (e.g., the kind of devices studied in electronic circuits and

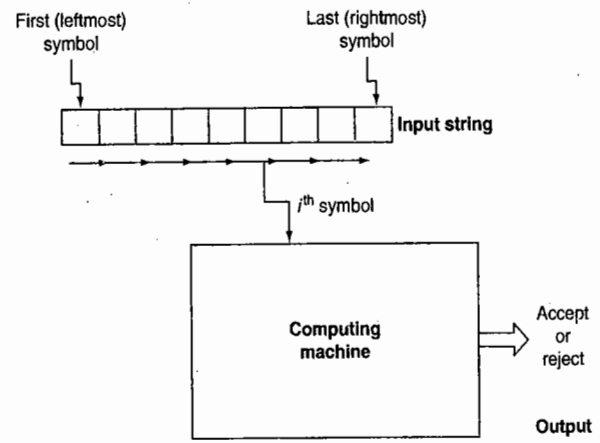


FIGURE 1.1 A computing machine.

computer architecture). We will see below how this is not a limitation of the idea of computing being presented here.

In the kind of computation that we study, there are no side effects produced by processing an input string. The automaton is assumed to come back to a state identical to its original state. Nothing is remembered from the computation or the input string once the output is produced and the computation is over. There is no persistent change to the computing machine. As a consequence, the output of the machine is purely a function of the input. Prior history or other external factors have no role to play in whether the machine accepts an input string or not. In other words, computation, as we consider it here, is like how a function is executed in a pure functional programming language.

In computing the output or, in other words, while processing the input string, the automaton changes its internal state similar to how we change our mental states. The state of a computing machine is nothing but a set of behaviors that it exhibits at a certain point in its computation, and these behaviors are determined by what the machine remembers about the part of the input string that it has processed up to that point in the computation. In general, we are interested only in computing machines that have a finite number of states, as we will see in Chapter 2.

For the most part, we do not consider how fast or efficient a computation is. We do not even attempt an abstract measure of time or cost, for example, in terms of the number of steps involved in the computation (except at the end of the book in Chapter 12). Rather, we are in general interested only in studying whether the computing machine *can* compute an output for a given input. We study different kinds of computing machines that can compute outputs for corresponding kinds of inputs.

1.3 Computing Machines and Languages

Why aren't we talking about computers solving problems or calculating numbers? Instead, why talk about computing machines processing input strings? Any (computing) problem can be stated in the following two ways:

1. As a conventional problem: "Given ... as inputs, ... find/calculate/compute ..." where the computer has to calculate or compute the output number(s) or value(s) (and print it out, store it or display it).

2. As a yes-or-no (or true-or-false) question: "Is ... the correct answer for ...?" where the computer has to calculate or compute the output number(s) or value(s) and compare it with the answer already provided and say "Yes" (or "True") or "No" (or "False").

The second kind of problem is very convenient in describing and understanding abstract computing machines. It also eliminates the need to add output devices such as display monitors and printers to our basic computing machines. Furthermore, it helps in grouping problems of comparable levels of difficulty into *sets of input strings*.

Any set of input strings is called a *formal language* (see Chapter 4 for more details). For example, the problem of addition, or the set of all problems in adding two positive integers, can be considered a set of strings of the form "Is z the sum of x and y?" In other words, the set of all such strings, some of which are true (when $z = x + y$) and others are false (when $z \neq x + y$), represents the entire problem of addition and is considered a (formal) language. Such equivalence between types of problems and formal languages is used throughout this book. Sometimes, the words *problem* and *language* are even considered interchangeable.

Sometimes, we also talk about mathematical functions and the idea of computing a function. Again, a function $y = f(x)$ can be converted to a Boolean (or True–False) function by including the result of the function as an additional argument to the function, $f'(x, y)$, thereby converting it to a string. Thus, a function can also be considered a set of strings and therefore computable functions too are considered equivalent to problems and languages.

Why are they called languages?

A language such as English can be considered to be nothing but the set of all valid words, phrases, clauses and sentences in English. That is, even a human or natural language such as English is nothing but a set of strings! Moreover, not all possible strings are in the language; there are invalid or ungrammatical strings that do not belong to English. Since the sets of strings that we consider in the context of computing machines are not natural to humans, we call them *formal languages*. Any language also has a *syntax* (or grammar for its patterns) and *semantics* (meaningfulness). For example, English sentences have a syntactic structure that is controlled by the grammar of English and its words, phrases and sentences have meanings associated with them. In the case of formal languages, it appears as though we are primarily concerned with the syntax or structure and patterns of the strings. The symbols we use may have meanings in an application or in the real world. Yet, we treat computation as merely a manipulation of symbols from the input string to the output.

The languages that we consider are usually infinite languages, that is, sets with an infinite number of strings in them. For example, the set of all positive integers is an infinite set; there is no upper limit on the value of an integer, in theory, although in any implementation of integers in a programming language, there is usually such a limit making the set of integers a finite set. Although it is impossible to ever store or process an infinite set in a computing machine with a finite amount of memory, we are concerned with infinite sets because, just like in mathematics, we do not want to impose any arbitrary upper limit on the value of a number or the size of a string in a language. Also, we will learn that finite sets are not very interesting from the point of view of computing; they are rather simple. Only infinite sets pose interesting challenges to various types of computing machines.

It must also be noted that finite does not mean a small or known number. A finite number can be very large and very difficult to determine as well. For example, the total number of oxygen molecules in the

Earth's atmosphere is finite. We may never be able to count or otherwise determine this number. Nevertheless, it cannot be infinite.

Strings in a formal language are made up of symbols from its alphabet. The symbols themselves can be anything; each symbol merely needs to be unique, distinct from all other symbols in the alphabet. The same symbol can appear any number of times in a string. Unlike in programming languages, there are no other data types such as integers and real numbers. Everything is a string and every string is simply a sequence of symbols. A string can be of infinite length, again in theory, although only a finite string can ever be stored or processed in a computing machine. In fact, if we do not include strings of infinite length, every set of strings will also become finite and thus all formal languages also become finite.

How do we describe a formal language? It is not always convenient to list all the strings in the language and often not precise enough to describe in words the patterns in a language. Hence, we borrow the idea of a grammar from natural languages such as English (i.e., from the field of linguistics) and develop precise ways of specifying a formal language using a grammar. A grammar is essentially a set of rules that specify how symbols come together in various ways to constitute valid strings in the language.

In this book, we study the three basic ideas of languages, grammars and automata. We begin with the simplest kinds of computing machines called *finite automata* (Chapters 2 and 3) and the corresponding formal languages (Chapter 4) and grammars (Chapter 5) called *regular languages* and *regular grammars*, respectively. Finite automata are so primitive that they do not have any memory unit at all. They can remember only some properties of the input string by being in a particular state or by changing states. The idea of enhancing memory capabilities and thereby the computing power by adding memory in the form of a stack is considered next (Chapter 8) after introducing correspondingly more powerful grammars and formal languages known as *context-free grammars* and *context-free languages* (Chapters 7 and 9). In between, we also show that not all formal languages are regular (Chapter 6). After introducing the all powerful class of computing machines called *Turing* machines (Chapter 10), we finally develop a complete hierarchy of machine–language–grammar combinations (Chapter 11). The last chapter introduces some of the limitations of the idea of computing and demonstrates that there are problems that cannot be solved (Chapter 12).

1.4 What is the Science of Computing?

The science of computing asks and answers the following questions:

1. What does it mean to compute?
2. What is computable, what is not?
3. What is a computer or a computing machine? What is the essential nature of a computer?
4. What determines how powerful a computer is?
5. What is computable by which kind of machine?
6. What happens if we take memory out from a computer?
7. What happens if we change Random Access Memory to special types of memory (e.g., a stack)?
8. What classes of problems can a computer (be programmed to) solve?
9. Are there unsolvable problems?
10. What are the limitations of the very idea of computing?
11. What is the relationship between types of computers and classes of mathematical functions?
12. What is the relationship between languages, grammars and computing devices?

13. How fast or efficient can a computation be?
14. To what extent can we parallelize a computation?

Computing has a science of its own. Computing phenomena and questions such as the above are not explained by traditional sciences such as physics and chemistry. The science of computing is based essentially on the model of a computing machine. Computing science is all about the implications of certain assumptions made in the model of computing for computing phenomena. An understanding of the science of computing enables us to answer computing questions such as

1. What is the minimal number of steps involved in sorting a set of numbers? Why does it take that many steps?
2. Why are trees more efficient than arrays for searching?
3. Why can't we do binary search on a linked list?
4. Why does sorting take the same number of steps no matter whether we sort:
   (a) Heavy or light objects?
   (b) Solids, liquids, colors or names?
   (c) On Earth, in outer space or in the Andromeda Galaxy?

Clearly, the science of computing must explain how computing phenomena such as the above are beyond physics, are unaffected by changes in physical parameters. The computing model on which the science of computing is based comes from Alan Turing's seminal contribution to computer science, namely, the abstract yet all powerful Turing machine. This model of computing involves a computing machine or automaton that has a control unit, an optional memory unit with a reading head and the ability to accept an input string and compute an output as shown earlier in Fig. 1.1 (although we must note that the computer shown in Fig. 1.1 has no memory unit). In reality, we may also need a model of information and a model of communication, among others, but our focus here is only on the model of computing.

This model makes the following six fundamental assumptions:

1. **Stored table of instructions:** The general-purpose computer computes by executing a stored table of instructions (which in modern terms we call a compiled or executable program) that is loaded onto its memory.
2. **Linear sequence of memory units:** Its memory consists of a linear array of memory cells; each memory cell can store a single symbol; it has only two neighboring cells: the one to its left and the one to its right; there is no other structure or organization to the memory.
3. **Single reading head:** There is a single read–write head that can read the symbol in the memory cell to which it is pointing so that the control unit can make a decision on the basis of the symbol; it can also write a single symbol into the current cell; it can move to the immediate neighboring cell to the right or left of the current position; these are the only memory operations that the computing machine can perform.
4. **Behavior depends only on current configuration:** The control unit can decide to remain in the current state or to change to a different state; it can also decide to write a particular symbol onto the memory cell at which the reading head is pointing; furthermore, it can decide to move the reading head to the left or right by one cell; these are the only behaviors and they depend only on the current state and the current symbol in the input and the symbol in the current memory cell; previous history of computation has no effect on the present behavior except in the form of a change to a particular state; the computing machine neither remembers nor learns from previous computations.
5. **Minimalistic instruction sets:** The above are the only operations or instructions that the computing machine can execute; every computation must be expressed through these basic operations; there are no built-in "high-level" instructions for various computable functions.
6. **No awareness:** The computing machine has no awareness of what it is doing or what it just did (which can be compared with human consciousness); it has no "big picture" of its computation; it blindly and mechanically executes the table of instructions with no awareness, intuition or common sense.

While the above constraints made it possible to build real computers using electronic hardware, they have also been the major reasons for the limitations of computing machines whether in terms of efficiency or in terms of being unable to exhibit intelligent behaviors.

In summary, we can define the *science of computing* or *computer science* as a science that explains computing phenomena that cannot be explained by traditional sciences.

The science of computing is often also called *Theory of Computation* or *Theory of Computing* or even *Theory of Computer Science*. The theory is somewhat mathematical and formal. At the same time, it is full of challenging problems (almost like puzzles). This subject is what makes computer science different from *software engineering* or *information technology*.

With the above assumptions, the model of computing that is at the center of the science of computing behaves in ways very different from human behavior in similar situations. Computing machines do not think the way we think; they do not communicate the way we do; they do not perceive, comprehend or interpret the world the way we do; and they do not understand, learn or evolve the way we do. Moreover, we will see in later chapters that what is computable for us is often very different from what is computable for them.

Also because of the assumptions, computers often fail to meet the requirements and expectations of their users, thereby raising questions such as

1. Why computers have all our pictures in high definition but cannot recognize any of us?
2. Why computers can be our accountant but not our secretary?
3. Why computers can calculate so accurately but still have errors (as in "computer errors")?
4. Why computers have entire dictionaries but don't understand a word?
5. Why computers need programming?
6. Why computers can run medical systems but can't cure themselves of viruses?
7. Why a Web search engine does not distinguish mouse (the animal) from mouse (the computer gadget)?

While it is beyond the scope of this book to answer all such questions, a clear understanding of the science of computing is essential to answer them. It is also important to remember that the capabilities and limitations of a computer heavily depend on the underlying computing model. Let us consider for a moment possible alternatives (none of which can be readily implemented today) such as

1. Multi-dimensional memory so that every memory cell has many immediate neighbors.
2. Self-aware processors that "naturally" remember where they have been and "know" what they are doing.

3. Human-like short-term memory so that the computer knows "naturally" what it just did or where (in which state) it had just been.
4. Data-centric model, not a processor-centric one, wherein instead of specifying a table of instructions for processing the input we specify input and output data patterns.
5. Associative memory so that any data element stored in memory can be retrieved instantly without an expensive search or indexing mechanisms.
6. Non-deterministic processing or truly unlimited parallel processing or the ability to always guess the right choice! (See Chapter 3.)

The science of computing would be very different with any of the above models! The present science of computing though is derived from the present model of computing based on Turing's computing machine that has none of the above capabilities. However, some significant changes are beginning to take place in computer science. For example, with the enormous growth of Internet content and social networking platforms, it is becoming increasingly infeasible to use regular data structures and algorithms to handle the kinds and amounts of data involved. A social networking platform can never apply any well-known graph algorithm to find the most optimal choices to recommend friends to a user, when the entire network contains hundreds of millions of users.

In computer science, we are especially concerned about the implications of the science of computing for the following core areas of computer science (some of which we explore very briefly in the following section):

1. Data structures.
2. Algorithm design and analysis.
3. Programming.
4. Computer organization.
5. Operating systems.

On top of the science of computing, we also need proper engineering of computer systems and solutions. Such engineering involves practical knowledge of scalable systems, tools and technologies, domains and applications. Like other traditional branches of engineering, it is concerned with the qualities of a (computer-based) solution such as robustness, usability, reliability, longevity, "idiot proofing," scalability, performance, economy and security.

1.5 Programming, Data Structures, Algorithms and the Science

Given that the fundamental model of computing is a general-purpose computing machine with a minimal instruction set, most computations require a suitable table of instruction to be prepared (which can also be seen as the design of an automaton for the particular computation). This task is called programming and is usually done in a "high-level" programming language that provides many high-level instructions and "built-in functions." Such programs need to be translated or compiled to the minimal instruction set of the underlying computing machine.

Specifying a table of instructions for a typical real-world application of computing is non-trivial. Various principles and methodologies of software engineering have to be applied to develop programs accurately

and efficiently. At the same time, since most computations involve non-trivial amounts of data and data processing, the simple model of memory in the underlying computing model with no structures or objects of any kind is not convenient for managing large amounts of data. It may be noted, however, that one particular data structure – a stack – is used in the kind of computing machine we study in Chapter 8. A variety of "bigger" data structures and database technologies have been developed that are easier to map to real world entities. Ultimately, however, all data structures are compiled to the underlying machine so that the execution of the computation happens in the simple linear memory of the computing machine.

Various techniques and strategies have also been developed for solving different computing problems. These techniques, known as *algorithms,* provide efficient ways of computing a solution to a problem or for processing a particular kind of input. Algorithms, their design and analysis are essential for building scalable, efficient and robust solutions to computing problems. However, we must remember that algorithms are a high-level construct, a method or procedure meant for human comprehension; for the underlying machine, the algorithms that are implemented in the programs are actually translated to a table of instructions that are executed by the computing machine. One of the greatest contributions of Alan Turing was showing (albeit without a formal proof) that his universal computing machine can compute anything that can be computed.

Although we talk about general-purpose computers here in this introduction, the following chapters deal exclusively with automata designed to solve specific problems that typically use small amounts of data (Chapters 2, 3 and 8). These machines cannot compute anything other than the one function they were designed to compute. They are as though, for example, an adding computer cannot multiply and a multiplying computer cannot subtract! We return to the idea of a general-purpose computing machine only in Chapter 10.

1.6 Birth of the Science of Computing

Is it possible to answer every question in the world? Is there a universal oracle that can answer any question? Can all problems be solved? Is there a systematic way to prove all possible theorems in mathematics? Can everything be computed? Can we say for sure whether any given logical statement is true or false? Mankind was looking for an all-powerful solution to answer these questions until, early in the 20th century, Kurt Gödel showed that it was impossible to determine the truth (or falsehood) of an arbitrary statement. His famous *incompleteness theorem* showed that any formal system (such as number theory or formal logic) necessarily includes statements that can be neither proved nor disproved within the system. This result continues to have a profound impact on science, mathematics and philosophy.

Soon after the incompleteness theorem was published, Alan Turing who is considered the father of computer science, in response to Hilbert's grand challenge known as the *Entscheidungsproblem*, showed the reason why incompleteness exists; the reason why not everything is computable. In his famous paper in 1936, *On Computable Numbers with an Application to the Entscheidungsproblem,* Turing not only showed why a computing machine cannot compute certain functions but also illustrated the idea of a general-purpose computing machine with a central control unit (which was a simple automaton) and a memory unit with a reading head. This was the birth of computer science.

It took a couple of decades before Turing's vision of a general-purpose computing machine could be realized in electronic hardware. The rest of the development of computer science – from low-level machine languages to high-level programming languages, the rapid growth in CPU speeds and memory capacities with an equally rapid reduction in the physical size of computing machines, the large-scale engineering of servers and software systems, analysis and design of advanced data structures and

algorithms, and the development of computer networking and the Internet – are all rather well known and need not be elaborated upon here. Strangely, computer science was born of an attempt to show the limitations of computer science! We introduce this topic briefly in the following section and revisit the topic and study the limitations of computer science in Chapter 12.

1.7 Computability, Undecidability, Intractability and Intelligence

Although Turing's model of a general-purpose computing machine can compute anything that is computable, not every function is computable. Even among those that are computable, some are so expensive to compute (e.g., they take too long or demand too much memory) that they are intractable. We will learn in the last two chapters of this book that not all functions are computable and that not all formal languages are enumerable or describable in compact ways using grammars. Turing showed that any computing machine, when asked to compute a function that is not computable, goes into an infinite loop and never halts for some input strings. This problem, known as the *halting problem*, explains why not every function is computable. We will even see examples of practical problems that are not solvable (or computable) in Chapter 12. With reference to formal languages, enumerating a formal language means systematically listing all the member strings of a language, even if it is an infinite set, so that any particular string is listed in finite time. The reason why some languages are not enumerable is that there are "too many" formal languages, many more than there are computing machines (or algorithms) that can process them. There are as many formal languages as there are real numbers but only as many computing machines (or algorithms) as integers. We explore these strange phenomena in Chapters 11 and 12.

Although there are such limitations on the very idea of computing, although there are functions that are not computable and problems that are not solvable, fortunately, most practical problems can be solved through algorithmic computing procedures. Most formal languages that are useful to us are, in fact, enumerable and can be described well using compact grammars.

Among solvable problems and computable functions, not all of them have the same degree of difficulty. Some are inherently harder than others and take more time, or more steps in computation, to arrive at a solution, or to produce an output for an input string. The relative difficulties of problems are studied in a field known as *computational complexity* that is introduced briefly in Chapter 12. One of the key concerns of complexity analysis is figuring out what happens when the size of the problem increases. Will the time required grow slowly in relation to the size so that the computing solution remains tractable or will it grow exponentially in the size of the problem so that the computing solution becomes *intractable*? However, even intractable problems can often be solved efficiently in practice by following an approximate algorithm or by aiming for a locally optimum solution as opposed to a globally optimum solution to the problem.

Even when computers built on Turing's model can compute a function or process a formal language, their behavior rarely can be seen as an intelligent behavior. Computers based on Turing's model, although they are capable of elaborate calculations and intricate symbol manipulations, have remained dumb for the most part in spite of extensive research and development in the area of *artificial intelligence*. It appears to be too expensive in terms of both memory and computation to carry out intelligent computation in this model. What is natural and effortless to us humans is not at all easy for computers: recognizing images, understanding natural languages and so on. Algorithms for natural or intelligent human capabilities and functions (if indeed such functions are algorithmic) have remained elusive. We explore this issue briefly in concluding the book in Chapter 12.

1.8 Why Study the Science of Computing

We conclude the introduction by listing some of the reasons for studying the theory of computing:

1. It constitutes the essential science behind all computing. Knowledge of this science helps one appreciate the nuances and challenges of other subjects such as data structures, algorithms, programming languages, compilers and operating systems.
2. It takes one through the historical development of computing machines, thereby enabling one to truly understand what a computer is. It also helps one separate the essential characteristics of a computer from the myriad specifications and idiosyncrasies of present-day digital electronic computers. One can even imagine other forms of computers of the future.
3. The theory of formal languages came from linguistics and the computational approach that it takes has, in turn, enormously influenced the further progress of language sciences including (computational) linguistics and natural language processing. With the rapid growth of textual content on the Internet, knowledge of formal language and grammar theory is essential for anyone dealing with text processing, information retrieval, search engines or other technologies for knowledge management. Search engine design and application in particular make extensive use of regular expressions and other pattern-matching techniques of formal languages. Another example of a recent application of language processing technology both in electronic commerce and in social networking platforms can be found in text-based recommender systems on the Web.
4. Similarly, the design of compilers and the grammar and algorithmic techniques that they use can be seen as a direct application of the theory of computing. The design of programming languages is another related area that derives extensively from the model of computing and the theory of formal languages and grammars.
5. In addition to programming languages, another type of computer languages, namely, data representation languages have taken prominence with the recent advances in the eXtensible Markup Language (XML) family of language technologies. The design and processing of XML languages in Web and business applications requires proper understanding of the theory of regular expressions and context-free grammars. XML also finds applications in a wide range of application domains including Web Services, Semantic Web, bioinformatics and enterprise application integration.
6. The study of automata originated from attempts in designing automatic machines such as vending machines. Automata theory continues to play a major role in the design and efficient implementation of special-purpose computing machines, kiosks and various cyber-physical systems.
7. Automata and formal language theory also forms the basis for significant aspects of digital and VLSI design, and hardware as well as software testing and verification. The specification of test plans and the generation of automatic testing scripts are but applications of grammars and the derivation of formal language strings from grammars.
8. Finally, it appears that one who has a solid foundation in the theory of computing can cope with the rapidly changing world of new computing technologies and ever more programming languages and paradigms with confidence and efficiency. Even if there is a breakthrough in the science of computing that changes the very model of computing, one who understands the present science well may be expected to make a smooth transition to a new science of computing.

1.9 Key Ideas

1. A computer is a machine that accepts inputs, processes them to produce outputs, and is capable of storing and making decisions about its inputs and outputs. A computer is a machine that *computes* an output for a given input while being controlled by a *program* or *table of instructions*. A computing machine is also called an automaton.
2. The process of transforming the input of a computing machine to its output is called *computing*. It usually involves storing some of the input, output, or their intermediate forms in memory. It also involves making decisions during processing.
3. Inputs and outputs are strings made up of symbols from an alphabet.
4. In general, a computing machine is expected to process its input in a single left-to-right pass.
5. The output is usually binary: the machine either accepts or rejects the input string. If the output is not just a yes-or-no answer, the computing machine is called a transducer.
6. During a computation, an automaton changes its internal state from one state to another.
7. If, at the end of processing the entire input, the automaton halts in a final or accepting state, it is said to accept the input. If it halts in a non-final state, it is said to reject the input.
8. A formal language is a set of strings. A set of related problems can also be considered a formal language. The set of all strings accepted by a computing machine is its language.
9. A formal language is specified by the rules of its grammar.
10. We study classes of formal languages, their grammars and computing machines that accept them.
11. Computing has a science of its own. We can define the *science of computing* or *computer science* as the science explains computing phenomena that cannot be explained by traditional sciences. The science of computing is often also called *Theory of Computation* or *Theory of Computing*.
12. The science of computing makes several significant assumptions about the model of computing. The science is all about figuring out the consequences of the assumptions made in the computing model on computability, data structures, algorithms and programming.
13. Computer science was born out of philosophical and mathematical challenges faced in trying to determine the truth or falsehood of all possible formal statements in mathematics.
14. Not everything is computable. There are limitations to the very idea of computing. Even among computable problems, many are intractable, that is, they take too long to compute.
15. The behavior of computers built on Turing's model rarely can be seen as intelligent. Computers, although they are capable of elaborate calculations and intricate symbol manipulations, have remained rather dumb. It is not clear whether Turing's model of computation is good enough to explain how human beings behave intelligently.
16. Apart from being the foundational theory of computer science, the theory of formal languages and automata has a number of important practical applications in areas ranging from programming language design and compilers to XML, web technologies and hardware and software testing.

Chapter Two

Automata

Learning Objectives

After completing this chapter, you will be able to:

- Recognize that everyday machines such as vending machines are computing devices.
- Learn to design a finite automaton for a given problem.
- Learn to handle end conditions in automata.
- Learn to handle reject states in automata.
- Learn to use states as memory.
- Use a step-by-step method for constructing complex automata.
- Appreciate some limitations of deterministic finite automata.

This chapter introduces the idea of a computing machine. Taking the example of a simple real-life computing machine such as a vending machine, it illustrates the idea of a state transition automaton. The bulk of the chapter shows how to design and construct finite automata to accept or reject various sets of inputs. This chapter presents the first set of *mantras* that will be enhanced throughout the book for different classes of formal languages and automata.

2.1 *The Idea of* a Computing Machine

A computing machine, as noted in Chapter 1, accepts some input and produces some output by processing the input. Usually, it also makes decisions during its processing. For example, a calculating machine or a calculator takes two numbers as input, divides the first by the second and returns the quotient as the output. In the process, it must examine to see if the second number is zero in which case it must not try to perform the division. Similarly, a search engine such as Google takes the user's query as input, processes it against all the pages on the Web, makes complex decisions about which web pages to include in the results and gives an ordered list of web page links as output.

A modern digital computer is a sophisticated computing device. We begin our study of the theory of computing by first looking at the most simple computing machines known as automata. The primitive kind of automata that we study in this chapter is not provided with any memory element at all! Yet, it is able to make decisions and compute its output from an input string of symbols. Let us start with a simple real-world machine and see how it can be described as an automaton.

Consider a ticket vending machine such as the ones we see at railway stations. Let us say the cost of a ticket is ₹3 and we can buy one or two tickets at a time. The machine accepts coins of ₹1, 2 or 5 in any order. It prints a ticket as soon as the amount inserted exceeds either ₹3 or 6. It also returns appropriate change to the user. There are several ways to buy a ticket for ₹3:

1 + 1 + 1 (i.e., three coins of ₹1 each)

or 1 + 2

or 2 + 1

or 1 + 1 + 2

or 2 + 2

Similarly, there are several ways of getting two tickets for ₹6 (assuming that ₹5 coins are to be used only to buy two tickets):

1 + 5 or 5 + 1 or 2 + 5 or 5 + 2 or 1 + 1 + 5 or 5 + 5

The machine starts out by waiting in an idle state. We can say that the machine is hungry for coins in this state. As soon as a coin is inserted, the machine changes to a different state. The machine clearly needs to remember how much money the user has inserted. If the machine were to forget how much money was given to it, there would be some really angry customers at the railway station! The machine remembers the amount by changing to a different state after each coin is inserted, each of the states corresponding to the total amount inserted thus far by the customer.

Figure 2.1 shows the workings of this machine diagrammatically where each circle indicates a different state of the machine. The number in the name of the state shows the total amount inserted by the customer up to that point. For example, if the machine is in state q_2, it means the customer has so

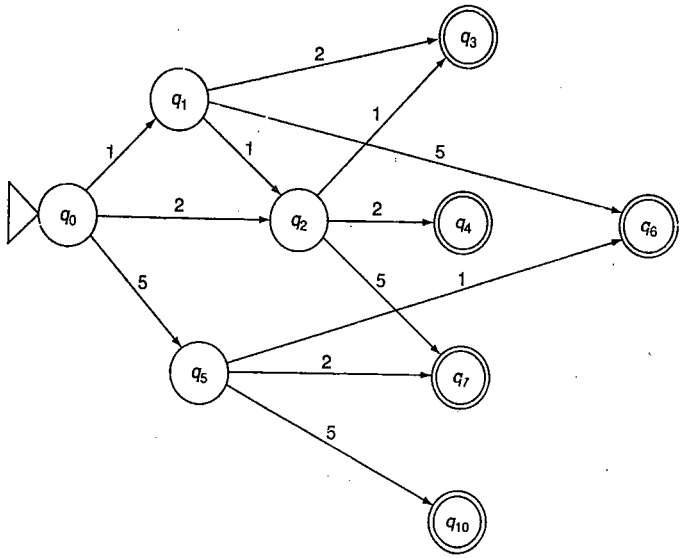


FIGURE 2.1 An automaton for a ticket vending machine.

far given ₹2 to the machine (either a single coin of ₹2 or two coins of ₹1). An arrow from one state to another indicates a change of state and the label on the arrow indicates the denomination of the coin inserted by the customer. The initial waiting state of the machine is labeled q_0 (also called the start state) and marked by an incoming arrow. A change of state is triggered by the customer inserting a valid coin into the machine. You will see that some states – q_3, q_4, q_6, q_7 and q_{10} – are highlighted by drawing two concentric circles. They are called final states; they are the states where the machine prints a ticket and returns appropriate change to the customer. They are all different from one another because either they print different numbers of tickets or the amount of change to be returned is different. For example, in q_4 one ticket is printed and ₹1 is returned while in q_7 two tickets are printed and ₹1 returned. It is assumed that after printing the tickets and returning change to the customer, the machine goes back to its waiting state q_0 and looks forward to the next opportunity to serve tickets.

Such a machine is called an *automaton*. It is a computing machine since it takes input, processes it to produce output and makes decisions along the way. In this example, the machine makes a number of decisions on the basis of the particular coins inserted by the user, apart from determining the validity of the coins inserted. We see examples of such automata in other more sophisticated vending machines, automated teller machines (ATMs), home appliances, laboratory and medical instruments and video game parlors in addition to computers, smart phones and other hand-held digital devices such as GPS navigation gadgets.

Automata are the simplest computing devices. Interestingly, they do not have any memory: no RAM, no flash memory and no hard disks. They do have a rudimentary control unit or "operating system" that can be described as follows:

1. Start in the initial state q_0.
2. Wait for an input symbol.
3. If a symbol is received as input,
   (a) Examine current state for an outgoing arrow marked with the given input symbol.
   (b) If a matching arrow is found, traverse the arrow, change to the state at which the arrow is pointing and consume the input symbol.
   (c) Wait for the next input symbol; continue processing until input is exhausted.
4. When the input is exhausted, if the current state is a final state, the input is declared as accepted; otherwise, the input is declared as rejected by the machine.

Input symbols are supplied to the automaton externally, one symbol at a time in a left-to-right order. The machine neither has the ability to look ahead at what symbols might come next (i.e., to the right of the current symbol) nor it can go back (i.e., to the left) and re-process input symbols that are already processed. As you can see from the above procedure, the automaton merely needs to remember the current state of the machine. There is no need for it to store any other data: no local variables and no data assignments of any kind!

A diagram such as Fig. 2.1 is called a *state transition diagram* where an arrow between a pair of states is called a *move* or *transition*. The transition diagram specifies the complete set of behaviors of the automaton. That is, for each state and for each possible input symbol, it specifies exactly what the behavior of the machine must be, namely, what new state it should reach upon consuming that particular symbol.

2.2 Automata Definition

An automaton M has five elements, which are defined as follows:

1. M.alphabet, denoted by Σ, is called the alphabet and is the finite set of symbols that can appear in the input.
2. M.states, also denoted by Q, is the finite set of all states in the automaton.
3. M.startState, usually denoted by q_0, is the start state of the automaton.
4. M.finalStates, denoted by Q_F, is the subset of M.states that are the final states of the automaton.
5. M.transitionFunction is a function δ from $Q \times \Sigma$ to Q, that is, it is a mapping from the current state and the current input symbol to a new state.

For example, the automaton M for the ticket vending machine shown in Fig. 2.1 can be specified as

1. M.alphabet = $\{1, 2, 5\}$
2. M.states = $\{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_{10}\}$
3. M.startState = q_0
4. M.finalStates = $\{q_3, q_4, q_6, q_7, q_{10}\}$
5. M.transitionFunction = $\{(q_0, 1) \rightarrow q_1, (q_0, 2) \rightarrow q_2, (q_0, 5) \rightarrow q_5,$
$(q_1, 1) \rightarrow q_2, (q_1, 2) \rightarrow q_3, (q_1, 5) \rightarrow q_6,$
$(q_2, 1) \rightarrow q_3, (q_2, 2) \rightarrow q_4, (q_2, 5) \rightarrow q_7,$
$(q_5, 1) \rightarrow q_6, (q_5, 2) \rightarrow q_7, (q_5, 5) \rightarrow q_{10}\}$

It may be noted that in this particular automaton, the final states do not have any transitions going out of them. Each non-final state has exactly three transitions (see Fig. 2.1) corresponding to the three symbols in the alphabet.

These automata are also called *finite automata* to emphasize the fact that they contain a finite number of states. We also assume that the automata are deterministic. That is, for a given state and input symbol, there is only one possible transition. In other words, the transition function is indeed a mathematical function. As such, these simple computing machines are also known popularly as *deterministic finite automata* (DFAs). We relax the determinism constraint in Chapter 3 to study a class of computing machines known as *non-deterministic finite automata.*

A *deterministic finite automaton* (DFA) meets two criteria:

1. Every state has only one outgoing arrow for any given input symbol (i.e., the transition function is indeed a function).
2. No transition can occur without consuming an input symbol, that is, the automaton cannot jump to a new state on its own volition.

An automaton is said to accept an input string if, given the symbols in the string, in order, one at a time, the machine processes it starting from M.startState and halts in any one of its final states (in finite time if the input is of finite length). If the automaton gets stuck in a non-final state, it is said to reject the input string.

We also note that the term *final state* should not be confused with any state in which the machine might finally halt upon consuming the entire input. Such a state can be a rejecting state. *Accepting state* is a better term that indicates clearly that these are the states where the machine halts after accepting the input string.

As a computing machine, the automaton is neither general-purpose nor "programmable." That is, a given automaton can compute only one function. The ticket vending automaton above can compute only whether the coins inserted by the customer are valid so that it can issue one or two tickets of ₹3 each. It cannot be "programmed" to issue movie tickets for ₹45 or to perform addition of integers or to search the World Wide Web, for example. In order to do that, we have to change the states and the transitions in the machine. In other words, we have to construct an entirely different automaton for every new function.

In general, a DFA is an acceptor; that is, given a string of input symbols it either accepts it or rejects it. It does not produce any output as such. However, in practical implementations of automata such as the ticket vending machine above, it is useful for the machine to generate real output, for example, the printing of tickets and return of change. A machine that also produces such output from a final state when an input string is accepted is called a *finite state transducer* or simply a *finite state machine* (see Sec. 3.6).

As introduced in Chapter 1, we study these simple machines because they show us the essential nature of computation. They also help us realize the limitations of such a simple computing mechanism and, in later chapters, we see how providing different kinds of memory capacity to automata increases their computing power or their ability to compute more complex functions.

2.3 Constructing Simple Automata

Let us see how to construct a few simple DFAs.

EXAMPLE 2.1

Our first example is an automaton whose alphabet is the binary alphabet {0, 1}. It should accept any input string that begins with 0 and reject all other inputs (i.e., those beginning with a 1). What is the simplest string that begins with 0? 0 itself. The automaton in Fig. 2.2 meets these requirements. Let us see how.

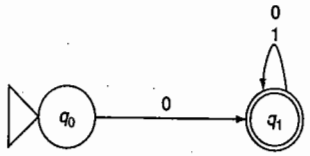


FIGURE 2.2 An automaton for strings that begin with 0.

Our automaton begins in its start state q_0. Since it should accept only strings that begin with a 0, the first input symbol must be a 0. If it is 0, the automaton remembers the fact that the first symbol was a 0 by changing from state q_0 to q_1. Thus, the only arrow going out of q_0 is labeled with a 0. The state q_1 is a final state as shown in Fig. 2.2. The simplest possible input, namely, just a 0, is accepted by the machine because it reaches the final state on seeing the 0. If the input is longer, that is, there are more symbols that follow the initial 0, they all need to be consumed. However, it does not really

matter what they are; the only requirement for acceptance in this automaton is that the first symbol is a 0. As such, the automaton need not remember what the other symbols were. It need not change to any new state upon consuming any of the subsequent symbols. We therefore have an arrow from state q_1 to q_1 itself labeled with both 0 and 1, indicating that no matter whether the subsequent symbols were 0s or 1s, in any order, they are all consumed while the machine remains in the final state q_1. Notice that the loop from q_1 to q_1 can be traversed by the automaton any number of times consuming a corresponding number of input symbols without changing the state of the automaton. This is a useful mechanism to consume parts of the input that do not matter. We sometimes refer to such loops as "jolly rides" to indicate that riding them is in a sense free of cost.

The set of all inputs accepted by this automaton {0, 00, 01, 000, 001, 010, 011, 0000, ...} is indeed an infinite set. Given any member of this set, the automaton will consume all its symbols and halt in its final state. The set of inputs rejected by the automaton is {1, 10, 11, 100, 101, 110, 111, 1000, ...}, also an infinite set. What happens when any of these inputs is presented to the automaton? The machine in state q_0 is expecting only a 0; there is no outgoing arrow for the symbol 1 at all. As such, the machine is stuck in state q_0 itself, a non-final state, thereby rejecting the input.

What do the states in this automaton mean? The initial state q_0 means that it has not yet seen an initial 0. The state q_1 means that the first symbol was a 0; hence it is a final state.

2.4 Handling End Conditions

Let us consider examples of automata wherein the constraint on an acceptable input has to be applied at the end of the input, not the beginning.

EXAMPLE 2.2

Our next example is an automaton that recognizes numbers divisible by 2, that is, even numbers. We are once again dealing with the binary alphabet. As you know, all even numbers in binary end with a 0. We saw above how easy it was to construct an automaton for numbers that begin with a 0. Now let us see why it is a bit more complicated when we want strings to end with a 0. Figure 2.3 shows the automaton that accepts only numbers that end with a 0.

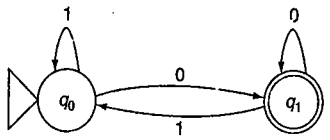


FIGURE 2.3 An automaton for even numbers.

Since input symbols are presented one by one to the automaton, there is no way of knowing which symbol is the last one. In other words, the automaton does not know the length of the input string. It therefore adopts a greedy strategy wherein every time it sees a 0 in the input, it assumes that it is the last symbol in the input and it goes to the final state q_1. If there are additional symbols, the automaton must modify its behavior accordingly. Let us see how.

In the start state q_0, if the input symbol is a 1, it really does not make a difference as far as determining the evenness of the number is concerned. As such the automaton need not remember this fact and it remains in the same state; hence the jolly ride loop from q_0 to q_0 labeled 1. If, on the other hand, a 0 is encountered in the input, the machine greedily decides that it is the end of the input and changes its state from q_0 to q_1, which is a final state. If it so happens that there is (at least) one more symbol in the input, then there are two choices for the automaton. If the next symbol is a 1, the number seen up to this point is not an even number (it is ending with a 1). Therefore, the automaton must in a hurry get out of the final state q_1. Where can it go? Back to the start state q_0 or some new state q_2? The correct choice depends on whether the machine needs to remember that it saw a 1. As before, there is no use remembering that fact for the present problem. Thus, the automaton comes back to state q_0 from q_1; there is no need to introduce a third state q_2. Thus, we have a loop from q_0 to q_1 and q_1 back to q_0. Every time a 0 is followed by a 1, the automaton goes to q_1 and comes back to q_0, looking for another 0, which may be the final input symbol.

If, on the other hand, after reaching the final state q_1 on consuming a 0, one more 0 is seen in the input, the automaton can remain in the final state q_1 since the input, if it were to end, is still ending with a 0. Numbers ending with 00, for example, are (divisible by 4 and therefore) still divisible by 2. Hence the loop from q_1 to q_1 labeled 0. The automaton goes to its final state when it sees a 0 and remains there as long as the input symbols are all 0s. The moment it encounters a 1 in the input, it realizes its mistake in assuming that it had reached the end of the input and hurries back to q_0.

We can trace the execution or computation carried out by the automaton for various inputs as sequences of state transitions ending in the halting state. For example, given 1100, the trace is $q_0 \rightarrow q_0 \rightarrow q_0 \rightarrow q_1 \rightarrow q_1$ (accepted). For 1001101, the trace is $q_0 \rightarrow q_0 \rightarrow q_1 \rightarrow q_1 \rightarrow q_0 \rightarrow q_0 \rightarrow q_1 \rightarrow q_0$ (rejected).

The meanings of the two states in this automaton are as follows – q_0, the start state, says that either we are at the beginning of processing the input or that the last symbol seen was not a 0 (i.e., was a 1). The state q_1 says the most recent symbol seen was a 0. At the end of the input, if the last symbol was, in fact, a 0, the input is accepted since the machine has halted in a final state.

EXAMPLE 2.3

We extend the above example in the automaton shown in Fig. 2.4, which accepts all binary numbers divisible by 4 (i.e., ending with 00). The meaning of q_0 is the same as before that the previous symbol was a 1. State q_1 is also similar to its counterpart in the previous example: we have just found a 0 in the input. However, since ending with a single 0 is no longer sufficient, a new state q_2 is introduced. The meaning of this final state is that the previous two symbols were 0 and 0. If in either q_1 or q_2 the symbol 1 is encountered, the automaton comes right back to state q_0. If additional 0s are present at the end of the input (i.e., if the number is divisible by 8, 16, etc.), the machine takes jolly rides from q_2 to q_2 to remain in that final state.

Another way to explain the meaning of the three states is as follows: q_0 is the halting state for all odd numbers (ending with a 1); q_2 is for all numbers divisible by 4 (ending with 00) and q_1 is the halting state for all even numbers that are not divisible by 4 (ending with a single 0, e.g., 2, 6, 10, ...).

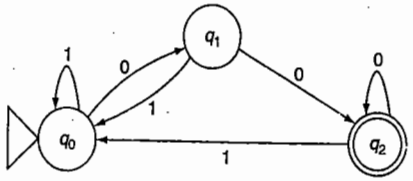


FIGURE 2.4 An automaton for numbers divisible by 4.

2.5 Handling Reject States

Our next example introduces the concept of a reject state, that is, a state introduced explicitly to reject bad inputs, a state that is never reached during the processing of acceptable inputs. A reject state is sometimes also called a *dead state* since the automaton dies in that state; it has no transition to go to any other state once it reaches a reject state.

EXAMPLE 2.4

Let us consider again the automaton of Example 2.2 (Fig. 2.3) but now with an additional specification: numbers with leading 0s are to be rejected (e.g., 0010). The only input beginning with a 0 that is to be accepted is 0 itself. To accommodate this new requirement, we need to split the start state of the automaton in Fig. 2.3 into two states, the new q_0 and q_1, as shown in Fig. 2.5.

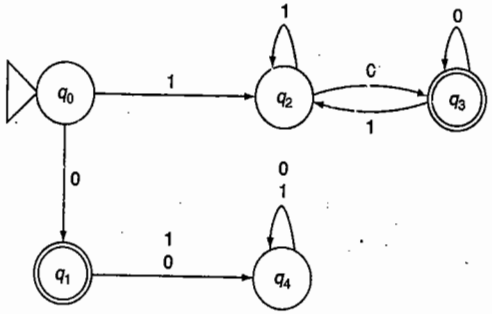


FIGURE 2.5 An automaton for even numbers without leading zeros.

The new automaton has two branches going out of the start state: the normal branch when the input does not begin with a 0 and the other branch for leading 0s. If the input begins with a 1, the machine goes from q_0 to q_2. From then on, q_2 and q_3 and the transitions among them are identical to (the states q_0 and q_1 of) the earlier automaton (Fig. 2.3), q_3 being the new final state for accepting even numbers.

If the input begins with a 0, the automaton reaches the state q_1, which is a second final state that accepts just the number 0. If there are unfortunately any more symbols in the input (i.e., the input really has leading zeros), the machine goes to state q_4 and remains stuck there. Once it reaches q_4, any further symbols are consumed there itself as shown in Fig. 2.5.

The state q_4 is called a reject state since its very purpose is to reject bad inputs (those with leading 0s in this case). The state has no role to play in processing inputs that are accepted (by halting in states q_3 or q_1). The meanings of the other states can be explained as follows: q_0 is simply the start state unlike in the earlier automaton (Fig. 2.3) where q_0 had mixed the start state with the state of having seen an odd number. The new q_2 is the state for odd inputs and q_3 the final state for even inputs. The state q_1 is for a leading 0. It is a final state only if there are no further symbols. If there are further symbols, the automaton takes us to the reject state q_4.

Reject states are often considered optional. If they are omitted, we must ensure that the state from which the reject state would have been reached is not a final state. For example, in the present automaton, q_1 is a final state and if we omit the reject state q_4, for the input 011, the automaton would get stuck in state q_1 that is a final state, thus wrongly accepting the input! In such situations where the automaton must get out of a final state to reject unacceptable inputs, a reject state is essential. It is optional in other automata – for example, in our first automaton in Fig. 2.2, where we could have shown a reject state q_2 that is reached from q_0 upon consuming a 1. A good way to recognize a missing reject state is to find a state for which there is no outgoing arrow on a particular symbol from the alphabet. In Fig. 2.2, there is no transition from state q_0 on symbol 1.

The requirement of not accepting even numbers with leading zeros is a boundary condition and should have been considered earlier in Example 2.2 itself. However, it was not specified initially and hence not considered in the design of the earlier automaton (Fig. 2.3). This is typical of software development where exceptions to mainstream cases and boundary conditions are often not specified in the requirements and often not considered in the design of the software solution, thereby requiring expensive redesign or bug fixing of the software product.

It is interesting to note that our method of trying to analyze the meaning of each state would have helped us in figuring out that something was amiss with state q_0 in the earlier automaton (Fig. 2.3). This could have led us to discover the missing (or ambiguous) specification about leading zeros. We will explore this further in Section 2.10 when we present other such *mantras* for constructing automata.

EXAMPLE 2.5

For the sake of comparison, let us construct an automaton to accept binary numbers without leading zeros that are odd (i.e., they end with a 1). The automaton is shown in Fig. 2.6.

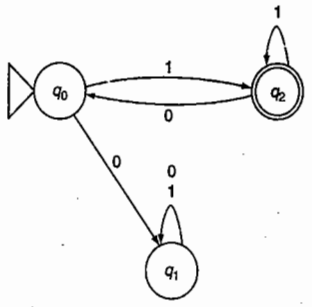


FIGURE 2.6 An automaton for odd numbers without leading zeros: buggy!

There is something wrong with this automaton. It has a bug. It will be a useful exercise for the reader to figure out the bug and fix the automaton. As a hint, try to explain the meaning of state q_0. If that does not help, try to compare this with the automaton in Fig. 2.5.

2.6 A Step-by-Step Method for Constructing Automata

So far, we have been looking at automata that are already constructed and analyzing their behaviors. How do we go about designing or constructing an automaton for a new problem? While there is no exact procedure for this, let us look at a step-by-step illustration and then deduce a set of rules of thumb or *mantras* for designing automata. We have also been using the binary alphabet exclusively. Automata can handle any symbols, not just binary or other numbers. Let us consider an example where the alphabet is $\{a, b\}$.

EXAMPLE 2.6

The requirement is to design an automaton for all inputs with symbols a and b that contain at most three as. There can be any number of bs, but the number of as can be 0, 1, 2 or 3, not more than 3. Let us begin by listing some of the simplest strings that must be accepted by the automaton: a, aa and aaa. How do we construct an automaton that accepts these three inputs? Figure 2.7(a) shows the automaton.

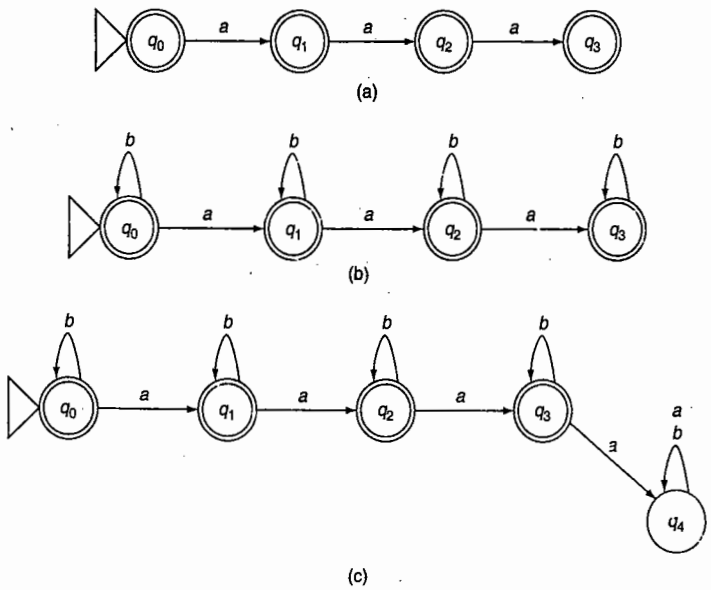


FIGURE 2.7 (a) Partial automaton for strings with three or fewer as. (b) More complete automaton. (c) Final automaton for strings with at most three as.

The state q_1 accepts the input a, q_2 accepts aa and q_3 accepts aaa. However, since q_0 has also been marked as a final state, it accepts a special input, namely, the null input with no symbols at all! The null string is denoted by the symbol λ. Thus, we say state q_0 accepts λ. Nothing else is accepted by the automaton in Fig. 2.7(a).

After constructing an automaton to accept the simplest strings with no more than three as, our next step is to consider what other symbols can occur in acceptable inputs. What about the symbol b? In fact, there is no constraint on where b can occur or how many of them can occur. How do we introduce bs into our automaton? When the symbol b occurs in the input, no matter where in the input it occurs, we observe that the state of the automaton need not change; it need not remember having seen b symbols. Therefore, we can introduce jolly rides or loops on b symbols. Such loops can be added to every state in the automaton since bs can occur before any as, between the first and the second a, between the second and the third a and after the third and last a. Figure 2.7(b) shows the enhanced automaton with these loops. This automaton now accepts all possible inputs with at most three as.

Finally, we need to consider inputs to be rejected by the automaton. When does the automaton reject an input and how? The only reason for rejection is the input having more than three as. In other words, when the input presents the fourth a to the automaton. Such a string is not rejected by the above automaton since there is no transition for the a symbol from state q_3. The automaton will be stuck in state q_3 but it is a final state. To prevent such inadvertent acceptance in state q_3, we need to introduce a reject state q_4 as shown in the final and complete automaton in Fig. 2.7(c).

The states in this automaton have rather simple but clear meanings: q_0 means no a symbol has been seen in the input; q_1 means one a has been consumed; q_2 means two and q_3 means three as have been seen already. The reject state q_4 means four or more as have been seen – too many to accept the input.

In addition to the state transition diagrams that we have been using to represent automata, one can also use *transition tables*. These tables are very concise but less useful in visualizing the working of an automaton. As an example, the transition table for the automaton in Fig. 2.7(c) is shown in Table 2.1.

TABLE 2.1 Transition Table for the Automaton in Fig. 2.7(c)

State	Input = a	Input = b
$\rightarrow *q_0$	q_1	q_0
$*q_1$	q_2	q_1
$*q_2$	q_3	q_2
$*q_3$	q_4	q_3
q_4	q_4	q_4

In a transition table, the start state is marked with an arrow $\rightarrow$ and final states are marked with a star *. Each entry in the second column shows the new state when the symbol a is processed in the state shown in the first column. The third column is applicable when the input symbol is b. We can also observe that every state has a transition for both symbols in the alphabet. Further, no entry has more than one state, thus making the automaton deterministic. Finally, a reject state such as q_4 always has all entries the same as itself, that is, there is no way to get out of a reject state for any symbol. A reject state is a point of no return for the automaton.

2.7 States as Memory

Since automata are not provided with a memory element, the only way for them to remember anything is by changing to a corresponding state. For example, if an automaton has to remember that the first input symbol was a 0 or that the previous symbol was a 1, it has to move to corresponding states. As seen above in Example 2.6, one state was needed to remember that one a had been seen, another state to remember that two as had been seen and a third state for three as. A new state is needed to remember every little fact. If the problem had specified that inputs with less than 50 as had to be accepted, we would have needed 52 states in the automaton! As we will see later (Sec. 2.11), this is a rather severe limitation of finite automata. Owing to their lack of other forms of memory, they need too many states to compute certain functions.

2.8 Why a Finite Number of States?

As we saw in several of the above examples, automata can process and accept infinite sets of input strings. For example, the automaton for even numbers can accept an even number no matter how large it is. In general, automata can accept infinitely many inputs because they do not impose any limit on the length of the input string.

Why then do we want our DFAs to have a finite number of states? This is because it simply does not make any sense to have an infinite number of states in an automaton. Finiteness is important because even for computing problems that are complex, we do not want programs that are infinitely long! Program size can be reduced by identifying "regularity," that is, patterns in the input. It is impossible to construct an automaton that has an infinite number of states or infinitely many transitions defined among its states. Every software program or algorithm must be of finite size and so also the number of states in an automaton.

2.9 Constructing More Complex Automata

Let us consider a few more computing problems for which the design of automata is not so straightforward.

EXAMPLE 2.7

Our first such computing problem is to design an automaton that accepts all binary numbers (non-negative integers) that are divisible by 3. In earlier problems such as divisibility by 2 or by 4, it was very easy to recognize the patterns that all acceptable inputs followed (e.g., ending with 0). There is no such apparent pattern in numbers that are divisible by 3. What is the recurring pattern among 0, 11, 110, 1001, 1100, etc., which are all numbers divisible by 3? None!

Let us first build the automaton to accept 11, the simplest possible number divisible by 3. We can easily build an automaton to accept 11: go from q_0 to q_1 on consuming the first 1 and from q_1 back to q_0 on consuming the second 1 [as shown in Fig. 2.8(a)]. The initial state q_0 is an accepting state. We need to change states in both cases because it is important to remember that the symbol 1 has been processed. This way the automaton will not accept fewer or more 1s (since both 1 and 111 are not divisible by 3). Why did we choose to come back to q_0 instead of going to a new state? We came back to q_0 and created a loop because we observe that not only 11 but also 1111, 111111, 11111111, etc. are all divisible by 3 (but not 111 or 11111, etc., that is, $2^n - 1$ is divisible by 3 whenever n is even).

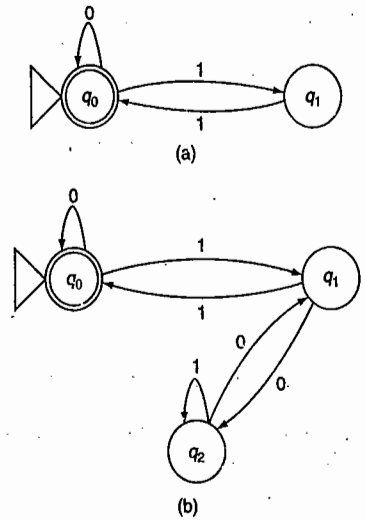


FIGURE 2.8 (a) Partial automaton for binary numbers divisible by 3. (b) Complete automaton.

What do we do about other numbers divisible by 3 such as 110, 10010, etc.? Adding zeroes at the end doubles the number and thus maintains divisibility by 3. This is easy to accommodate by adding a loop from q_0 to q_0 labeled 0. Notice that this automaton will also accept numbers such as 11011, 1101111, etc. that are indeed divisible by 3.

We are left with numbers such as 1001, 10101, etc. How to handle them will become clear if we try to explain the meanings of the states in Fig. 2.8(a). What is q_1? This state is reached when an odd number of 1s has been processed. What is its significance in terms of divisibility by 3? Does it not represent the state where the remainder of dividing by 3 is 1? State q_1 is reached for inputs such as 1, 111, 1101, etc. each of which produces a remainder of 1 when divided by 3. If q_1 is for remainder of 1 and q_0 is for remainder of 0 (i.e., the number is divisible by 3) should there not be a state where the remainder is 2, the only other possible value for the remainder when dividing by 3? Introducing this as state q_2 gives us the final automaton shown in Fig. 2.8(b).

We get a remainder of 2 when the input is 10, 1000, 1110, etc. but we also get a remainder of 2 for inputs such as 101, 11101, etc. All of them can be accommodated by introducing the 0 transitions between q_1 and q_2 and the loop on 1 in state q_2.

All these could have been easily designed if we had somehow realized to begin with that we should think of states in terms of different remainders (i.e., in terms of the *mod* 3 operation). There are only three possible values for the remainder: 0, 1 and 2. Thus, there are only three states in this automaton, one for each value of the remainder. Next, we figure out the transitions among the states by reasoning as shown in Table 2.2. Note that when a 0 is added at the end of a binary number, it gets doubled while adding a 1 makes its value 1 more than double the previous value. From the table, it is clear that these are the only transitions possible in this automaton.

TABLE 2.2 Transitions Among States for Division by 3

From State	Next Symbol	New State and Remainder
q_0 (n mod 3 = 0)	0	q_0 ((2n) mod 3 = 0)
q_0 (n mod 3 = 0)	1	q_1 ((2n + 1) mod 3 = 1)
q_1 (n mod 3 = 1)	0	q_2 ((2n) mod 3 = 2)
q_1 (n mod 3 = 1)	1	q_0 ((2n + 1) mod 3 = 0)
q_2 (n mod 3 = 2)	0	q_1 ((2n) mod 3 = 1)
q_2 (n mod 3 = 2)	1	q_2 ((2n + 1) mod 3 = 2)

EXAMPLE 2.8

Let us see if we can extend this approach to a slightly more complex example: Constructing an automaton for binary numbers divisible by 5. There are now 5 possible values for the remainder: 0, 1, 2, 3 and 4. Let us construct the automaton mechanically by creating five states that correspond to these remainder values and adding transitions by reasoning about what happens to the remainder when either a 0 or a 1 is the next incoming symbol. Figure 2.9 shows the complete automaton where the requirement of excluding numbers with leading zeroes has also been handled by separating the start state and the final state (as done before in Example 2.4 and Fig. 2.5). Table 2.3 shows the transitions obtained by reasoning about remainder values.

Another way to understand this automaton is through the idea that the remainder (or *mod* 5) is the only information that the machine needs to remember and therefore it needs only 5 states. The actual value of the number does not matter as far as divisibility by 5 is concerned. Also, we can prove that

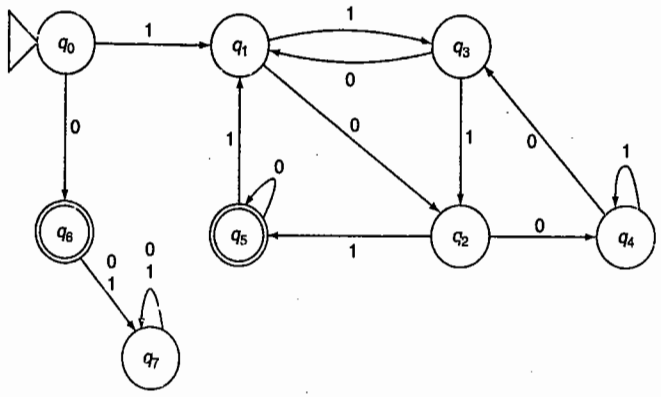


FIGURE 2.9 An automaton for numbers divisible by 5.

TABLE 2.3 Transitions Among States for Division by 5

From State	Next Symbol	New State and Remainder
q_0 (λ)	0	q_0 (0 mod 5 = 0)
q_0 (λ)	1	q_1 (1 mod 5 = 1)
q_1 (n mod 5 = 1)	0	q_2 ((2n) mod 5 = 2)
q_1 (n mod 5 = 1)	1	q_3 ((2n + 1) mod 5 = 3)
q_2 (n mod 5 = 2)	0	q_4 ((2n) mod 5 = 4)
q_2 (n mod 5 = 2)	1	q_5 ((2n + 1) mod 5 = 0)
q_3 (n mod 5 = 3)	0	q_1 ((2n) mod 5 = 1)
q_3 (n mod 5 = 3)	1	q_2 ((2n + 1) mod 5 = 2)
q_4 (n mod 5 = 4)	0	q_3 ((2n) mod 5 = 3)
q_4 (n mod 5 = 4)	1	q_4 ((2n + 1) mod 5 = 4)
q_5 (n mod 5 = 0)	0	q_5 ((2n) mod 5 = 0)
q_5 (n mod 5 = 0)	1	q_1 ((2n + 1) mod 5 = 1)

the machine will accept any number divisible by 5 however large it is because each state corresponds to a particular value of the remainder upon division by 5, and it is a property of binary numbers that appending a 0 (or 1) will change the remainder to a new value exactly as defined by the transition function of the automaton as shown in Table 2.3.

2.10 *Mantras* for Constructing Automata

In Sanskrit, a *mantra* is a statement that when memorized helps you tide over a difficulty. Although there is no exact procedure for designing a finite automaton, let us see if we can list a few mantras for constructing them. When you are asked to construct an automaton for a given problem, repeating these mantras to yourself or asking them as questions to yourself usually takes you in the right direction toward an appropriate design for the problem. In chapters that follow, as we study other more complex computing machines, we will keep enhancing the following set of 10 mantras:

1. *What is the simplest string that the automaton is expected to accept?*
   First construct an automaton that accepts just this string. If you follow this mantra, there is never a problem in how to start constructing the automaton.
2. *What are a few other strings that the automaton must accept?*
   If your automaton does not already accept them, try to enhance it so that it accepts the strings that you have listed.
3. *What are all the other inputs that the automaton must accept?*
   Asking this question and ensuring that all the types of acceptable inputs are handled appropriately will guarantee that the automaton is complete in all respects and that it is not too strict in accepting inputs. In particular, it is worth considering the symbols with which strings in the language begin or end.

4. *What does the machine need to remember?*
   This is an excellent *mantra* to figure out whether the automaton in a given situation needs to change to a different state or remain in the same state (i.e., loop back to the same state). If the input symbol is significant in meeting the requirements of the problem, then the machine must remember it, and it can do so only by changing to a different state. If the input symbol is not significant, the automaton can loop back to the same state on that symbol thereby forgetting that it ever saw that symbol. In fact, whether state change happens or not, the automaton forgets the actual input symbol. It merely marks or remembers a property of the input symbol(s) by changing states.
5. *When the automaton needs to remember an input symbol consumed in a given state, is the transition that we have introduced from that state for that symbol appropriate?*
   Is there already a state to which the machine can transit or do we need to add a new state? Answering this question avoids the introduction of duplicate states and certain errors that result from them.
6. *What is the meaning of each state?*
   Can you express the meaning of each and every state in the automaton concisely and precisely in plain English? If you can do this readily, the automaton is usually the right one. If there are states for which it is difficult to explain meanings, or, for a state, there are two possible interpretations it is very likely that there is something amiss in the design of the automaton. Every state in the machine should have a clear and unique purpose in the computation that the automaton performs.
7. *What are the input strings to be rejected by the automaton?*
   Does it reject all of them or are any of them being accepted unintentionally? Ensuring that all the unacceptable inputs are rejected will confirm that the automaton is not too loose in accepting inputs. In particular, we need to ensure that the automaton does not halt in a final state for an unacceptable input merely because we forgot to put a transition to take it out of that state.
8. *Does the automaton have exactly one transition from every state for every symbol in the alphabet?*
   If so, the machine is complete; if not, we need to add those missing transitions. However, it must be noted that sometimes we leave out transitions that are needed merely to take the machine to a reject state.
9. *Are special cases such as leading zeroes (or other symbols), trailing symbols (e.g., extra zeros after reaching the final state) and the null input λ handled properly? Are they to be accepted or rejected?*
10. *Is the start state also an accepting state?*
    While it is perfectly alright for the start state to be a final state in some automata, it is useful to raise and answer this question for a particular problem in order to avoid errors. Similarly, it is also useful to ask whether there can be outgoing transitions from a final state.

These mantras constitute an approximate method for constructing well-designed automata. Let us apply the mantras to construct a few more automata.

EXAMPLE 2.9

Let us consider the problem of *information retrieval*, the essential problem of a search engine such as Google. In a highly simplified version of this real-world problem, the user specifies a keyword and the machine must find all Web pages or documents that contain the keyword somewhere in them. The pages and documents themselves can be considered as strings so that the problem is essentially one of determining whether a given string contains the keyword as a sub-string of it. Let us say the keyword is 101, the alphabet is binary and the string is given as the input to the automaton. The automaton accepts the input and halts in a final state if the input string contains 101 as a sub-string; it halts in a non-final state and thereby rejects the input if it does not contain the keyword 101. Figure 2.10 shows such an automaton.

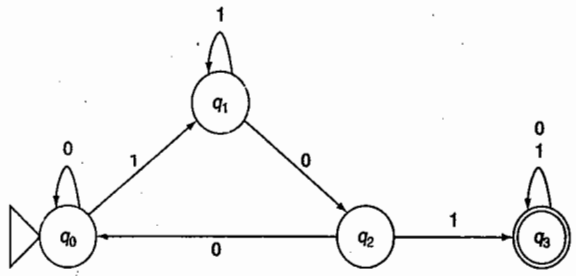


FIGURE 2.10 An automaton to search for the keyword 101.

Applying the already discussed mantras to this problem, the simplest input string to be accepted is 101 itself. The automaton accepts this input by going through the states $q_0 \rightarrow q_1 \rightarrow q_2 \rightarrow q_3$. No input shorter than 101 can be accepted and as such none of the other states can be a final state. What else can be a valid input to this automaton? There can be any symbol, any number of 0s and 1s in any order after the keyword is already found. This is accommodated by having jolly rides (i.e., loops) on both 0 and 1 in the final state q_3.

What should happen in states q_0, q_1 and q_2 when the input symbol is other than what is expected for matching the keyword? For example, in q_0, when the input is 0 and not 1, does the machine need to remember this input? Obviously not, since in that initial state, the particular input symbol 0 does not contribute towards matching the required keyword 101; hence the loop on 0 in state q_0. Similarly in q_2, having already found the part 10 of the keyword, if the input is 0 instead of the expected 1, the machine does not need to remember the particular symbol but it cannot remain in the same state. This is because the input so far is 100, which does not match the keyword 101. Where should it go from q_2? It can come back to q_0 because it has to start afresh looking for the keyword 101 all over again. In state q_1, on the other hand, when it finds a 1 in the input instead of the expected 0, the automaton need not remember the input symbol but it can afford to remain in the same state q_1. The machine is saying that the previous input symbol 1 was not the start of the keyword (since the next symbol did not match the 0 in 101) but perhaps the current input symbol 1 is the start of the keyword 101, so let me continue to stay in state q_1.

We observe that every state has a transition for both 0 and 1 and thus the automaton is complete in all respects. Also, the meanings of the states in this automaton can be expressed clearly:

q_0 means we have not yet found any part of the keyword 101; q_1 means we have found a potential first symbol of the keyword; q_2 means we have found 10; and q_3 means we have successfully found the entire keyword 101 and the input can be accepted no matter what else is there to follow.

EXAMPLE 2.10

Figure 2.11 shows an automaton that accepts all strings over the alphabet $\{a, b\}$ with as and bs appearing in any order, as long as the string contains an even number of as and an even number of bs. Briefly, following our *mantras* again, we can reason that some of the acceptable strings are the null string λ, *aa*, *bb*, *aabb*, *baab*, etc. What does this automaton need to remember? Does it need to remember the total number of as and the total number of bs or merely whether those numbers are odd or even? The total number does not matter, and there are only four combinations of oddness and evenness possible:

1. Even number of as and even number of bs: start state q_0 that is also the accepting state.
2. Odd number of as and even number of bs: state q_1.
3. Odd number of as and odd number of bs: state q_2.
4. Even number of as and odd number of bs: state q_3.

There is no room for any other state. Also, there are no loops from any state back to the same state since every input symbol is significant: any a or b changes the evenness (or oddness) of that symbol and thus the state of the machine. All the transitions are also fairly obvious once we see the meanings of the four states as listed above.

What will the automaton accept if we make q_2 the final state instead of q_0? What if both q_1 and q_3 are final states but not q_0 or q_2?

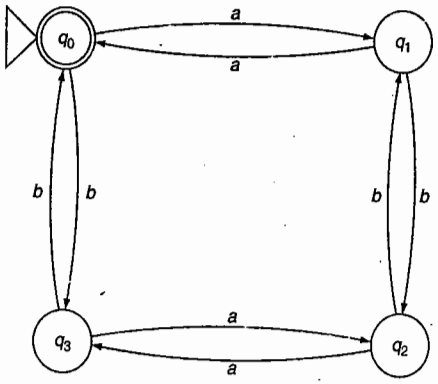


FIGURE 2.11 An automaton for even numbers of as and bs.

The above example shows that trying to count the actual number of symbols in the input is often unnecessary. If only significant properties of the input symbols and their combinations that are relevant to the problem are considered, the number of such combinations is usually finite. A finite automaton can accept the set of strings by mapping each combination of significant properties of the strings to one of its finite set of states. In this example, the only significant property is evenness or oddness of the number of each symbol and there are only four possible combinations of this property.

EXAMPLE 2.11

Our next example is for the set of all input strings where the first symbol in the input is the same as the third symbol and the fifth symbol, and so on. That is, all the symbols in the odd-numbered positions are the same. If the input begins with an a, then the third, fifth, seventh and so on symbols are all a; if it begins with a b, then the third, fifth, seventh and so on symbols are all b. The even-numbered symbols in the input can be either a or b. Figure 2.12 shows an automaton for this problem.

In the start state q_0, the machine clearly needs to remember whether the first symbol was an a or a b. This is a crucial piece of information because this determines what happens later in all odd-numbered input symbols. Hence the machine branches out to two different states: q_1 for a and q_3 for b. This is a typical "branching" automaton where the branches do not merge (except in sharing a common reject state q_5). As long as the input is acceptable, the machine must continue to remember the condition for the branch it is in – the first symbol in this case. Trying to combine the two branches prematurely, for example, by combining states q_2 and q_4, results in errors.

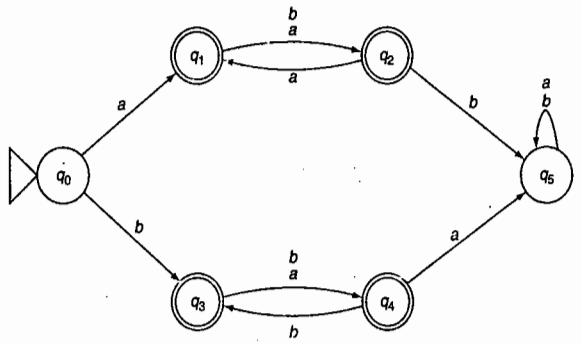


FIGURE 2.12 An automaton for same symbols in all odd positions.

Since the even-numbered symbols do not matter, the transition from q_1 to q_2 and from q_3 to q_4 are labeled with both a and b. The reverse transitions, however, are carefully labeled to match the first symbol in each branch. The meanings of the states of this automaton are as follows:

1. q_0: Start state.
2. q_1: The first symbol was a; we are in the a-branch; the previous odd-numbered symbol was a; the input so far is acceptable.
3. q_2: We are in the a-branch and the input so far is acceptable.
4. q_3: The first symbol was b; we are in the b-branch; the previous odd-numbered symbol was b; the input so far is acceptable.
5. q_4: We are in the b-branch and the input so far is acceptable.
6. q_5: An odd-numbered symbol did not match the first symbol; the input is rejected no matter what other symbols follow.

Notice that q_1, q_2, q_3 and q_4 are all accepting states. This is typical of problems where all inputs except those that violate a certain condition are acceptable. In this case, the input can be of any length, even or odd. Hence both q_1 and q_2 are final states and similarly both q_3 and q_4 are final states. The start state q_0 can also be an accepting state if the null input is to be accepted; here we have assumed that it is not.

2.11 Limitations of Finite Automata

After constructing nearly a dozen automata for various problems, we may be led to believe that these simple computing devices are powerful enough to solve all computing problems. Why then should we bother with more complex devices such as those with extensive memory units? Let us consider a couple of problems which, in the general case, cannot be solved by finite automata and understand the reason for this limitation.

EXAMPLE 2.12

We have seen how an automaton can determine if the numbers of as and bs in an input string over the alphabet $\{a, b\}$ are both even. Can we modify the problem so that this time we want the two numbers to be equal? There should be as many as as there are bs, but the two symbols can appear in any order and the input can be of any length. Figure 2.13 shows an attempt to construct an automaton for this problem. The states are as follows:

1. q_0: Start state and final state where the number of as is equal to the number of bs.
2. q_1: There is one more a than b.
3. q_2: There are two more as than bs.
4. q_3: There are three more as than bs.
5. q_4: There is one more b than a.
6. q_5: There are two more bs than as.
7. q_6: There are three more bs than as.

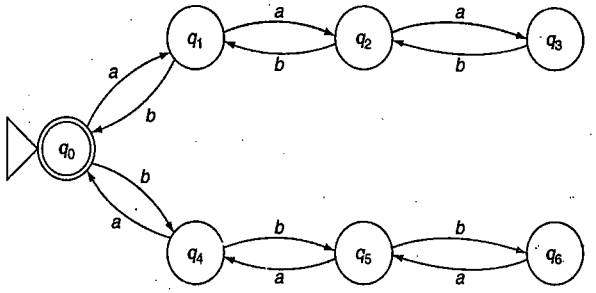


FIGURE 2.13 A partial automaton for equal numbers of as and bs.

Transitions away from q_0 increase the difference between the numbers of a and b while transitions bringing the automaton closer to the accepting state q_0 cancel out as and bs and reduce the difference. This machine does the job as long as the number of as does not exceed the number of bs by three at any point; and the number of bs does not exceed the number of as by three at any point. What if there are four more as than bs? The machine needs to remember this because it has to look for four more bs to cancel out the extra as. Hence it needs a new state to represent the condition of having seen four more as than bs. This state cannot be the same as any of the above states since the numbers are different in each state. What if there are five extra as? Since there is no limit on the number of input symbols or the order in which they can appear, we need a limitless number of states in this automaton. It is no longer a *finite* automaton. As already noted, we can never implement a computing machine with an infinite number of states. If there is no limit on the input size, there is no solution using simple automata; even if there is a limit in practice, if the size is very large, the automaton will not be an efficient machine: it has too many states and is roughly equivalent to a program in a high-level programming language with a very long sequence of if-then-else statements (or a switch statement with too many cases) instead of a simple loop that counts the number of as and bs. This limitation is often stated as *the inability of finite automata to do absolute counting, that is, to count the number of symbols in its input.* Finite automata are able to do this only if they have as many states as the number to be counted.

EXAMPLE 2.13

Does it help if the symbols are ordered in a particular way? For example, what if all the as appear first and only then the bs? Would an automaton be able to count and match the number of as to the number of bs? Such a string is often abbreviated as $a^n b^n$, which stands for n number of as followed by n number of bs. Figure 2.14 shows an automaton that attempts to solve this problem but, once again, only up to a count of 5.

It is apparent from this automaton that we need new states for each successive value of n. It needs a new state to remember each new value for the number of as that is different from all previously represented values. The number 6 is different from each of 1, 2, ..., 5 and therefore we need a new state beyond q_5 to handle $n = 6$.

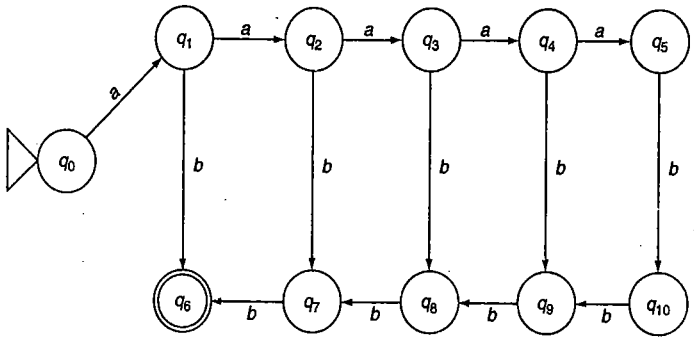


FIGURE 2.14 An automaton for the language $a^n b^n$ with $n < 6$.

Note also that the above automaton is incomplete. Apart from not showing a reject state for an input that begins with the symbol b, there is no transition for the symbol a from state q_5. This in itself indicates that something is amiss with the automaton. The automaton in Fig. 2.13 was also incomplete: there was no arrow for a from q_3 and no arrow for b from q_6 of that automaton.

In Chapters 7 and 8, we will see how to devise machines and computing solutions for problems of this kind. We will also see that problems of this kind are very important in practice, for example, in matching parentheses in a program. The simple automata that we studied in this chapter do not even have the ability to count (the number of opening and closing parentheses, for instance). Their capability is equivalent in a sense to that of a high-level programming language without any looping construct! Finite automata are unable to count in order to iterate through a loop or do the equivalent of assigning an integer value to a variable, as we do routinely in programming modern computers, and as such are often impractical. Nevertheless, finite automata with a large number of states are used effectively in a variety of state-machine models including lexers, communication protocol design and testing of hardware and software.

2.12 Automata with Combinatorial States

There is another class of problems that finite automata are able to solve but not with any degree of efficiency. Typically, in such problems, there are a number of possibilities and the automata must consider all their combinations individually by assigning separate states to each combination. This results in a combinatorial explosion in the number of states making the construction of such automata a tedious job. For example, we have seen in Section 2.4 how to construct automata to handle conditions at the end of the input string. As the condition is moved farther away from the end of the input, larger and larger numbers of states would be needed in a DFA.

EXAMPLE 2.14

Consider an automaton that is required to accept all binary strings where the fifth digit from the end of the string is a 0. Figure 2.15 shows such an automaton. It accepts all strings such as 01111, 001001, 0011101011, etc.

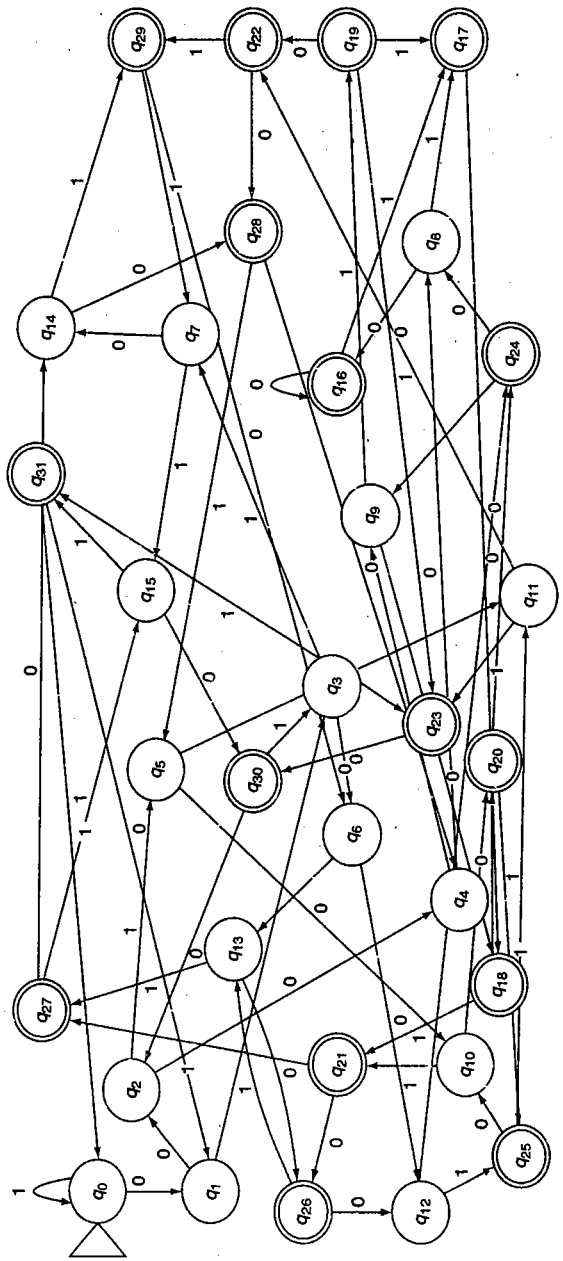


FIGURE 2.15 A DFA for binary strings with a 0 five positions from the end.

It is clear that this automaton, while it may be able to compute correctly, is too unwieldy for human comprehension, construction or maintenance. In the next chapter, we will see how to use the idea of non-determinism to build equivalent computing machines for such problems efficiently.

2.13 Key Ideas

1. A computing machine accepts some input and produces some output by processing the input, usually making some decisions during its processing.
2. Automata are the simplest kinds of computing machines. They are abstract models of computing usually represented as state transition diagrams.
3. Automata are defined in terms of their states and state transitions. They consume an input string made up of symbols taken from an alphabet and either accept or reject the input by reaching an appropriate state at the end of processing the input.
4. States are the only memory that automata have. They need to change states to remember anything.
5. Automata are used to model hardware and software solutions to computing problems. Since real solutions can only have a finite amount of hardware or a finite number of lines of code in a software program, automata must also be finite. The number of states in an automaton must be finite.
6. Deterministic finite automata never encounter multiple choices. For a given input symbol, when the automaton is in a given state, there is never more than one possible transition.
7. Although there is no algorithm to design an automaton, the set of mantras discussed in the chapter help us in designing an accurate finite automaton for a given problem.
8. Automata, being primitive computing machines, are limited in the variety of problems they can solve. A key limitation is that they are unable to count a set of symbols in the input string. As such, they cannot solve some classes of problems without requiring an infinite number of states.
9. Even when they can solve a problem, deterministic finite automata often require a large number of states.

2.14 Exercises

1. Show that there exists a deterministic finite automaton with just one state that can accept an infinite set of strings.
2. Show that for the alphabet $\{a\}$ there cannot be a deterministic finite automaton with just two states that accepts more than two strings but only a finite number of strings.

3. What is the minimum number of states in a finite automaton for implementing a soft-drink vending machine that accepts ₹2, 5 and 10 coins and returns correct change after delivering the drink? The price of the soft-drink is ₹22 and it delivers the drink as soon as the amount put in by the user reaches or exceeds ₹22.

A. Construct a finite automaton for each of the following.

4. Binary strings in which every 0 is followed by 11.
5. Strings containing at least one a and at least one b.
6. Strings containing at least two as and ending with an even number of bs.
7. Some number of as followed by some number of bs with the total length of the string being odd.
8. Binary strings with no consecutive 0s. Show the computation for the input string $w_1 = 1011101$ and for $w_2 = 1001$.
9. Strings over the alphabet $\{a, b\}$ of the form $(ab)^n$, for example, *ababab*.
10. Strings over the binary alphabet that do not contain the sub-string 010.
11. Strings over the alphabet $\{a, b\}$ that end with either *ab* or *ba*. Show the computation for the input string $w = ababaab$.
12. Student grades in an examination are represented with the letters $\{A, B, C, D, F\}$. An input string such as *ABFCAD* indicates the grades obtained by a student in six different subjects. The automaton must accept only those students who have scored at most three Ds and no Fs.
13. The automaton must accept only those students who have scored at least three As or Bs (overall).
14. Strings over $\{a, b\}$ in which either all even-numbered symbols are a or all odd-numbered symbols are b. Show the computation for the string $w_1 = aababaa$ and for $w_2 = babaab$.
15. The automaton must reject any string over $\{0, 1, 2\}$ where a 2 is immediately preceded by a 0. It should accept all other strings.
16. Binary strings with an even number of 0s and an odd number of 1s.
17. Strings of the form *abwba* over the alphabet $\{a, b\}$, where w is any string over the same alphabet.
18. The set of all binary strings that do not contain three or more consecutive 0s.
19. The set of all binary strings that start with 111 and end with 100. Assume that each string is at least five characters long. Show the computation for the string $w_1 = 111010100$ and for $w_2 = 11100$.
20. The set of all binary strings having a sub-string 100 before a sub-string 111 (both must be present). Show the computation for the string $w = 00100110111$.
21. The set of all strings that are palindromes of length 4. The alphabet is $\{a, b, c\}$.

22. A *run* in a string is a sub-string made up of a single symbol. For example, *aabbbcd* has a run of as of length 2 and a run of bs of length 3. The automaton must accept all strings over the alphabet $\{a, b, c, d\}$ with at least one run of as of length 2 and a run of cs of length 3.
23. Strings over $\{a, b\}$ such that the length of the string is a multiple of 3 but not a multiple of 4.
24. Binary strings representing positive integers divisible by 5 but not divisible by 3.
25. Binary strings with at least two occurrences of at least two consecutive 1s, the two occurrences not being adjacent (i.e., 011011 and 011111 are acceptable but 01111 is not).
26. Consider the unary number system with the alphabet $\{1\}$ where a number n is represented by a string of n 1s, for example, 4 is 1111 and 7 is 1111111. Construct a finite automaton that accepts all unary numbers that are divisible by 4 but not divisible by 3.
27. Strings of the form ab^3wb^4 where the alphabet is $\{a, b\}$ and w is any string over the same alphabet.
28. All strings over $\{a, b\}$ where the total number of as is equal to the total number of bs. Further, at any point in the string, the number of bs seen thus far cannot exceed the number of as seen by more than 2; similarly, the number of as seen thus far cannot exceed the number of bs seen by more than 2. For example, *aababb* is acceptable but *aabaabbb* is to be rejected.
29. All binary strings where the third and second symbols from the end are not the same.
30. All binary strings where the second symbol is the same as the penultimate (i.e., the second from the end) symbol.

B. For the following problems, a transition table is specified.

31. Describe the language (i.e., set of all strings) accepted by the following automaton:

State	Input = a	Input = b
$\rightarrow q_0$	q_1	q_3
q_1	q_3	q_2
$*q_2$	q_2	q_2
q_3	q_3	q_3

32. Describe the language (i.e., set of all strings) accepted by the following automaton:

State	Input = a	Input = b
$\rightarrow q_0$	q_2	q_1
q_1	q_1	q_1
q_2	q_3	q_2
$*q_3$	q_3	q_2

C. Debug and fix the following defective finite automata.

33. The automaton in Fig. 2.6, Example 2.5.
34. The automaton shown in Fig. 2.16 for accepting all binary strings with alternating 0s and 1s.

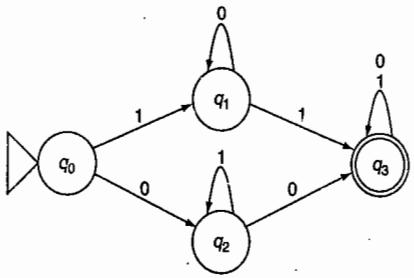


FIGURE 2.16

35. The automaton shown in Fig. 2.17 for accepting strings over $\{a, b\}$ that either begin and end with an a and contain an even number of bs in between or begin and end with a b and contain an even number of as in the middle.

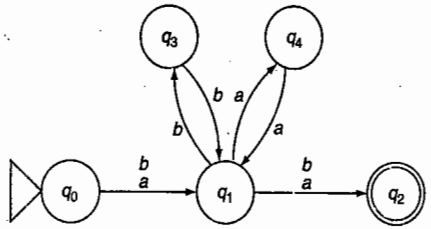


FIGURE 2.17

36. The automaton shown in Fig. 2.18 for all binary strings representing positive numbers divisible by 7.

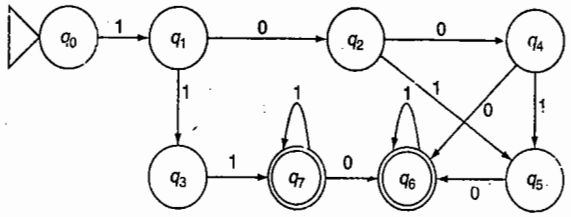


FIGURE 2.18

D. Explain why we cannot construct a finite automaton for the following.

37. In coding theory, the *weight* of a binary string is the number of 1 s in the string. Consider the set of all binary strings whose weight is less than half the length of the string.

38. Unequal numbers of as and bs, in any order, for the alphabet $\{a, b\}$.

39. Palindromes (i.e., strings that read the same forward or backward, for example, *malayalam*) of arbitrary length.

40. Some number of as (however large) followed by double the number of bs.

41. Formulate a set of *mantras* to decide whether a finite automaton can be constructed to accept a given set of strings.

(*Hint:* Generalize from exercises 37–40.)

Chapter Three

Non-Deterministic Finite Automata

Learning Objectives

After completing this chapter, you will be able to:

- Learn to use non-determinism as a tool in designing automata.
- Learn how to design non-deterministic automata for a given problem.
- Learn to convert a non-deterministic automaton to a deterministic one through subset construction.
- Learn the use of λ-transitions in designing non-deterministic automata.
- Learn to minimize the states in an automaton.
- Understand how finite state transducers work.

The key idea presented in this chapter is that of non-determinism. The reader will understand how non-determinism is very useful to us human beings in thinking about and constructing automata but why, not just how, it does not add any additional computing ability to the machine. Towards the end of the chapter, the student will also learn how to reduce an automaton to a minimal number of states so that its implementation in hardware or software is economical. The chapter also introduces finite-state transducers such as Moore and Mealy machines.

3.1 *The Idea of* Non-Determinism

We saw in Chapter 2 how it was easier to construct a deterministic finite automaton (DFA) for inputs that begin with a given symbol than for those that end with the symbol. Let us try another example to appreciate the relative ease or difficulty in constructing automata for strings that begin with, say, 10 and strings that end with 10.

EXAMPLE 3.1

Figure 3.1 shows a DFA for accepting all binary strings that begin with 10. After verifying, using states q_0 and q_1, that the input does begin with 10, the automaton accepts in its final state q_2 any further symbols from the input.

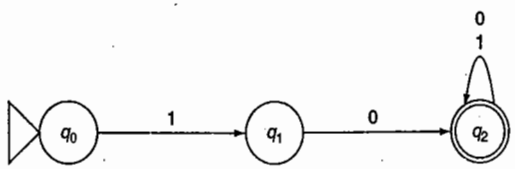


FIGURE 3.1 Deterministic finite automaton for strings that begin with 10.

In comparison, the DFA in Fig. 3.2 accepts strings that end with 10. Although states q_0 and q_1 still recognize the 1 and the 0 to take the automaton to its final state q_2, this automaton has to take care of several more transitions since it does not know when the end of the input string is nearing. Every time it sees a 1, it assumes that it is seeing the penultimate symbol of the input (i.e., the 1 in the 10). If it was wrong, it has to backtrack to appropriate states as shown in Fig. 3.2.

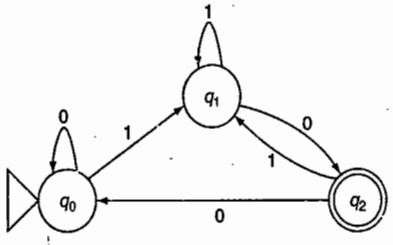


FIGURE 3.2 Deterministic finite automaton for strings that end with 10.

Is there a way to make this automaton as simple as the previous one? Can we make the construction of this automaton as trivial as the other? We introduce the concept of *non-determinism* to create the necessary magic. Consider the automaton shown in Fig. 3.3. It too accepts input strings that end with 10 but it looks as simple as the DFA for strings beginning with 10 (Fig. 3.1). If you observe our new automaton closely, you will see that it has a problem: there are two transitions for the symbol 1 from its state q_0! If the automaton is in state q_0 and the input symbol is 1, it has two options. It can either remain in the same state or it can transit to state q_1. Its behavior is no longer deterministic.

How does it know which arrow to follow? There are two ways of understanding the behavior of such a *non-deterministic finite automaton* (NFA):

1. It guesses and somehow makes the right choice every time magically.
2. It tries multiple possibilities in parallel and one of the choices will turn out to be the right one eventually.

In practical terms, non-determinism can be seen as an exhaustive search for possible solutions to a problem and this search can be conceived in terms of parallel processing. In general, there may be more than one such non-deterministic transition with a resulting *combinatorial explosion* in the number of possibilities (or paths) to be searched before a solution can be found. If any guess or any

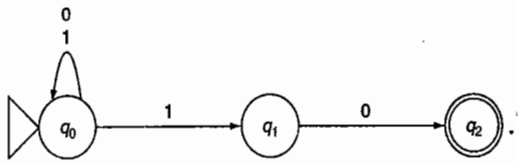


FIGURE 3.3 Non-deterministic finite automaton for strings that end with 10.

parallel thread leads to an accepting state exactly when the entire input is consumed, we say that the machine has accepted the input string. As such, non-determinism remains essentially a theoretical construct; to realize a non-deterministic machine in practice it takes unlimited capabilities for parallel processing using which we can execute every possible combination of choices in parallel processors or threads.

The NFA in Fig. 3.3 remains in its start state q_0 when it sees one or more 0s in the input. When a 1 occurs, it either remains in the same state or it changes to state q_1. Suppose it decided to change to q_1. If the choice is incorrect for the given input, that particular choice will end up rejecting the input. However, since other configurations are being tried in parallel, in at least one of them the NFA would have decided to remain in state q_0 for the same input symbol which turns out to be the right choice for the given input. In a way, this NFA is able to guess when it is two symbols away from the end of the input. Only at that point, it goes to state q_1 and then to q_2 to accept the string.

In an NFA, therefore, even if it accepts an input, it will have tried configurations that rejected the string. For example, if the input to the NFA in Fig. 3.3 is 011010, it tries each of the following configurations:

$q_0 \to q_0 \to q_1 \to$ reject	Transition out of q_0 on 1st occurrence of 1
$q_0 \to q_0 \to q_0 \to q_1 \to q_2 \to$ reject	Transition out of q_0 on 2nd occurrence of 1
$q_0 \to q_0 \to q_0 \to q_0 \to q_0 \to q_1 \to q_2 \to$ ACCEPT	Transition out of q_0 on 3rd occurrence of 1
$q_0 \to q_0 \to q_0 \to q_0 \to q_0 \to q_0 \to q_0 \to$ reject	No transition out of q_0 at all

Of these configurations, only the third one will result in successfully accepting the input. The second choice rejects the string because although it has reached the final state q_2, the input is not exhausted. In other words, if we were to show the complete automaton, there would have been a transition from the final state q_2 to a reject state for both symbols 0 and 1 (since we are interested in accepting the pattern 10 at the end of the input, not somewhere in the middle).

Thus, an NFA is the same as the DFA that we studied in the previous chapter except that its transition function now maps to sets of states. That is, in a given state and for a given input symbol the machine can make transitions to a set of states.

3.2 Constructing Non-Deterministic Automata

Let us work out a few examples to see how it is often easier to construct a non-deterministic than a deterministic automaton.

EXAMPLE 3.2

On similar lines, an NFA for accepting even numbers over the binary alphabet, as shown in Fig. 3.4, is simpler than a corresponding DFA (shown in Chapter 2, Fig. 2.3). When it encounters a 0, it has to make the right guess about whether that symbol is indeed at the end of the input since automata, being very simple computing devices, do not have the ability to look ahead and see what comes next in their input.

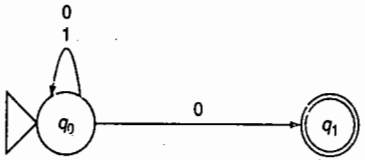


FIGURE 3.4 NFA for even numbers.

EXAMPLE 3.3

Our example from the previous chapter of an automaton for searching for the keyword 101 (Example 2.9, Fig. 2.10) can also be simplified by constructing it as an NFA as shown in Fig. 3.5. Here, non-determinism once again makes it convenient to design the automaton since we no longer have to add transitions for backtracking when the automaton wrongly guessed that the current symbol 1 was the beginning of the keyword 101. Instead, we give the machine the choice to either remain in the start state or move ahead on seeing 1 and non-determinism takes care of making the correct guess about when to leave the start state.

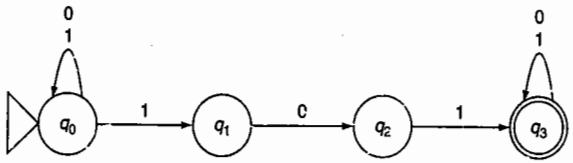


FIGURE 3.5 NFA to search for a keyword.

EXAMPLE 3.4

Often, it is easier to first construct a non-deterministic automaton and then convert it to a deterministic one if necessary. To illustrate this approach, consider the set of all strings over the alphabet $\{a, b\}$ where the first and last symbols are the same. That is, if the input string starts with an a, it should end with an a; if it starts with a b, it should end with a b. Figure 3.6 shows an NFA for this problem. The first and last symbols match in the two branches of the NFA. The states in the middle, q_1 and q_2, consume the rest of the input symbols. Transitions out of q_1 for a and q_2 for b are non-deterministic. The NFA guesses exactly when to go from q_1 or q_2 to the final state q_3. Notice, in particular, that there are no outgoing arrows from the final state. This state is reached only when the last symbol of the input matches the first symbol. If the state is reached too early when there are still input symbols remaining because of an incorrect guess, then the input is rejected (but will be accepted in another parallel path where the correct guess is made).

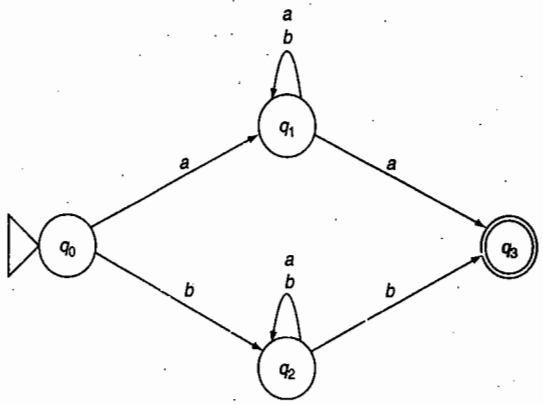


FIGURE 3.6 NFA for strings with the same first and last symbols.

This NFA can be converted to an equivalent DFA as shown in Fig. 3.7 (a general method for such conversion is provided in Section 3.3). Notice that in the DFA, every time an input symbol that matches the first symbol of the branch is found, the automaton goes to the final state hoping that the current symbol is the last symbol. If it encounters a different symbol while in the final state (b in the a-branch or a in the b-branch), it comes back to the intermediate state (q_1 or q_2) as shown in the figure. It may also be noted that the DFA needs two separate final states (q_3 and q_4). Any attempt to merge the two final states will lead to errors; the two branches must be kept separate so that the machine does not accept inputs that do not have matching first and last symbols. There was no such requirement in the case of the NFA which had a single final state and no outgoing transitions from the final state. This is yet another way by which constructing an NFA is often easier than constructing an equivalent DFA.

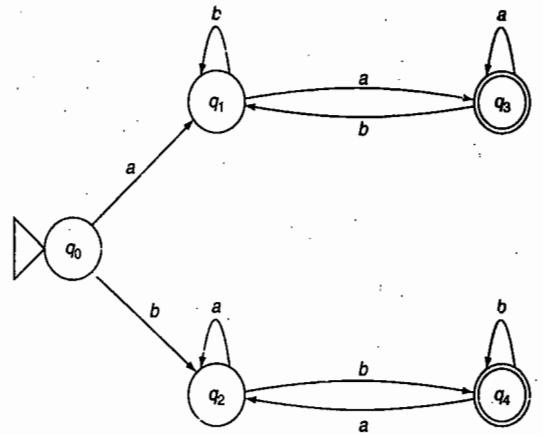


FIGURE 3.7 DFA for strings with the same first and last symbols.

EXAMPLE 3.5

Even the most cumbersome deterministic automaton that we constructed in the previous chapter to illustrate a key limitation of deterministic finite automata (DFAs; Example 2.14, Fig. 2.15) looks rather simple when constructed as an NFA as shown in Fig. 3.8. In this problem, the machine had to accept an input string only if the fifth symbol from the end of the input was a 0. The NFA non-deterministically guesses when it is facing the fifth symbol from the end of the input. It remains in the start state until that point.

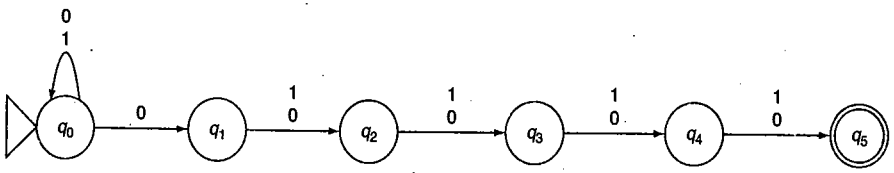


FIGURE 3.8 NFA for strings with a 0 as the fifth last symbol.

It may be noted that the *mantras* for non-deterministic finite automata (NFAs) are the same as the ones we applied for constructing deterministic automata in the previous chapter except for a minor change: we no longer check to see if there is exactly one transition for every symbol from every state (*mantra* 8 in Sec. 2.10). NFAs are allowed to have more than one transition for the same symbol. As such, in the case of NFAs, we need to only verify whether there is at least one transition for every symbol from every state.

In the case of some problems, non-determinism makes no difference; there is nothing to be gained from non-determinism in constructing automata for such problems which do not involve a non-deterministic choice in their state transitions. For example, the automaton in Fig. 2.7(c) for accepting all strings with three or fewer as and the one in Fig. 2.8(b) for binary numbers divisible by 3 do not need non-determinism. An NFA for the problems would be the same as the corresponding DFA.

Also, problems that could not be solved by DFAs due to their inability to count cannot be solved by non-deterministic automata either. The automata in Fig. 2.13 for equal numbers of as and bs or the one in Fig. 2.14 for $a^n b^n$ cannot overcome their limitations by being converted to NFAs. Even the NFA would need an unlimited number of states to be able to count the number of symbols in its input.

From these considerations it appears that non-determinism helps us in only two ways:

1. Non-determinism makes it easier to design and construct an automaton by ensuring that it is equally easy to handle constraints in any part of the input string without a need for backtracking in the case of incorrect decisions made by the automaton; and
2. Non-determinism reduces the number of states (and transitions) in the automaton.

In fact, we will show in the following section that every NFA can be converted to an equivalent DFA, thereby showing that their computing powers are the same. NFAs can solve exactly the same problems that DFAs can.

3.3 Eliminating Non-Determinism: Converting NFA to DFA

The method for eliminating non-determinism to convert an NFA to a DFA can be understood as follows. The primary difference between the two kinds of computing machines is that the DFA is always in a single state after processing one or more input symbols, whereas the NFA can be in multiple different states at the same time (in its multiple parallel attempts to process the input string). Can we enumerate all the possible states in which an NFA can be present after processing a particular sequence of inputs symbols? We will see below how such an enumeration can be done in a compact manner using a suitable table. The table keeps the enumeration compact by considering only the set of transitions possible in the NFA from a given state of the NFA and for a given input symbol. Once we exhaustively enumerate the possible next states for all possible states and for all possible input symbols, we can construct an equivalent DFA where each distinct set of alternative states of the NFA is mapped to a unique state in the new DFA.

EXAMPLE 3.6

Starting from its initial state q_0, let us say the NFA can reach any of the states q_1, q_2 and q_3 for the input symbol 0 (Fig. 3.9). The corresponding new DFA will have a single state Q_1 which is equivalent to the set of NFA states $\{q_1, q_2, q_3\}$ as shown in Fig. 3.10. From any of these three states, if the next input symbol is 1, let us say the NFA can reach any of its states q_0, q_1 and q_4. We construct a new state Q_2 in the DFA that corresponds to the set $\{q_0, q_1, q_4\}$.

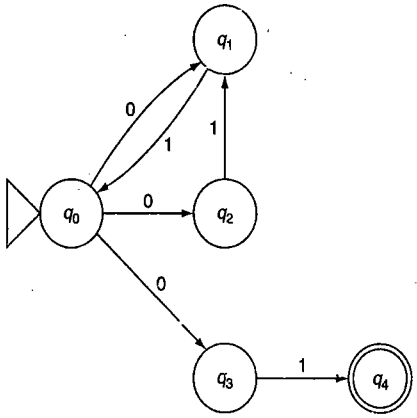


FIGURE 3.9 NFA example for conversion to DFA.

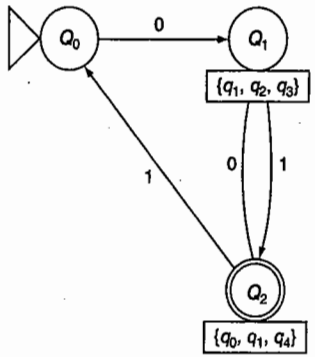


FIGURE 3.10 DFA converted from NFA.

Why is there a transition from Q_2 to Q_1 on 0 and Q_2 to Q_0 on 1? In the NFA, there is a transition from q_1 to q_0 on 1 because of which we need to add a transition from $\{q_0, q_1, q_4\}$, that is Q_2, to $\{q_0\}$, that is Q_0. Similarly, there are non-deterministic transitions from q_0 to $\{q_1, q_2, q_3\}$, that is, Q_1 on 0. This gets translated to a transition in the DFA from $\{q_0, q_1, q_4\}$, that is, Q_2 to Q_1.

Why will such an enumeration terminate?

It does so because the NFA has a finite number of states and as such there can only be a finite number of distinct subsets of its states. In the worst case, it is the power set of the set of states in the NFA and this will be the number of states in the resulting DFA.
This algorithm for converting the NFA to a DFA is known as *subset construction.* It works as follows.

Algorithm

Starting from the initial state q_0 of the NFA,
For each symbol in the alphabet, repeat
Determine the set of states that the NFA can reach from the current state(s) for the input symbol;
If this set is different from all the sets already enumerated,
Create a new state Q_i in the DFA for this set.
Add the new state to the set of states in the enumeration until no new set can be added.
For each state Q_i in the DFA and for each symbol in the alphabet,
Add a transition to a DFA state that corresponds to any of the transitions in the NFA.

The tabular method of enumerating the sets of states is shown in Table 3.1 for the example of Figs. 3.9 and 3.10. As you can see, this is similar to the transition table used for DFAs (see Table 2.1 for an

example) except that entries in the table are all now sets of states instead of individual states in the case of a DFA. It should also be mentioned that the set containing the start state of the NFA $Q_0 = \{q_0\}$ is always the start state of the DFA; any set that contains one or more of the final states of the NFA is a final state in the DFA. As such, Q_2 which contains q_4 is the final state in this example.

TABLE 3.1 Example of Converting NFA to DFA

DFA State	Input Symbol = 0	Input Symbol = 1
$\rightarrow Q_0 = \{q_0\}$	$\{q_1, q_2, q_3\}$	$\{\}$
$Q_1 = \{q_1, q_2, q_3\}$	$\{\}$	$\{q_0, q_1, q_4\}$
$^* Q_2 = \{q_0, q_1, q_4\}$	$\{q_1, q_2, q_3\}$	$\{q_0\}$

EXAMPLE 3.7

Consider the problem of constructing an automaton to accept the set of all inputs that are binary numbers divisible by 4 or by 6. This automaton must accept binary representations of 0, 4, 6, 8, 12, 16, 18, 20, etc., but should reject 2, 10, 14, 22, etc. apart from all odd numbers. Let us first construct an NFA for this problem as shown in Fig. 3.11. We observe that all acceptable inputs are even, that is, they all end with a 0. The benefit of constructing an NFA first is that we can deal with the two choices – divisible by 4 or divisible by 6 – separately from one another. Trying to construct a DFA directly would force us to consider various combinations of the details of both choices together. It may be noted that the NFA in Fig. 3.11 has non-determinism in three places: state q_0 on input symbol 1, q_1 on 0 and q_6 on 0.

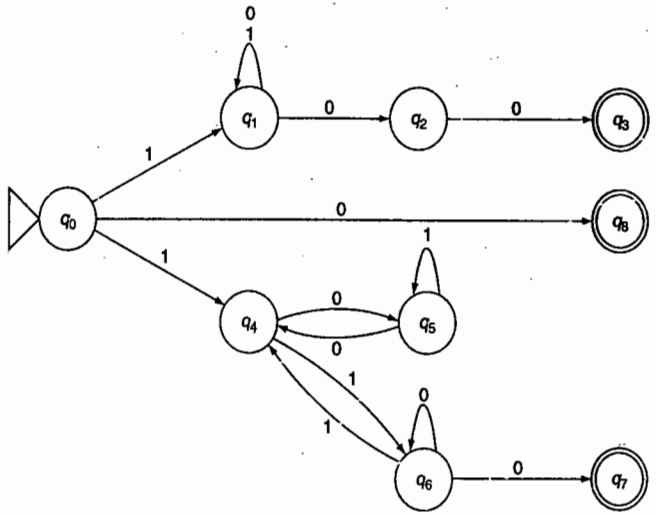


FIGURE 3.11 NFA for binary numbers divisible by either 4 or 6.

The NFA has three accepting states: q_3 accepts all numbers divisible by 4; q_8 is for the special case of 0 itself (since we do not want to accept other inputs with leading 0s); and q_7 accepts all

numbers divisible by 6. The part of the NFA consisting of states q_4, q_5 and q_6 checks for divisibility by 3 [and is similar to the automaton shown in Fig. 2.8(b)]. Also, it is possible in this NFA for the same input to be accepted in multiple final states. For example, given 1100 (12) or 100100 (36), the machine accepts them in both q_3 and q_7 in its parallel attempts at trying multiple configurations non-deterministically.

Before trying to convert this non-deterministic automaton to a deterministic one by applying the algorithm outlined above, let us list the meanings of the states in the NFA:

1. q_0 is the start state.
2. q_1 is the state where all symbols are consumed before looking for an ending with 00.
3. q_2 is where one 0 has been seen, that is, the number so far is an even number.
4. q_3 is the final state where the input is divisible by 4.
5. q_4 is the state where *mod* 3 of the number so far is 1.
6. q_5 is the state where *mod* 3 of the number so far is 2.
7. q_6 is the state where the number so far is divisible by 3.
8. q_7 is the final state where the number is divisible by 6.
9. q_8 is the special final state for accepting the number 0.

Now let us construct sets of equivalent states of the NFA to convert it to a DFA. We begin with the set $\{q_0\}$ and make it the start state Q_0 of the DFA in the first row of Table 3.2. From the start state, the NFA reaches $\{q_8\}$ on 0 and the set of states $\{q_1, q_4\}$ on 1. We add these two new states Q_1 and Q_2 to the DFA being constructed. As we fill in the 0 and 1 column for each row, if any new set of states is added, we create a new row for that set and make it a new DFA state Q_i. As shown in the table, this process eventually ends when no more new sets (and hence no more new rows) can be added. Figure 3.12 shows the resulting DFA that is constructed by simply reading the states and transitions from the table.

TABLE 3.2 Converting NFA to DFA for Divisibility by 4 or 6

DFA State	Input Symbol = 0	Input Symbol = 1
$\rightarrow Q_0 = \{q_0\}$	$\{q_8\} = Q_1$	$\{q_1, q_4\} = Q_2$
$* Q_1 = \{q_8\}$	$\{\}$	$\{\}$
$Q_2 = \{q_1, q_4\}$	$\{q_1, q_2, q_5\} = Q_3$	$\{q_1, q_5\} = Q_4$
$Q_3 = \{q_1, q_2, q_5\}$	$\{q_1, q_2, q_3, q_4\} = Q_5$	$\{q_1, q_5\} = Q_6$
$Q_4 = \{q_1, q_6\}$	$\{q_1, q_2, q_6, q_7\} = Q_7$	$\{q_1, q_4\} = Q_2$
$* Q_5 = \{q_1, q_2, q_3, q_4\}$	$\{q_1, q_2, q_3, q_5\} = Q_8$	$\{q_1, q_6\} = Q_4$
$Q_6 = \{q_1, q_5\}$	$\{q_1, q_2, q_4\} = Q_9$	$\{q_1, q_5\} = Q_6$
$* Q_7 = \{q_1, q_2, q_6, q_7\}$	$\{q_1, q_2, q_3, q_6, q_7\} = Q_{10}$	$\{q_1, q_4\} = Q_2$
$* Q_8 = \{q_1, q_2, q_3, q_5\}$	$\{q_1, q_2, q_3, q_4\} = Q_5$	$\{q_1, q_5\} = Q_6$
$Q_9 = \{q_1, q_2, q_4\}$	$\{q_1, q_2, q_3, q_5\} = Q_8$	$\{q_1, q_6\} = Q_4$
$* Q_{10} = \{q_1, q_2, q_3, q_6, q_7\}$	$\{q_1, q_2, q_3, q_6, q_7\} = Q_{10}$	$\{q_1, q_4\} = Q_2$

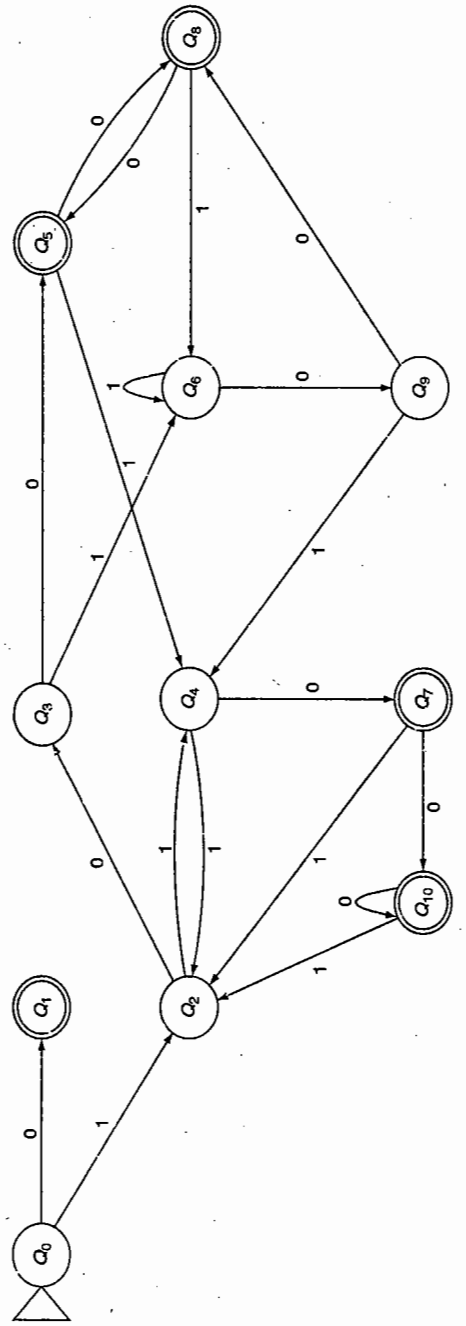


FIGURE 3.12 DFA obtained by converting the NFA of Fig. 3.11.

Examining the DFA in Fig. 3.12, we appreciate how much more difficult it would have been to construct the deterministic automaton directly instead of first constructing the much simpler NFA of Fig. 3.11. To complete the example, let us list the meanings of the final states in the DFA:

1. Q_1 is the special final state for accepting the number 0.
2. Q_5 is the state where the number is divisible by 4 but not by 6 (*mod* 6 is 4).
3. Q_7 is the state where the number is not divisible by 4 but is divisible by 6.
4. Q_8 is the state where the number is divisible by 4 but not by 6 (*mod* 6 is 2).
5. Q_{10} is the state where the number is divisible by both 4 and 6.

What are the meanings of the non-final states in the DFA?

For example, the number so far is odd in states Q_2, Q_4 and Q_6 and even but not divisible by either 4 or 6 in Q_3 and Q_9, but how do they differ from one another? In the case of a DFA constructed from an NFA, since states are in fact sets of NFA states, the meanings of some of the DFA states often cannot be stated easily or concisely.

3.4 Jumping States Without Input

So far, in both deterministic and non-deterministic automata, we have enforced a constraint on the computing machine that it can change its internal state only on consuming an input symbol. What if we relax this and allow the machine to change its state without looking at the input symbol? In other words, what would happen if we allowed the machine to jump to another state on its own will?

In such a machine, there would be a transition from a state q_i to a state q_j with no input symbol marked on the arrow. By convention, we mark the null symbol λ on such transitions and call such machines *automata with λ-transitions*. Necessarily such a machine would be non-deterministic because its behavior is no longer fully determined by the input string. It has the freedom to change its state and it may or may not choose to do so when it is in a certain state, thereby making its behavior unpredictable or non-deterministic.

Why do we need to give such freedom to computing machines?

The reasons are the same as for non-determinism: allowing λ-transitions makes it easier for us to construct automata, and, we would like to see if this makes the computing machine more powerful, that is, whether it can solve problems that deterministic automata could not.

Let us consider an example where λ-transitions help us in designing the automaton.

EXAMPLE 3.8

We need to construct an automaton that accepts all binary input strings which are numbers divisible either by 3 or by 5. We have already constructed DFAs for divisibility by 3 and by 5 separately [see Figs. 2.8(b) and 2.9]. We can now combine them using λ-transitions to quickly construct an NFA for the present problem as shown in Fig. 3.13. The upper half of the NFA computes divisibility by 3 and the bottom half divisibility by 5. The two parts are connected from the start state through λ-transitions.

For the purpose of illustrating the complexity of computing *mod* 3 and *mod* 5 jointly in a deterministic automaton, the NFA of Fig. 3.13 converted using a modified method of subset construction for eliminating λ-transitions is shown in Fig. 3.14.

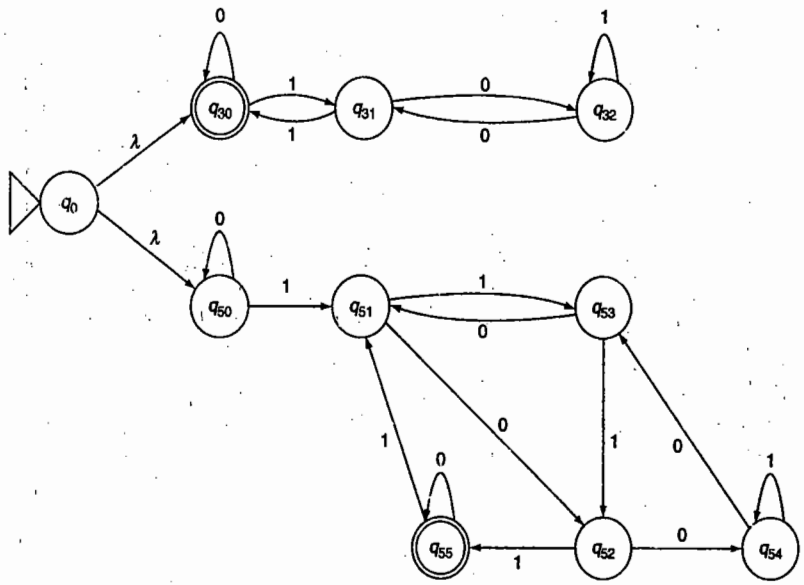


FIGURE 3.13 NFA with λ-transitions for divisibility by 3 or 5.

The modification to the method of subset construction for handling NFAs with λ-transitions is as follows:

1. We do not consider λ as an input symbol; we still construct the table only for the other symbols, that is, only two columns for the two symbols 0 and 1 when the alphabet is binary.
2. But we consider λ-transitions in computing the sets of states for a given current state and a given input symbol.

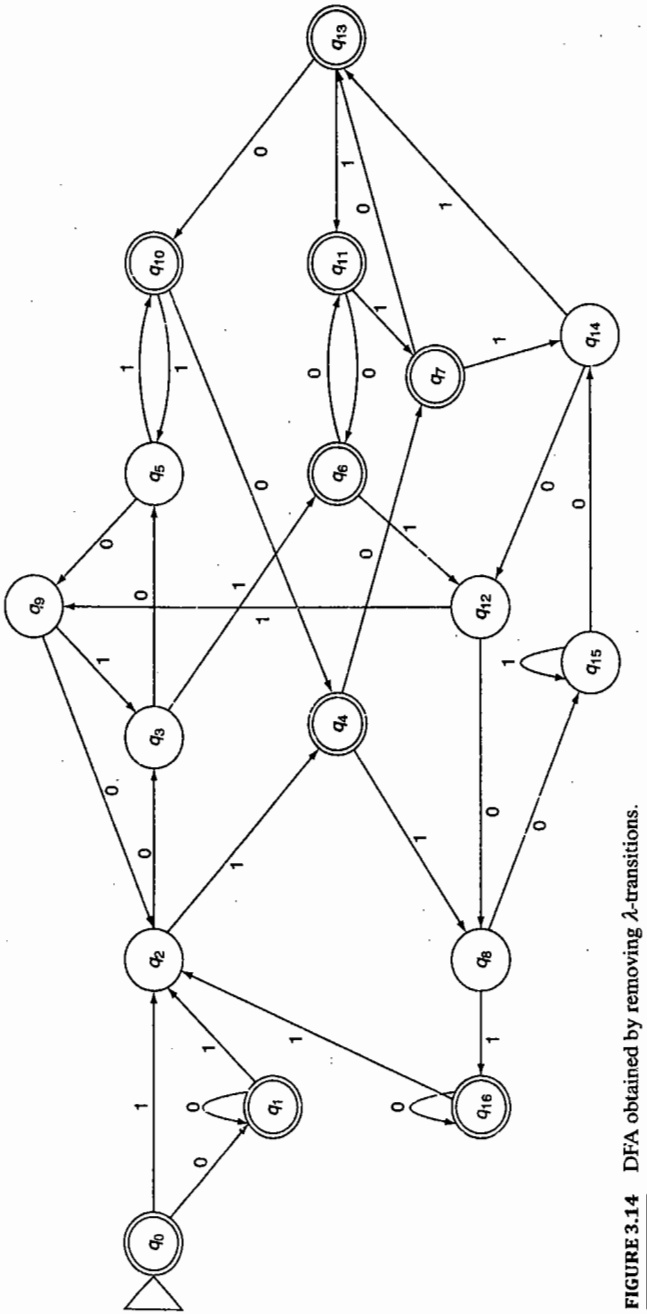


FIGURE 3.14 DFA obtained by removing λ-transitions.

For example, from the start state $\{q_0\}$, on input symbol 0, we recognize that although there are no outgoing arrows labeled 0 from that state, the automaton can on its own accord jump to either state q_{30} or state q_{50}. We, therefore, consider outgoing arrows labeled 0 from those states as though they start from q_0 itself. As such, the first row of the table for converting the NFA in Fig. 3.13 will be:

DFA State	Input Symbol = 0	Input Symbol = 1
$Q_0 = \{q_0\}$	$\{q_{30}, q_{50}\}$	$\{q_{31}, q_{51}\}$

We proceed from here on with the conversion through *subset construction* since there are no more λ-transitions in this NFA. If there was one more λ-transition say from q_{30}, then all the states reachable through this λ-transition should also be included in set of states. In other words, the set that we enter into the table is the set of all states reachable from the current state by consuming the current input symbol including any transitions that can occur without consuming any input symbol either before or after consuming the current input symbol.

This example illustrated an important idea in programming and software engineering known as *modularity*. The given problem clearly had two modules in its solution: computing divisibility by 3 and computing divisibility by 5. Using the flexibility of non-deterministic automata with λ-transitions, we were able to exploit the modularity by independently designing and building solutions for each module and then integrating the modules through λ-transitions.

EXAMPLE 3.9

Let us say we need to construct an automaton for accepting all strings over the alphabet {0, 1, 2}, where no 1 can occur before a 0 and no 2 can occur before a 1 or a 0. There is no requirement that all the three symbols must be present in an acceptable input. A DFA for this problem is shown in Fig. 3.15. Once again, it is much easier to construct an NFA with λ-transitions as shown in Fig. 3.16 than the DFA. While the DFA needed all three states to be final states and a transition on 2 from the start state to q_2 to accommodate acceptable inputs that do not contain 1 s or do not contain 2 s, the NFA has a single final state and is a lot easier to understand.

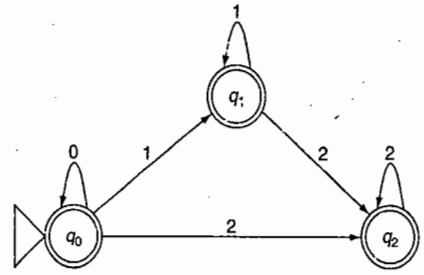


FIGURE 3.15 DFA for strings with 0, 1 and 2 in that order.

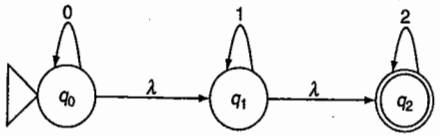


FIGURE 3.16 NFA for strings with 0, 1 and 2 in that order.

The differences between the DFA in Fig. 3.15 and the NFA in Fig. 3.16 are very significant in ensuring the *maintainability* of computing solutions in general and software in particular. In addition to the ease of design and construction, software solutions need to be easy to comprehend so that they can be used, understood, debugged, enhanced and updated by others in the future.

From the above methods and examples it is clear that, NFAs, with or without λ-transitions, are equivalent to DFAs. For every NFA we can construct an equivalent DFA through subset construction. Every DFA is already an NFA in a trivial sense of not having any non-deterministic transitions. In the case of finite automata, non-determinism does not add to the computing power of the machine since these simple automata do not have any memory elements. Various combinations of the states of the NFA can be captured by corresponding states in an equivalent DFA. The DFA may have more states than the NFA but it can still compute anything that the NFA can. In later chapters we will see that in the case of more powerful classes of computing machines and formalisms, non-determinism does add to the computing power of the machine or formalism.

3.5 A Method for Minimizing Automata

Finite automata are not unique for a given problem. We can often construct multiple automata with different numbers of states and transitions that compute the same thing. Some of them may have redundant or useless states and transitions. For example, there may be states that are not even reachable from the start state or two states may be identical in terms of their behaviors and incoming and outgoing arrows. This is especially true in the case of DFAs obtained by converting from NFAs; such DFAs often contain many pairs of equivalent states and redundant transitions.

It is useful to have a method for detecting such redundancies and reducing a given automaton to its most minimal equivalent. Let us look at a method for minimizing an automaton by reducing the number states and consequently also the number of transitions in it. The method presented below works only for deterministic automata. This is not a problem since we already know how to convert an NFA to an equivalent DFA.

Minimizing a DFA is all about identifying duplicate states and collapsing them to a single state. This will automatically also reduce the number of transitions in the automaton. *How do we identify a pair of duplicate states?* This is rather difficult since we cannot simply examine the transitions in the immediate vicinity of the states to decide whether they are equivalent or not. Instead, we apply the technique of negation. That is, we first eliminate all the pairs of states that are not equivalent. Any pair that is left must be a pair of duplicates!

The method is based on the idea of *distinguishable* and *indistinguishable states*. Two states are distinguishable if, for a given input symbol, they take the automaton to states that are known to be different from one another. For example, states q_1 and q_2 are distinguishable if, for the input symbol 0,

q_1 takes the automaton to a final state but q_2 takes the automaton to a non-final state. They are clearly distinguishable since one state results in accepting the input and the other results in rejecting the input. In other words, we have the following rules for distinguishing a pair of states:

1. No final state can be equivalent to a non-final state since an input string that makes the automaton reach the final state will be accepted while one that makes it reach a non-final state will be rejected.
2. Two states are not equivalent if, on the same input symbol, they take the automaton to a pair of states that are already known to be distinguishable.

These rules are used in a procedure known as *marking distinguishable states*.

Algorithm

First delete any state that is unreachable from the start state;
For each pair of states where one is a final state and the other is non-final,
    mark them as distinguishable;
For each pair of states q_i and q_j
    For each symbol in the alphabet,
        If q_i takes the automaton to q_m and q_j to q_n and
        If q_m and q_n are already marked as distinguishable,
        Then mark q_i and q_j as distinguishable;
Repeat the above until no more pairs can be marked;
All the pairs of states that are not marked are indistinguishable;
Collapse indistinguishable pairs to single states and merge their transitions.

An efficient way to keep track of all the marked pairs of states is to construct a two-dimensional matrix of states as shown in Table 3.3. Note that just the lower half triangle of the matrix is sufficient since a pair of marked states is distinguishable in both directions. Use of such a matrix is sometimes called *table filling*.

EXAMPLE 3.10

Consider the NFA shown in Fig. 3.17 for accepting input strings that contain either the keyword 000 or the keyword 010. If we convert this to an equivalent DFA using the method of subset construction, we obtain the DFA shown in Fig. 3.18. The DFA has four final and four non-final states. Let us now try to minimize this DFA using the above method.

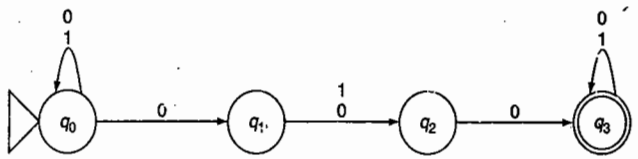


FIGURE 3.17 NFA for multiple keyword search.

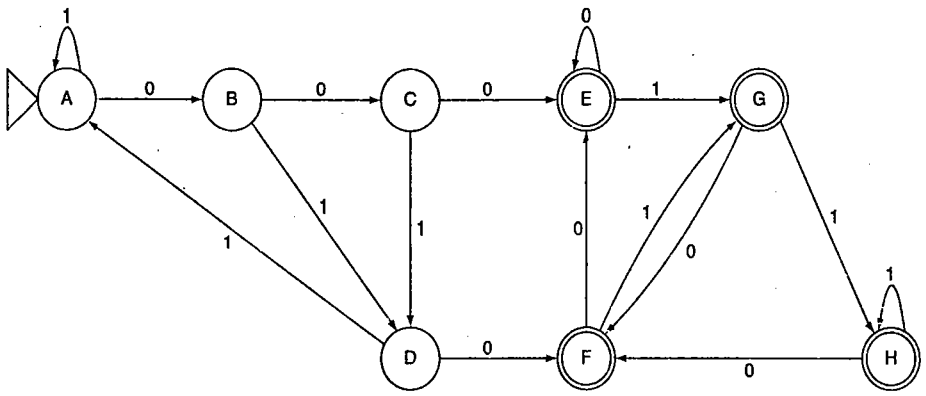


FIGURE 3.18 DFA for multiple keyword search.

We begin filling Table 3.3 with the "fnf" markers which stand for final-non-final pairs of states. Since A, B, C and D are non-final and E, F, G and H are final states, we can quickly enter "fnf" in the 16 cells. Next, we apply the second rule and mark the pairs A-C, A-D, B-C and B-D as distinguishable. For each of these pairs, on the symbol 0, the first state takes the automaton to a non-final state while the second one takes it to a final state. For example, A on 0 goes to B, a non-final state, but C on 0 goes to E, a final state. Hence the four cells are filled with the marker "on 0" in Table 3.3.

The pairs A-B and C-D are a bit more involved. A and B on 0 both go to non-final states but the states they reach namely, B and C respectively, are already marked as a distinguishable pair. Hence A and B are also distinguishable. This is marked in Table 3.3 as "on 0 (B×C)." C and D both take the automaton to a final state for the symbol 0. They are indistinguishable as far as the symbol 0 is concerned. However, for the symbol 1, C goes to D and D goes to A. A and D are already marked as distinguishable and therefore C and D are also distinguishable.

TABLE 3.3 Marking Distinguishabie States for Minimizing a DFA

B	on 0 (B×C)						
C	on 0	on 0					
D	on 0	on 0	on 1 (D×A)				
E	fnf	fnf	fnf	fnf			
F	fnf	fnf	fnf	fnf	?		
G	fnf	fnf	fnf	fnf	?	?	
H	fnf	fnf	fnf	fnf	?	?	?
States	**A**	**B**	**C**	**D**	**E**	**F**	**G**

At this point, we are left with just all the pairs of final states. Examining the DFA in Fig. 3.18, we find that all the transitions from any of the final states go to final states. No matter which rule we try, we are not going to be able to mark any pair of the final states as distinguishable. Thus, in this example, we find that all the non-final states are distinguishable but all the four final states are unmarked (i.e., indistinguishable). The minimal DFA is obtained simply by collapsing the four final states into a single state.

EXAMPLE 3.11

We need an example with several pairs of indistinguishable states to see how the final steps of collapsing equivalent states and merging of transitions work. Let us try to minimize the automaton shown in Fig. 3.19. Table 3.4 shows the markings and Fig. 3.20 the resulting minimized DFA. The steps followed in minimizing this automaton are:

1. We note that the given automaton is deterministic and hence our method is applicable.
2. We observe that the state H is unreachable from the start state since it has no incoming arrow. As such we remove state H and all arrows coming out of it from any further consideration.
3. D is the only final state. We mark all other states as distinguishable from D (with the mark "fnf").
4. Next, we are able to mark the pairs A-C, A-E, B-C, B-E, C-F, C-G, E-F and E-G on the symbol 0 since C and E reach the final state on 0 and A, B, F and G reach a non-final state on 0.
5. We are able to mark A-B because, on the symbol 1, they take the automaton to A-C a pair that is already marked as distinguishable.
6. Similarly, A-F, B-G and F-G can be marked on 1 because of already marked pairs.

TABLE 3.4 Marking Distinguishable States for Minimizing a DFA

B	on 1 (A×C)						
C	on 0	on 0					
D	fnf	fnf	fnf				
E	on 0	on 0	?	fnf			
F	on 1 (A×E)	?	on 0	fnf	on 0		
G	?	on 1 (C×G)	on 0	fnf	on 0	on 1 (E×G)	
H							
States	**A**	**B**	**C**	**D**	**E**	**F**	**G**

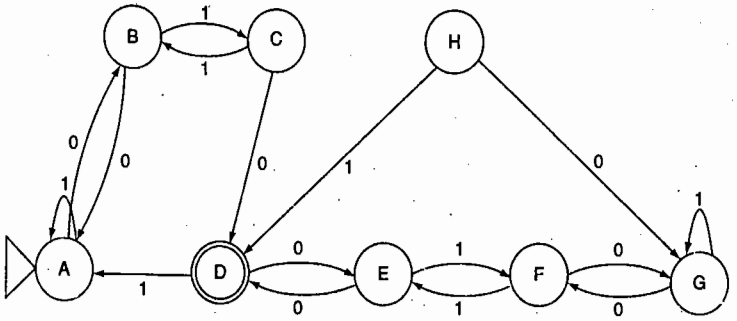


FIGURE 3.19 DFA for minimization.

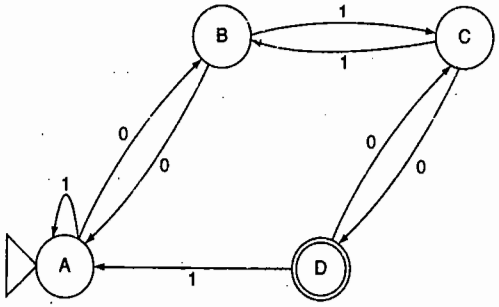


FIGURE 3.20 Minimized DFA.

In the end, A-G, B-F and C-E remain unmarked. Their outgoing arrows are all similar. When there are several pairs of indistinguishable (i.e., unmarked) states, not all of them are indistinguishable across pairs. That is, we cannot reduce them all to a single state. Each pair of indistinguishable states can be collapsed to a single state (e.g., G merges with A, F merges with B and E merges with C). In other words, indistinguishability divides the states into *equivalence classes*. Merging of states and transitions can be done one pair (or equivalence class) at a time.

It is also possible to start with a transition table of a machine and minimize it directly from the table without first rewriting it as a transition diagram. We also see from Tables 3.3 and 3.4 how the state matrix is organized. To avoid duplicate entries in the table, we must remember that there is no need for a row for the first state or a column for the last state of the DFA in the table.

A final question before we end our study of finite automata: is it always possible, given a finite automaton, to describe succinctly the set of input strings that it accepts? We address this in the next chapter

wherein we introduce the idea of a formal language. In particular, the languages or the sets of strings accepted by finite automata are called *regular languages.*

States in an automaton which are indistinguishable from one another constitute an *equivalence class.* The above method for minimizing an automaton essentially partitions the states in the original automaton into a set of equivalence classes and then collapses each equivalence class into a single state in the minimized automaton. Indistinguishable states which fall into the same equivalence class take the automaton to an accepting state for the same set of (sub-) strings. This method of minimizing an automaton was made possible by a well-known result called *Myhill–Nerode Theorem* (covered in Appendix B) which says that a language is regular if and only if an automaton that accepts the language has a finite number of equivalence classes of its states. We explore properties of regular languages further in Chapter 6.

3.6 Finite-State Transducers

A finite automaton that not only accepts an input string but also produces useful output is called a *finite-state transducer.* Finite-state transducers have been used in a variety of practical applications ranging from electronic circuits to software testing. Two types of finite-state transducers known as *Moore and Mealy machines* are popular.

3.6.1 Moore Machine

A *Moore machine* is a finite-state machine whose output is determined only by its current state. In addition to the standard elements of a finite automaton, a Moore machine has a separate output alphabet and an output function that maps each state to a symbol in the output alphabet. Whenever the Moore machine reaches a particular state, it outputs the corresponding symbol from its output alphabet. Moore machines are usually clocked machines, that is, they change states and produce output synchronously with a clock.

EXAMPLE 3.12

Figure 3.21 shows a Moore machine that is a full adder for adding two binary numbers. It is assumed that corresponding pairs of digits of the two numbers to be added are presented to the machine as input. For example, 00 means that the corresponding digits of both numbers are 0. States q_0 and q_1 output a 0 and states q_2 and q_3 output a 1. The output of a state is shown in a small box attached to the upper right-hand side of the state. States q_0 and q_2 do not have a carry bit while q_1 and q_3 have a carry bit. If the input is 00 in state q_0 the output is 0 and the machine stays in state q_0 since there is no carry bit introduced by the adding 0 to 0. If the input is 01 or 10 in state q_0 then the output is 1 and there is no carry bit. The machine therefore goes to state q_2. If the input in state q_0 is 11, the output is 0 but a carry bit is introduced. The machine therefore goes to state q_1. All the other transitions in the Moore machine can be understood in similar ways.

It may be noted that this is different from the formal language of addition which cannot be handled by finite-state machines (see Example 6.5 in Chapter 6).

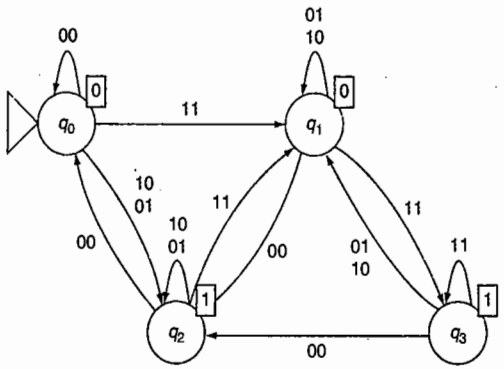


FIGURE 3.21 Moore machine binary adder.

3.6.2 Mealy Machine

In a *Mealy machine*, the output depends not only on the current state of the automaton but also on the input symbol. It behaves as though the output is determined by a transition to the state rather than the state itself. If we add an output symbol to each transition in a finite automaton, we get a Mealy machine. Mealy machines typically have fewer states than comparable Moore machines because they can have different outputs for the same state depending on which input symbol was consumed in bringing the machine to the state.

EXAMPLE 3.13

Figure 3.22 shows a Mealy machine binary adder that is equivalent to the Moore machine in Fig. 3.21. In this machine, there is no fixed output symbol for a state. There are only two states: q_0 with no carry bit and q_1 with a carry bit. The output is now defined by the labels on the transitions. For example if the input in state q_0 is 00, the output is 0 and the machine stays in state q_0. If the input is 01 or 10, it still stays in the same state but the output is now 1. The Mealy machine goes from state q_0 to state q_1 only upon encountering the input 11.

In state q_1, the machine stays in the same state if the input is 01, 10 or 11 because the carry bit continues to exist after adding each of these inputs. If the input is 00 though, the carry bit becomes the output 1 and the machine returns to state q_0.

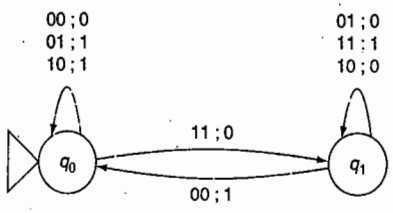


FIGURE 3.22 Mealy machine binary adder.

EXAMPLE 3.14

Figure 3.23 shows a simple Mealy machine for a translator machine that changes the symbols in the input string to a different alphabet, from {0, 1, 2} to {*a*, *b*, *c*}, for example.

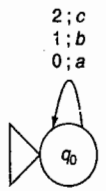


FIGURE 3.23 Mealy machine translator.

It may be noted that Moore and Mealy machines do not have the concept of a final or accepting state. The machines keep producing output as and when input symbols are encountered and may halt at any point in any state.

EXAMPLE 3.15

Figure 3.24 shows a Mealy machine for computing the exclusive-OR of the two previous symbols in the input. Such a machine is also called an *edge detector* since it detects a change in the input value from 0 to 1 or 1 to 0 (thereby indicating an edge in the pixels of an image). For example, given the input string 0011101010, the output is 0010011111.

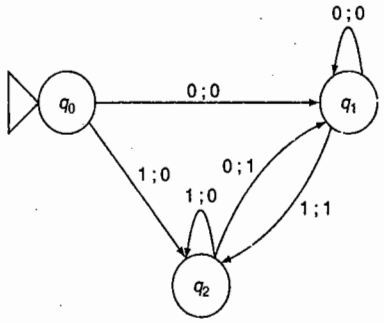


FIGURE 3.24 Mealy machine edge detector.

From its state state q_0, the machine goes to state q_1 to remember that the previous input symbol was a 0 and to state q_2 to remember that the input symbol was a 1. In either case the output is 0 since the machine has not yet encountered an edge. In state q_1, if the input continues to be 0, the machine stays in the same state and the output continues to be 0. If the input is 1 however, there is a change in the input symbol and an edge has been detected. The machine outputs 1 and goes to state q_2 to remember that the input symbol was a 1. It stays in q_2 as long as the input is 1 and continues to output 0. If the input changes to 0, an edge has been detected and the machine outputs 1 and goes to state q_1 to remember the input symbol 0.

3.7 Key Ideas

1. A non-deterministic finite automaton can have more than one transition from the same state for the same input symbol.
2. It can guess the right choice for accepting an input string. Equivalently, it can try out all the choices in parallel to see if any one leads to acceptance of the input string.
3. Non-deterministic automata are often easier for us to design.
4. Non-deterministic automata can also change states without consuming any input symbol. Non-deterministic finite automata with such λ-transitions make it further easy for us to design them for a given problem. They also help us combine two or more automata to solve a more complex problem.
5. Non-deterministic finite automata are strictly equivalent to deterministic ones. They have the same computing power as deterministic ones, in terms of the range of problems they can solve (or the range of input strings they can handle).
6. The method of subset construction is used to convert non-deterministic finite automaton to an equivalent deterministic finite automaton.
7. A deterministic finite automaton (DFA) obtained from an equivalent non-deterministic finite automaton (NFA) usually has more states than the NFA. In the worst case, the number of states in the resulting DFA can be exponential in the number of NFA states.
8. A deterministic finite automaton (DFA) can be minimized by identifying and merging indistinguishable states. Indistinguishable states denote the same equivalence class of (sub)strings and can be merged into a single state. The minimal number of states in an equivalent DFA is the total number of equivalence classes of strings. This number must be finite for a regular language according to the Myhill–Nerode theorem.
9. Book-keeping in minimizing a deterministic finite automaton is made easy by applying a table-filling technique to mark all distinguishable pairs of states.
10. Unlike finite automata which merely accept or reject an input string, finite-state transducers such as Moore and Mealy Machines produce an output while processing the input string.

3.8 Exercises

A. Construct a non-deterministic finite automaton (NFA) for each of the following (assuming that the shortest acceptable strings are long enough to accommodate all the requirements).

1. Binary strings that begin with 11 and end with 11 or begin with 00 and end with 00.
2. Binary strings starting with 000 or ending with 111 (or both).
3. Binary strings in which the sum of the last four digits is odd (e.g., 00101011 but not 00101001).
4. Strings over $\{a, b, c\}$ that contain at least one a and at least one b.

5. Strings over $\{a, b\}$ that contain at least three as or at least two bs.
6. Binary strings in which the first part of each string contains at least four 1s and the second part contains at least three 0s.
7. Strings over $\{a, b\}$ where the last two symbols in each string is a reversal of the first two symbols (i.e., last symbol = first symbol and penultimate symbol = second symbol). The NFA must contain only 10 states (not including reject states).
8. Binary strings of any length with alternating 0s and 1s. The NFA must have just three states (not including reject states). How many states does an equivalent minimal deterministic finite automaton have?

B. For the following problems, a transition table is specified. Convert the given table to non-deterministic finite automaton (NFA) diagram and then to an equivalent deterministic finite automaton.

9. The NFA specified by:

State	Input = a	Input = b	Input = c	λ
$\rightarrow q_0$	$\{\}$	$\{q_1\}$	$\{q_2\}$	$\{q_1, q_2\}$
q_1	$\{q_0\}$	$\{q_2\}$	$\{q_0, q_1\}$	$\{\}$
$*q_2$	$\{\}$	$\{\}$	$\{\}$	$\{\}$

10. The NFA specified by:

State	Input = a	Input = b	λ
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_1\}$	$\{\}$
q_1	$\{q_2\}$	$\{q_1, q_2\}$	$\{\}$
$*q_2$	$\{q_0\}$	$\{q_2\}$	$\{q_1\}$

C. Convert the following non-deterministic finite automata to equivalent deterministic finite automata (DFAs) (and minimize the resulting DFAs).

11. The NFA shown in Fig. 3.25.

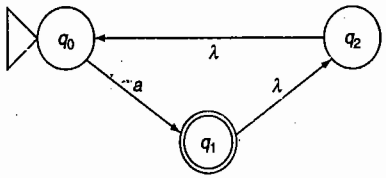


FIGURE 3.25

12. The NFA shown in Fig. 3.26.

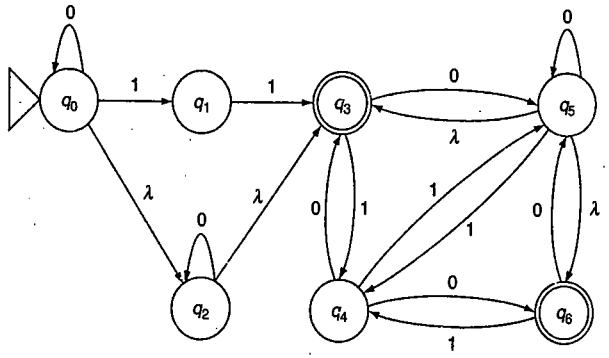


FIGURE 3.26

13. The NFA shown in Fig. 3.27.

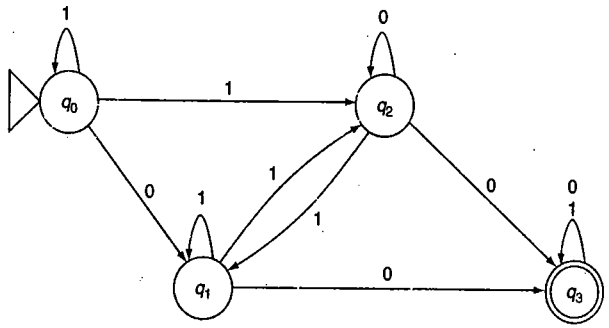


FIGURE 3.27

D. Minimize the following deterministic finite automata (DFAs).

14. The DFA shown in Fig. 3.28.

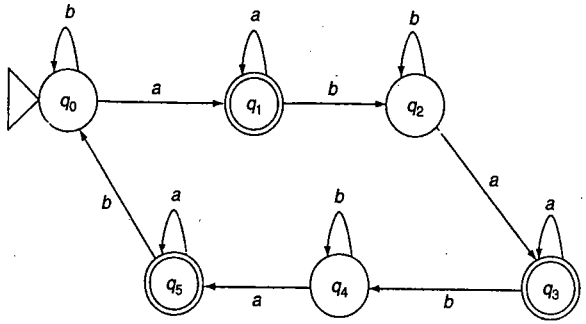


FIGURE 3.28

15. The DFA shown in Fig. 3.29.

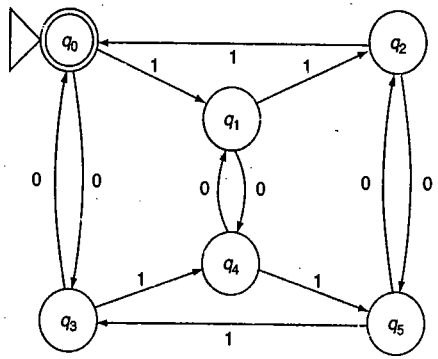


FIGURE 3.29

16. The DFA shown in Fig. 3.30.

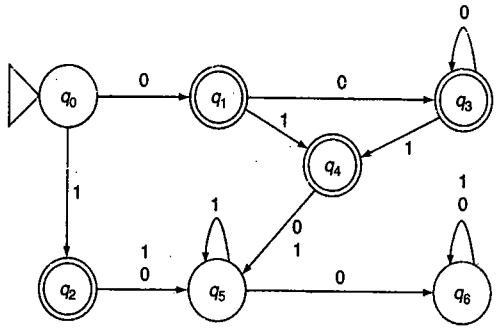


FIGURE 3.30

17. The DFA shown in Fig. 3.31.

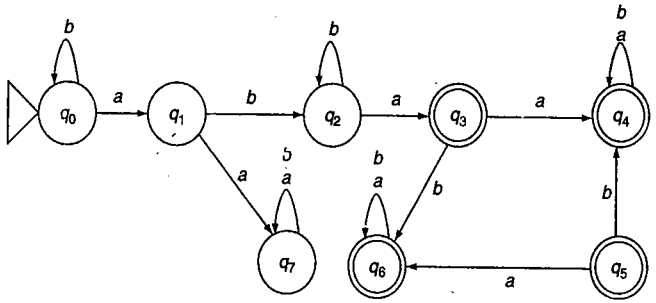


FIGURE 3.31

18. What happens if we apply the method of subset construction to a deterministic finite automaton instead of non-deterministic finite automaton?
19. Design a Moore machine that takes a binary string as input and outputs after each symbol the remainder obtained when the input thus far is divided by 5 (treating it as a non-negative binary number). The output alphabet is {0, 1, 2, 3, 4}.
20. Design a Mealy machine to do the same (as in Exercise 19).
21. Design a Moore machine to detect a run in the input, that is, sequences of two or more identical symbols. For example, given the input *abaabbbabaa*, the output shall be 00010110001.
22. Design a Mealy machine to do the same (as in Exercise 21).

Chapter Four

Regular Languages and Expressions

Learning Objectives

After completing this chapter, you will be able to:

- Learn what is a formal language and how it is related to an automaton.
- Learn to construct simple regular expressions.
- Learn to convert a regular expression to an equivalent automaton.
- Learn to convert an automaton to a regular expression.
- Learn a method to compare two regular expressions or finite automata to determine their equivalence.
- Learn to construct regular expressions for user data validation.

Having covered finite automata in the previous two chapters, this chapter introduces the idea of formal languages and how they are related to automata. The bulk of the chapter is devoted to regular expressions. Readers learn how to construct them directly or from automata. The *mantras* introduced in Chapter 2 for finite automata are now modified to facilitate the construction of regular expressions. The use of regular expressions in modern programming languages is illustrated through examples from Web page design for user data validation.

4.1 *The Idea of* Formal Languages

How do we describe the set of all input strings accepted by a given finite automaton? In all the examples in Chapters 2 and 3, we started with a description, in English, of the set of strings to be accepted by the automaton before constructing it. Is it always possible to describe the set of acceptable inputs concisely (i.e., without having to enumerate all the strings)?

We call any set of strings a *formal language*. A natural language, such as English, can also be considered to be a set of strings. In this view, English is nothing but the set of all possible valid sentences in English. That is, the English language is the set of all grammatically correct (and meaningful) sentences that one can construct using the words and the rules of grammar of English. Similarly, Java™, a programming language, is also a formal language, being the set of all valid Java™ programs that one could ever write.

Any language, formal or natural, is built on a set of primitive symbols (or sounds, in the case of a spoken language) which we call the *alphabet* of the language. Strings of symbols from the alphabet are

put together to form words, phrases, sentences and longer units of text (or speech) in natural languages. In the case of formal languages, we do not distinguish between different kinds of strings such as words or sentences. Rather, a formal language is merely a set of strings each of which is a sequence of symbols from the alphabet of the language.

For convenience, we normally use a small set of symbols as the alphabet in our examples. We have already seen in previous chapters alphabets such as $\{0, 1\}$ and $\{a, b\}$. Both natural languages and modern computer languages have much larger alphabets: the ASCII character set, the Unicode, Chinese pictographs and so on. In any case, the alphabet is always a finite set; there can never be an infinite number of unique symbols in an alphabet.

Since the alphabet is a finite set of symbols, if we place a limit on the maximum length of strings in a formal language, the language can have only a finite number of strings. We call such languages *finite languages*. For example, if there are only four symbols in the alphabet and the maximum length of strings in a language L is 10, then the language can have at most four strings of length 1, plus 4^2 strings of length 2, plus 4^3 strings of length 3, ..., plus 4^{10} strings of length 10. The language may also be considered to include one special string of length zero, namely, the null string λ. In any case, this total number is still a finite number and L is a finite language.

For reasons that will become clear later, we normally deal with *infinite languages*, that is, formal languages that are infinite sets of strings. The only way that a set of strings over a finite number of symbols can be infinite is by having no limit on the length of its strings. In other words, infinite formal languages contain strings that are infinitely long. Although in practice it is true that we can never store a string of infinite length in any real computer, for the purposes of understanding abstract notions of computation and for building theoretical models of computing machines, we do not impose any artificial limit on the length of a string. For example, the set of all even numbers over the binary alphabet is an infinite set, that is, an *infinite formal language*.

Natural languages are also infinite. English is an infinite language because there are infinitely many sentences that one can construct in English. A rather silly way to illustrate this point is to show that sentences such as "I have 1 apple," "I have 2 apples," "I have 3 apples" and so on are all valid sentences in English. And there is no limit on the number of such sentences you can construct in English.

This, however, does not mean that every string of symbols from an alphabet belongs to every language over the alphabet. Many strings of words are not valid English sentences. For example,

"A baby are sleeps."
"I slept the baby tomorrow."

and Chomsky's famous example

"Colorless green ideas sleep furiously."

are all strings that are not valid in English. Similarly, the string 101 is not a string in the formal language of even binary numbers. Any Java™ program that has syntactic or semantic errors is not a member of the language. From now on, when we say language, we mean a formal language.

In natural languages such as English, a set of rules known as the *grammar of the language* determines whether a given sentence is valid or not. For example, the invalid sentences above violate rules of number, tense or semantic agreement. Formal languages also have *grammars*. We will study them in Chapters 5 and 7 in sufficient detail.

In this chapter, we introduce a very simple version of grammars for formal languages known as *regular expressions* (RegEx). We will see below (Sec. 4.3) how the entire behavior of a finite automaton, whether it is deterministic or non-deterministic, can be described very concisely using a regular expression. Regular expressions are also highly practical constructs. They are supported in most modern programming languages and are extremely useful in tasks such as pattern matching and user input validation. We show a few examples at the end of this chapter.

4.2 Languages of Automata

Every finite automaton accepts a set of strings. As such, every finite automaton maps to a formal language. We call this the *language of the machine*. That is, the set of all input strings accepted by a finite automaton (or other computing machine) M is the language of the machine. We often denote this as M.language. Given a formal language, however, there can be more than one finite automaton that accepts the (set of strings in the) language. We have already seen examples of deterministic and non-deterministic automata and even redundant and minimal automata that accept the same language.

What about the set of all inputs rejected by a given automaton? Isn't this set also a formal language? We call this language the *complement* of the set of all strings accepted by the machine. A formal language and its complement together constitute the set of all possible strings over the alphabet. Note that all three sets are often infinite. For example, the complement of the set of all even binary numbers is the set of all odd binary numbers. Their union is the set of all binary numbers.

4.3 Regular Expressions

Descriptions in English such as "all strings over the alphabet $\{a, b\}$ in which the first and last symbols are the same" are not precise or concise enough to be used in programming. For this purpose, a popular and efficient notation known as *regular expression* has been developed. A regular expression is like an arithmetic expression in a programming language. However, it has its own operators instead of the operators of arithmetic. Every regular expression is a concise representation of a formal language. In other words, a regular expression represents a set of strings over the alphabet.

The most primitive regular expressions are the individual symbols in the alphabet themselves (including the special symbol λ for the null string). For example, a is a regular expression that stands for the set with a single string $\{a\}$. Primitive regular expressions can be put together to construct more complex expressions as follows:

1. $e_1 + e_2$ is a regular expression, where e_1 and e_2 are regular expressions; the + operator represents *set union*, that is, any string from the set represented by e_1 *or* from the set represented by e_2. The language of $e_1 + e_2$ is the union of the languages of e_1 and e_2.
2. $e_1.e_2$ is a regular expression; the . operator stands for *concatenation*. $e_1.e_2$ represents the set of all strings obtained by concatenating any string from e_1 with any string from e_2.
3. (e_1) is a regular expression where the parentheses are used only to eliminate any ambiguity in scope or precedence of other operators.
4. e_1^* is a regular expression; the * operator, also known as the *closure* of the expression, stands for the set of all strings obtained by repeatedly concatenating any string from e_1 *zero* or *more times*.

5. Any expression obtained by composing any of the above operators applied in any order a finite number of times is also a regular expression.

Such a definition showing how to construct more complex expressions from primitive expressions is called a *recursive definition.* Thus, the regular expression (often abbreviated as RegEx)

1. $a + b$ stands for the set of two strings or the language $\{a, b\}$.
2. $a.b$ stands for the set containing a single string $\{ab\}$.
3. a^* stands for $\{\lambda, a, aa, aaa, aaaa, \ldots\}$.

In this last language, the null string λ is obtained by repeating a zero times and the other strings are obtained by repeatedly concatenating the symbol a one or more times.

What about $(a + b)^*$?

This RegEx stands for the formal language $\{\lambda, a, b, aa, ab, ba, bb, aaa, aab, \ldots\}$. If the alphabet $\Sigma = \{a, b\}$, then the above set is the set of all possible strings over this alphabet. This set is sometimes denoted by Σ^* to indicate that it is the set of all strings obtained by repeating any symbol from the alphabet in any order zero or more times.

4.4 Constructing Regular Expressions

Let us learn the art of constructing regular expressions for given languages by working through a series of examples.

EXAMPLE 4.1

Let us construct a regular expression for all even binary numbers. These numbers must end with a 0 and therefore the RegEx must have a 0 at the end. Before that, 0s and 1s can appear in any order. The RegEx for zero or more repetitions of the symbols 0 and 1 is $(0 + 1)^*$ and thus the RegEx for even binary numbers is

$$(0 + 1)^*.0$$

Sometimes, we omit the concatenation operator and write

$$(0 + 1)^*0$$

We had previously constructed automata for this same language: a deterministic one in Fig. 2.3 and a non-deterministic one in Fig. 3.4. As you can see from those diagrams, the RegEx seems closer to the non-deterministic finite automaton (NFA) than the deterministic finite automaton (DFA). Also, the closure operator * appears to be equivalent to the loop (jolly ride) in the NFA. We will study the equivalence of automata and regular expressions later in this chapter.

EXAMPLE 4.2

Consider the language of strings that begin with at least two as and end with an even number of bs (with nothing else in between). The RegEx for two as is $a.a$ and for two or more as is $a.a.(a)^*$, that is, two as followed by zero or more as. The simplest string that ends with an even number of bs is $b.b$. The RegEx for all strings with an even number of bs is obtained by repeatedly concatenating the pair of bs, that is, $(b.b)^*$. Putting all these together, the RegEx for the given language is

$$a.a.(a)^*.(b.b)^*$$

or

$$aa(a)^*(bb)^*$$

This language is sometimes also described formally as $a^n b^m$ where n is ≥ 2 and m is even, although such a representation is not a RegEx.

EXAMPLE 4.3

Let us see if we can use the same method to obtain a RegEx for the language of all strings over $\{a, b\}$ that begin with as and end with bs (with nothing else in between) and are of even length (i.e., the total number of as and bs is even). All strings with an even number of as can be represented by the RegEx $(a.a)^*$.

How do we get a RegEx for all strings with an odd number of a s?

It is simply an a followed by an even number of as, that is, $a.(a.a)^*$. For the given language, we observe that if a string has an even number of as and an even number of bs, then it meets all the requirements and belongs to the language. The only other type of string that can belong to this language is one that has an odd number of as and an odd number of bs. Strings with an odd number of as and an even number of bs or with an even number of as and an odd number of bs are of odd length overall and do not belong to the language.

A RegEx for the strings that have even numbers of both symbols is

$$(a.a)^*.(b.b)^*$$

Similarly, a RegEx for strings that have odd numbers of both symbols is

$$a.(a.a)^*.b.(b.b)^*$$

Combining these two, the complete RegEx for the given language is

$$(aa)^*(bb)^* + a(aa)^*b(bb)^*$$

EXAMPLE 4.4

Our next example is the formal language containing all binary strings with exactly one pair of consecutive 0s in them. 00, 100, 1010100101 are examples of strings in this language; 0, 10, 101, 000, 00100, 10001 are examples of strings that are not in this language (i.e., they are in the complement of this language). Note that the string 000 has two consecutive pairs of 0s – the first and the second 0s, and the second and the third 0s.

What is our method for constructing a RegEx for this language?

Let us begin with the simplest string in this language, namely, the string 00. A RegEx for just this string is 0.0.

What else can be present in strings that belong to the language?

Anything can be present before or after the 00 as long as it does not produce another pair of consecutive 0s.

What is a RegEx for binary strings that do not contain consecutive 0s?

We can try (1.0)*. In fact, nothing prevents us from allowing consecutive 1s without any 0s and we can enhance the RegEx to

$$(1 + (1.0))^*$$

This expression says that we can repeat either a 1 or a 10 any number of times. In other words, every 0 must be preceded by a 1 (so that there can never be consecutive 0s).

Now we can write the RegEx for the given language as

$$(1 + (1.0))^*.0.0.(1 + (1.0))^*$$

that is, anything with no consecutive 0s followed by exactly one pair of consecutive 0s followed in turn by anything not containing consecutive 0s. However, there is a small bug in this expression: it allows strings such as 1.0.0.0.1 that cannot belong to the language. In order to prevent this, we can re-write the first part of the RegEx to obtain the following final RegEx:

$$(1 + (01))^*00(1 + (10))^*$$

EXAMPLE 4.5

Consider the set of all binary strings wherein the fifth last symbol is 0 (see Fig. 3.8 for an NFA for this language). A RegEx for this language is fairly easy to construct:

$$(0 + 1)^*.0.(0 + 1).(0 + 1).(0 + 1).(0 + 1)$$

The last four symbols can be either 0 or 1; the fifth symbol from the end must be 0; before that, any sequence of 0s and 1s can appear. This RegEx mirrors the NFA in Fig. 3.8 quite well.

EXAMPLE 4.6

Consider binary strings containing no more than two 0s. There are three ways of having no more than two 0s:

1. Having no 0 at all.
2. Having exactly one 0.
3. Having exactly two 0s.

We simply write separate RegEx's for each of the three cases and combine them with the + operator (i.e., by saying either the first one or the second one or the third one):

$$1^* + 1^*.0.1^* + 1^*.0.1^*.0.1^*$$

Of course, this expression is not unique. By factoring out common sub-expressions, it can also be written as

$$1^*(\lambda + 0.1^*(\lambda + 0.1^*))$$

EXAMPLE 4.7

Consider the formal language of all strings over the alphabet $\{a, b\}$ whose length is a multiple of 3.

What is the simplest string in this language?

It is one of *aaa*, *aab*, *aba*, *abb*, *baa*, *bab*, *bba*, *bbb*. The only way to get longer strings whose length is a multiple of 3 is to concatenate some of these strings repeatedly. As such, the final RegEx for this language is

$$(aaa + aab + aba + abb + baa + bab + bba + bbb)^*$$

Factoring out common sub-expressions results in a much more concise and elegant RegEx for this language:

$$((a + b).(a + b).(a + b))^*$$

EXAMPLE 4.8

A language is defined as the set of all strings over $\{a, b, c\}$, where each string contains at least one a and at least one b. A string that contains at least one a must have an a somewhere which can be preceded by anything and also followed by anything:

$$(a+b+c)^*.a.(a+b+c)^*$$

Similarly, a string that contains at least one b can be represented by

$$(a+b+c)^*.b.(a+b+c)^*$$

How do we combine these two?

If we concatenate them while eliminating some redundancy in the middle, we get

$$(a+b+c)^*.a.(a+b+c)^*.b.(a+b+c)^*$$

But this assumes that the mandatory a comes before the mandatory b. This RegEx does not include, for example, the simple string ba. In order to eliminate this bias, we have to consider both cases: where the required a comes before the b and where the b comes before the a:

$$(a+b+c)^*.a.(a+b+c)^*.b.(a+b+c)^* + (a+b+c)^*.b.(a+b+c)^*.a.(a+b+c)^*$$

EXAMPLE 4.9

Sometimes a formal language L is defined mathematically as, for example,

$$L = \{uvw \mid u, w \text{ belong to } \Sigma^* \text{ and } |v| = 2\}$$

where $\Sigma = \{a,b\}$. Here, u, v and w are strings themselves over the same alphabet and uvw is a concatenation of u, v and w in that order and the length of v is 2. Since Σ^* denotes the set of all possible strings over the alphabet, the specification merely says that u and w are any strings (of any length) over the alphabet. A RegEx for u or w is $(a+b)^*$. v is a string of length 2 and a RegEx for v is

$$(a+b).(a+b)$$

The complete RegEx for the language L is

$$(a+b)^*.(a+b).(a+b).(a+b)^*$$

EXAMPLE 4.10

Given a RegEx, how do we describe the language of the RegEx? For example, what is the language of the following expression?

$$((aa)^*b(aa)^*) + (a(aa)^*ba(aa)^*)$$

The expression clearly has two parts: strings of the first kind or strings of the second kind. The first kind must contain a b. Before or after the b, pairs of a can be repeated zero or more times. Hence, it is the set of all strings that contain exactly one b and an even number of as on either side of b. On the same lines, the second part is the set of all strings that contain one b and an odd number of as on either side. Thus, the entire language of the above RegEx is the set of all strings containing exactly one b and either an even number of as on either side of it or an odd number of as on either side of the b. Note that strings such as ba or $aabaaa$ are not in the language since they contain an even number of as on one side and an odd number of as on the other side of b.

Is there a more compact way of describing this language?

Yes, indeed. We can reason as follows. All strings in the language contain exactly one b. This symbol can be in an odd-numbered position from the beginning of the string or an even-numbered position. In the first case, there would be an even number of as before b and, as required by the RegEx, also an even number of as after the b. If the b is in an even-numbered position, there would

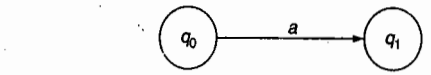
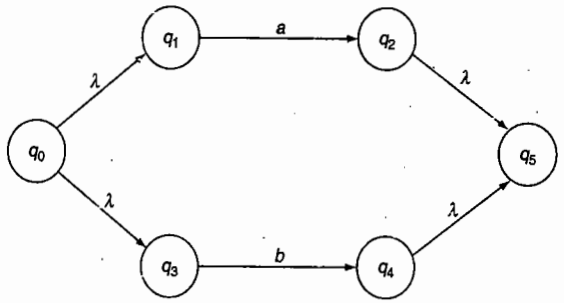
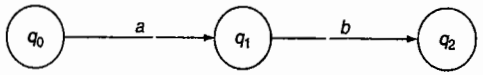
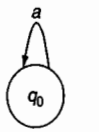
be an odd number of as before b and also an odd number of as after the b. In either case, the total number of as in the string is even. Thus, the language is merely *the set of all strings over {a, b} that contain exactly one b and an even number of a s.*

4.5 Converting Regular Expressions to Automata

As you may have observed from the above examples, writing regular expressions for a formal language is similar to constructing non-deterministic finite automata. We will now see that any RegEx can be converted to an NFA. We know from its definition that any RegEx is made up of one or more primitive expressions that are composed using the +, . and * operators, and optionally parentheses. Each of these constructs can be converted to an equivalent piece of an NFA as shown in Table 4.1.

Sometimes, we may have to introduce additional states or λ-transitions to ensure that the meanings of other states are not improperly affected by introducing the above mappings into the NFA. By applying the mappings repeatedly, any RegEx can be converted to an equivalent NFA.

TABLE 4.1 Mapping from Elements of Regular Expressions to NFA

A primitive RegEx a		A primitive RegEx is a transition in the NFA; the machine remembers the symbol by going to a different state.
$a+b$		A + operation is equivalent to two branches in the NFA connected through λ-transitions.
$a.b$		A concatenation in a RegEx is a sequence of two transitions in the NFA.
a^*		The * operator (closure) maps to a loop back to the same state (jolly ride).

EXAMPLE 4.11

For example, consider the RegEx from Example 4.3:

$$(aa)^*(bb)^* + a(aa)^*b(bb)^*$$

The given RegEx is in fact

$$((a.a)^*.(b.b)^*) + (a.(a.a)^*.b.(b.b)^*)$$

The equivalent NFA is shown in Fig. 4.1. The + operator translates to the two branches from the start state q_0. In the upper branch, states q_1–q_2–q_1 handle $(aa)^*$ and states q_3–q_4–q_3 handle $(bb)^*$. The concatenation in between them is handled by the λ-transition from q_1 to q_3, which is a final state. This λ-transition is essential; if we were to place both the loops $(aa)^*$ and $(bb)^*$ on the same state q_1, as shown in Fig. 4.2, the automaton could switch from one jolly ride to the other, thereby allowing bb to appear before aa. The automaton in Fig. 4.2 is, in fact, equivalent to the RegEx $(aa + bb)^*$ and includes strings such as $bbaa$ that are not in the language of the RegEx for the present problem.

The lower branch of the NFA in Fig. 4.1 is similar except that it has transitions for a (q_0–q_5) and b (q_5–q_7) instead of λs.

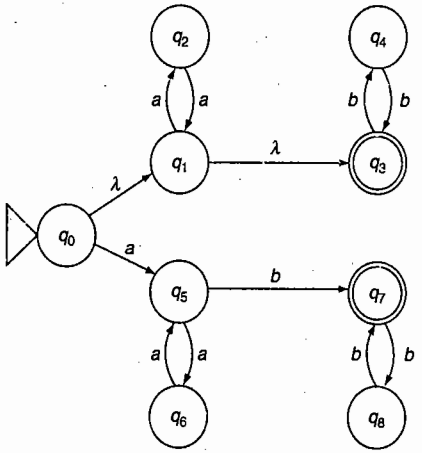


FIGURE 4.1 NFA from regular expression.

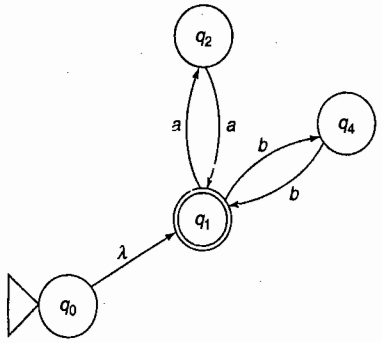


FIGURE 4.2 Incorrect NFA for regular expression.

EXAMPLE 4.12

Consider the RegEx from Example 4.4:

$$(1 + (01))^*00(1 + (10))^*$$

The NFA obtained from this by applying the above mappings is shown in Fig. 4.3. Transitions q_0–q_2 and q_2–q_3 take care of the mandatory 00. It is apparent from the diagram that in no other path through the automaton can a 0 be followed by another 0, thus meeting the requirement of the language. There are two jolly rides on q_0 and two more on q_3: q_0–q_0 on 1 and q_0–q_1–q_0 on 01 take care of the closure in the first part of the RegEx. Similarly, the loops q_3–q_3 on 1 and q_3–q_4–q_3 on 10 handle the other closure in the RegEx. We can also observe that non-determinism in the automaton is much easier to recognize than in the RegEx. In the case of a RegEx, whenever there is a loop and either a concatenation or another loop starting with the same symbol, there is non-determinism. For example, in the given RegEx, there is a loop starting with 0 and also a concatenation for 0.

$$(\ldots + (0\ldots))^*0\ldots$$

By the way, there is no separate formalism that could be called "deterministic regular expressions."

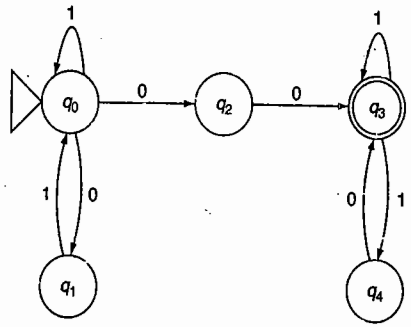


FIGURE 4.3 NFA from RegEx for exactly one 00.

The NFA in Fig. 4.3 has already been simplified after applying the mapping rules. Figure 4.4 shows a portion of the NFA, before simplification, for the first part of the RegEx, that is, $(1 + 01)^*$. Here there is one loop q_0–q_5–…–q_{10}–q_0 for the overall closure. Between q_5 and q_{10}, there are two parallel branches, one for 1 and the other for 01. The branch for 01 has a concatenation and therefore an intermediate state q_1 between q_8 and q_9. Eliminating all the redundant λ-transitions reduces the diagram to the one in Fig. 4.3.

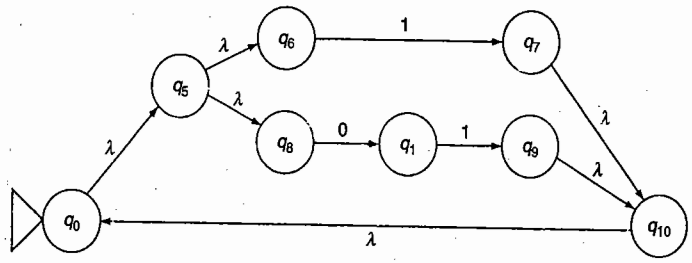


FIGURE 4.4 A portion of the NFA in Fig. 4.3 before simplifying.

In general, applying the mappings mechanically gives us an NFA with many states and lots of λ-transitions. We can simplify the NFA significantly by getting rid of unnecessary states and λ-transitions. We can also convert the non-deterministic finite automata (NFAs) to minimal deterministic finite automata (DFAs) by using the methods outlined in Chapter 3, or sometimes intuitively in more direct ways. Often, following our *mantras* for constructing a deterministic automaton (see Chapter 2) by figuring out what information the machine needs to remember (and thus what states it needs) gives us a simpler or more minimal DFA than what we get by converting from regular expressions.

4.6 Converting Automata to Regular Expressions

Converting from an automaton to a RegEx is not quite as simple as merely reversing the mappings used to convert a RegEx to an NFA. This is because there can be non-trivial loops and complex paths in the automaton all of which must be considered in the resulting RegEx. For example, consider the deterministic automaton shown in Fig. 2.11 [also shown in Fig. 4.8(a)] for the language of strings with an even number of as and even number of bs. There are just too many loops in this automaton for us to directly read a regular expression from it. The complexity in this example is due to fact that the as and bs can appear in any order unlike in Examples 4.2 and 4.3. In fact, the RegEx for this automaton is rather complex:

$$(aa + ab(bb)^*ba + (b + ab(bb)^*a)(a(bb)^*a)^*(b + a(bb)^*ba))^*$$

How do we obtain such a RegEx from an automaton?

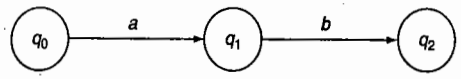
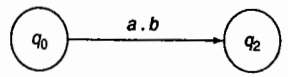
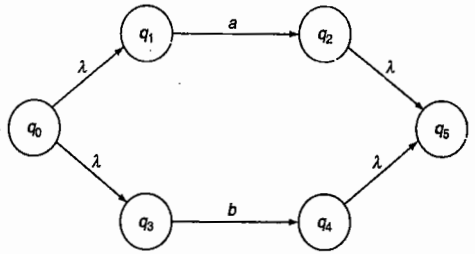
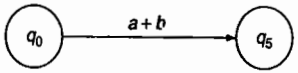
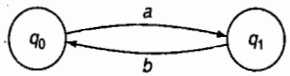
The basic idea is to eliminate states to reduce the automaton to a single state. We would like to eliminate all the "intermediate" states, that is, those that are neither the start state nor any final state.

States can be eliminated if we are allowed to place more than just an individual symbol on an arrow. What if we allow an entire regular expression to be the label of a transition in an automaton? Such an automaton is called a *generalized transition graph* indicating the generalization from the standard transition function δ to the one where regular expressions are allowed. Table 4.2 shows three simple configurations in which a state can be eliminated by introducing regular expressions.

Figure 4.5 shows a generic configuration in which an intermediate state can be eliminated. Figure 4.6 shows the regular expressions that result from eliminating the intermediate state q_4. In Fig. 4.5, there are two paths from state q_0 to state q_1: directly through the arrow labeled with the RegEx r_{01} or via state q_4 through r_{04} and r_{41}. In the second path, we can also take any number of jolly rides from q_4 to q_4 via the loop labeled r_{44}. Hence, when we eliminate the path through q_4, we need to add to the label on the direct arrow between q_0 and q_1 so that it becomes $r_{01} + r_{04}.(r_{44})^*.r_{41}$. Similarly, we can go from q_2 to q_3 directly or via q_4.

We can go from state q_0 to q_3 only via q_4 and if this state is eliminated, the path from q_0 to q_3 will have to be labeled $r_{04}.(r_{44})^*.r_{43}$. Similarly, we can go from q_2 to q_1 only via q_4. If all these four paths avoid going through state q_4, this state can be eliminated altogether, as shown in Fig. 4.6. By applying this procedure repeatedly, any NFA can be reduced to just one or two states.

TABLE 4.2 Eliminating States in an NFA to Convert to Regular Expressions

Before State Elimination	After State Elimination	Remarks
		A concatenation
		A + operation
		A loop

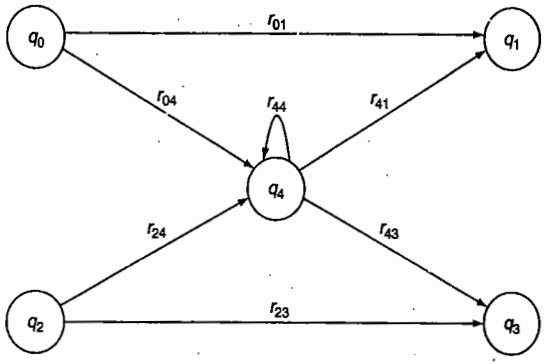


FIGURE 4.5 General configuration for eliminating a state.

It may be noted that in the most general case there will be more than two states "before" and more than two states "after" the state being eliminated. Nevertheless, this method can be applied to eliminate states one by one until the automaton is reduced to just one or two states.

When the start state is also the only final state, we will have just one state at the end. In this case, the RegEx on the label of the loop from this state back to itself is the final RegEx that is equivalent to

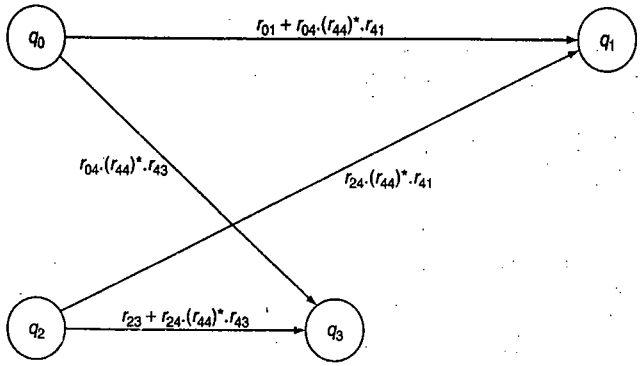


FIGURE 4.6 Regular expressions resulting from state elimination.

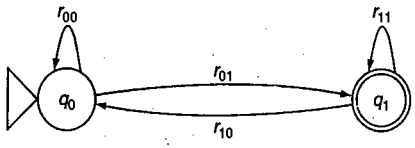


FIGURE 4.7 Final configuration in converting NFA to RegEx.

the given NFA. If the final state is not the same as the start state, we end with a configuration shown in Fig. 4.7, which can be read as follows:

> There is only one way to go from the start state q_0 to the final state q_1, namely, through the arrow labeled r_{01}. However, before embarking on this journey, the automaton can take jolly rides from q_0 to q_0: either through r_{00} or from q_0 to q_1 and back to q_0. And while it is in q_1, it might take a few jolly rides as well: from q_1 to q_1 or from q_1 to q_0 and back to q_1! Similarly, even after reaching the final state q_1, the automaton can go through loops from q_1 to q_1 itself or back to q_0 and then to q_1. These possibilities result in the final RegEx:

$$(r_{00} + r_{01}.(r_{11})^*.r_{10})^*.r_{01}.(r_{11} + r_{10}.(r_{00})^*r_{01})^*$$

The method of eliminating states can be understood analogically in terms of eliminating a traffic junction by building a flyover. Ultimately, when flyovers have been constructed for every junction in the city, there is a direct path from the starting state to the destination with no junctions in between. Of course, various configurations of states and transitions have to be addressed systematically before the entire automaton can be converted to a regular expression.

It is clear from Example 4.13 that regular expressions for such languages are rather unwieldy; the automaton was much easier to construct and comprehend. In the next chapter, we introduce the idea

EXAMPLE 4.13

Let us see how this method works for the earlier example of a DFA (Example 2.10 in Chapter 2) for all strings with an even number of as and an even number of bs. The original DFA from Fig. 2.11 is repeated here in Fig. 4.8(a). Figure 4.8(b) shows the automaton after eliminating state q_1 and Fig. 4.8(c) shows what remains after eliminating state q_2. Successive eliminations result in the final configuration shown in Fig. 4.8(d). From this, we can read the final RegEx as

$$(aa + ab(bb)^*ba + (b + ab(bb)^*a)(a(bb)^*a)^*(b + a(bb)^*ba))^*$$

of a grammar that often provides more compact and elegant representations of a formal language than regular expressions.

Note that at this point we have established the equivalences and correspondences between our first class of machines (i.e., finite automata) and formal languages. Thus far, we know how to convert

1. An NFA to a DFA (by subset construction).
2. A DFA to an NFA (trivial).

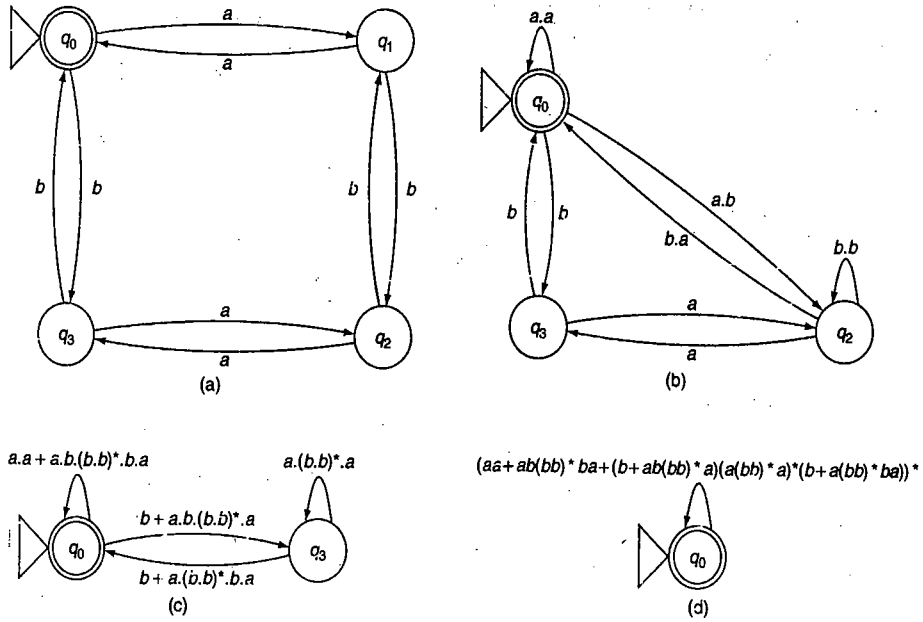


FIGURE 4.8 (a) DFA for even numbers of a and even numbers of b. (b) Generalized transition graph after eliminating state q_1. (c) Generalized transition graph after eliminating states q_1 and q_2. (d) Final generalized transition graph with just state q_0.

3. A DFA to a minimal DFA (by marking distinguishable states).
4. A RegEx to an NFA.
5. A DFA to a RegEx.

In the next chapter, we will see how another formalism for representing formal language known as regular grammars is equivalent to all of the above, thereby completing our coverage of the first triad of machines, languages and grammars.

4.7 Equivalence of Regular Expressions

How can we determine the equivalence of two regular expressions, that is, whether they represent the same formal language or not? For example, it is not obvious that

$$(0+1)^*0$$

is the same as

$$1^*0(0+11^*0)^*$$

both being the set of binary strings representing even numbers. The first RegEx says that strings in the language can begin with anything as long as they end with a 0. The second RegEx says that strings can begin with optional 1 s; the first 0 in the string will be treated as the mandatory 0; this can be followed optionally by more 0 s or by any number of 1 s followed by a 0.

Having learnt how to convert a RegEx to an NFA, we can now look at regular expressions as automata. We immediately recognize that the first RegEx above is the equivalent of an NFA for the language (see Fig. 3.4) and the second to a DFA for the same language (see Fig. 2.3). Since we know that these two automata are equivalent, we can argue that the two regular expressions represent the same language. However, given any two regular expressions, we will not be able to immediately recognize their equivalence through the corresponding automata.

A more reliable method of solving the problem of equivalence of regular expressions is to convert both regular expressions to automata and apply subset construction to convert the resulting NFA to an equivalent DFA and then to compare the two machines.

Comparing two machines also turns out to be not so easy. For instance, do we need to minimize them before comparing? Can our methods for marking distinguishable states and collapsing indistinguishable states to minimize automata (see Chapter 3) establish a one-to-one correspondence between the states and the transitions of the two minimal machines?

There is, fortunately, an easier way:

1. Treat the two machines (DFAs) together as a single machine for a moment with their two start states being reachable from a new start state through λ-transitions.
2. Mark all distinguishable states.
3. If the two start states are marked as distinguishable, then the two automata are different.
4. If the two start states are indistinguishable, then the two machines are equivalent (and hence the two regular expressions from which they came are the same) because indistinguishability requires that the machines reach their final states from the two start states on exactly the same set of strings.

EXAMPLE 4.14

Consider the two regular expressions:

$$0^*10^*1(0+1)^*$$

and

$$((00)^*1+0(00)^*1)\,((00)^*1+0(00)^*1)(0+1)^*$$

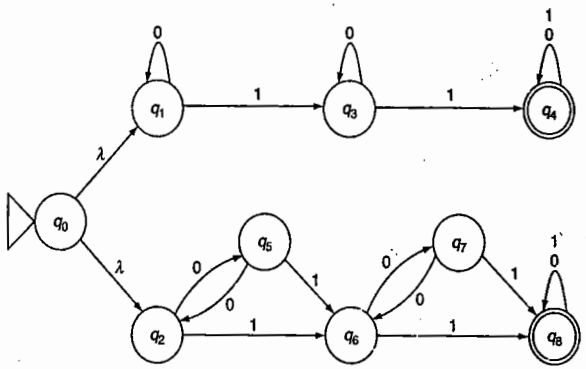


FIGURE 4.9 Combining automata to compare two regular expressions.

Figure 4.9 shows the automaton obtained by combining the two automata which are themselves obtained by converting the given regular expressions. Applying the algorithm for marking distinguishable states (see Sec. 3.5) (ignoring the dummy start state q_0), we find that many pairs of states are distinguishable but not all, as shown in Table 4.3 (see Sec. 3.5 for details of the table and the method for constructing it).

TABLE 4.3 Marking Distinguishable States for Comparing Regular Expressions

q_2	?						
q_3	on 1	on 1					
q_4	fnf	fnf	fnf				
q_5	?	?	on 1	fnf			
q_6	on 1	on 1	?	fnf	on 1		
q_7	on 1	on 1	?	fnf	on 1	?	
q_8	fnf	fnf	fnf	?	fnf	fnf	fnf
States	q_1	q_2	q_3	q_4	q_5	q_6	q_7

In particular, we find that $\{q_1, q_2, q_5\}$ is an indistinguishable set and $\{q_3, q_6, q_7\}$, $\{q_4, q_8\}$ are the other two indistinguishable sets. Thus, the pair of start states of the two automata q_1 and q_2 is

indistinguishable and therefore the two regular expressions must be the same. In fact, they both represent the language of all binary strings containing at least two 1s. Another way to look at the second RegEx is that it is unnecessarily remembering input symbols: 0s that occur in the input are immaterial in this language but still this RegEx and its equivalent automaton are remembering some of them. It may be noted that

$$((00)*1+0(00)*1)$$

means "a 1 preceded optionally by either an odd number or an even number of 0s," which is the same as saying 0*1 or "zero or more 0s followed by a 1."

4.8 Method for Constructing Regular Expressions: *Mantras*

The following *mantras* are useful in trying to construct an accurate RegEx for a given formal language:

1. *What is the simplest string in the language?*
   First construct a RegEx for the simplest string. For example, if the language is binary strings that contain at least two 1s, the simplest string is 11 and the RegEx for this string is also 11.
2. *What are some other strings in the language?*
   Extend the RegEx by including optional elements to cover all the listed strings. A good way to list other strings is to think about what might come before the elements in the RegEx for the simplest string, what might come after them or in their middle. Grow the RegEx accordingly. In the present example, 0s may come before, after or in the middle of the two 1s. Hence the revised RegEx is 0*10*10*.
3. *What are all the other strings in the language?*
   Try to find the expression that includes all the strings in the language while excluding all others. In our example, there is no rule that only two 1s should occur; there can be more 1s and they can appear anywhere. Hence the RegEx becomes

   $$(0+1)*1(0+1)*1(0+1)*$$

   which incidentally is also equivalent to the two regular expressions in Example 4.14.
4. *Is the null string a member of the given language?*
   If yes, every part of the RegEx must be optional, that is, either it must be inside a * closure or it must have λ as an alternative (i.e., in a + construct). For example, a RegEx for the language of binary strings not containing any consecutive 0s is

   $$(1+01)*(0+\lambda)$$

   in which strings can begin with a 0 or 1 and end with a 0 or 1 but every 0 except the last one must be followed by a 1. Remember that in the case of automata, the null string is accepted if the start state is itself a final state (or if there is a λ-transition from the start state to a final state).

5. *With what symbols can strings in the language begin and end?*
   As seen in the above RegEx for binary strings not containing any consecutive 0s, it is often useful to think of the symbols with which strings in the language can begin or end. Remember that in the case of automata, to determine what are the symbols (or prefix) with which accepted strings begin, we need to examine outgoing arcs from the start state that eventually lead to an accepting state. Similarly for ending symbols, we examine incoming arrows (back) into the final state(s). In the case of RegEx, we simply examine the first (or leftmost) and last (or rightmost) parts of the RegEx.
6. *What does an equivalent automaton need to remember?*
   Only those symbols that the machine needs to remember must result in state transitions and therefore mandatory elements of the RegEx. All other symbols belong in optional loops (i.e., * closures) in the RegEx. For example, in the RegEx for at least two 1s, the machine must remember whether it has seen one 1 or two 1s and the RegEx has corresponding concatenations, everything else being optional

   $$(0+1)*.1.(0+1)*.1.(0+1)*$$

7. *What is the complement of the language?*
   Ensure that no string in the complement is accepted by the RegEx. The complement of the language of at least two 1s is the set of binary strings that do not contain any 1 combined with the set of strings that contain a single 1. It is clear that the above RegEx does not match either of those since the two mandatory 1 s are not in any optional part (i.e., not in a + or a * closure).
8. It is useful to remember that (a transition to) each state in the automaton corresponds to a concatenation in the RegEx; a loop corresponds to a * closure and a parallel branch corresponds to a + operator. It is often useful to analyze the given language by first constructing an NFA for it and then arriving at an equivalent RegEx.
9. There is no concept of a reject state in a RegEx. Whatever inputs take an automaton to a reject state should never be present in a RegEx. For example, in the language of binary strings containing at most two 1s, the automaton would go to a reject state on seeing a third 1 in the input. A correct RegEx for the language, namely,

   $$0*(1+\lambda)0*(1+\lambda)0*$$

   does not contain a third 1 at all. Note also that the empty string is present in this language and, as such, every part of this RegEx is optional.
10. Consider all possibilities. We are often tempted to simplify the problem by ignoring one or more possibilities resulting in an incorrect RegEx for the given language.
11. Always consider extreme (or boundary) cases: Is the null string accepted? What is the shortest string possible in this language? What is the longest string possible? And so on.
12. Do not try to combine various choices into a single expression: This often compromises the accuracy by including strings that are not in the language. For example, the RegEx for binary strings, where the first and last symbols are the same, namely,

    $$0(0+1)*0+1(0+1)*1$$

must contain the two "branches" combined with a +. It would be incorrect to combine them and write (0 + 1)(0 +1)*(0 + 1). This RegEx imposes no constraints on the first and last symbols and merely represents all strings of length at least two!

13. In obtaining a RegEx by converting from an automaton, be careful to consider all the loops including those that may be newly generated when a state is eliminated. Leaving out any loop results in an incomplete and therefore incorrect RegEx.

It may be noted that these *mantras* are also useful for determining the language of a given RegEx or automaton (either from a transition diagram or directly from the transition table of the automaton).

EXAMPLE 4.15

Explain in plain English precisely and completely the language of the RegEx:

$$(1+\lambda)(00*1)*0*$$

Let us apply our *mantras* to analyze the nature of strings that match the given RegEx. Strings can begin with a 1 or with a 0; they can end with a 0 or 1. They can be of any length. The null string is also a member of the language of the RegEx. Every occurrence of 1 except as the first symbol in the string must be preceded by at least one 0. From this, we can conclude that this RegEx represents

The set of all binary strings with no consecutive 1 s

4.9 Regular Expressions in Practice

Most programming and scripting languages support regular expressions. In addition to providing efficient implementations of pattern matching using regular expressions, they also provide syntax enhancements such as the following:

1. Since typical alphabets used in programming languages such as the ASCII character set are much larger than the very small alphabets that we have been using in our examples, RegEx syntax provides two other constructs:
   (a) The alphabet is considered an ordered sequence and sets of consecutive symbols can be abbreviated using hyphens and parentheses. For example (*A*-*Z*) stands for $(A+B+C+\cdots+Y+Z)$.
   (b) A special symbol such as . is used to indicate any one symbol from the alphabet.
   (c) Similarly, the closure (i.e., zero or more repetitions of) any symbol in the alphabet, that is, Σ^*, is written as (.)*. For example, if the alphabet has 52 symbols and the language is the set of all strings in which the second symbol is *A*, the RegEx is *A*(.)*.
2. The vertical bar | is often used in place of +. For example, (*A*-*Z*|*a*-*z*) indicates any uppercase or lowercase letter.
3. A symbol such as ., |, or + that has a special meaning in regular expressions can be "escaped" to provide a literal interpretation of the symbol itself by prefixing it with a backslash. For example, \| means the actual | symbol.
4. In some implementations of regular expressions, placing a single ? after a symbol (or sub-expression) makes the symbol optional. Instead of writing $(\lambda + 0)$, we simply write 0?.

5. A special symbol + as a superscript (not to be confused with the use of + above to indicate parallel branches in a RegEx) is introduced for one or more repetitions of a symbol. Thus, while 0* indicates zero or more repetitions of the symbol 0, 00* indicates a 0 followed by zero or more repetitions of 0 (i.e., one or more 0s) and it can also be written as 0^+ in many programming languages.

Regular expressions are used extensively for validating data, especially data entered by users into forms in Web pages and other applications. The following examples illustrate the use of regular expressions in validating email addresses, Web URLs and dates.

EXAMPLE 4.16

Let us assume that email addresses must have an alphanumeric email id that can have numbers, except as the first symbol, and is followed by the @ symbol and then the domain name ending with one of .com, .org, .gov, .mil or .in. A RegEx for validating such email addresses is

(*A*-*Z*|*a*-*z*)(*A*-*Z*|*a*-*z*|0-9)*@((*A*-*Z*|*a*-*z*)(*A*-*Z*|*a*-*z*|0-9)*\.)*

(*A*-*Z*|*a*-*z*)(*A*-*Z*|*a*-*z*|0-9)*\.(com|org|gov|mil|in)

EXAMPLE 4.17

A Web URL address such as http://www.google.com or https://www.india.gov:8080/ can be validated by a RegEx such as

http(s)?://www\.(*A*-*Z*|*a*-*z*)*\.(com|org|gov|mil|in)(:(0-9)*)?/?

EXAMPLE 4.18

Dates valid in the year 2011 in the format dd/mm/yyyy where some months have 30 days, others 31 days and February has 28 days can be validated by a RegEx such as

((0(1-9)|(1-2)(0-9)|30)/(0(4|6|9)|11)|

(0(1-9)|(1-2)(0-9)|3(0-1))/(0(1|3|5|7|8)|1(0|2))|

(0(1-9)|1(0-9)|2(0-8))/02)/2011

EXAMPLE 4.19

Writing regular expressions can often be fun when dealing with real-world data. For example, take a list of names of people (such as the list of students in a class) and apply the following RegEx (in a database language such as SQL, for instance) and see whose names get selected:

(*A*-*Z*)(*A*-*Z*|*a*-*z*)*(*a*|*ee*|*i*|*A*|*EE*|*I*)(\ (*A*-*Z*)(\.|(*A*-*Z*|*a*-*z*)*))*

It is an approximate regular expression for recognizing girls' names; it simply looks for names in which the first name ends with an *a*, *i* or *ee*. Of course, it makes various other assumptions and simplifications:

1. First names begin with an uppercase letter.
2. First names must be at least two letters long.
3. First and other names must be separated by a space (i.e., blank).
4. Other names also must begin with an uppercase letter.
5. Other names may also appear as an initial (with a period).
6. There can be any number of other names.

The ability to construct good regular expressions as well as skills for reading and interpreting regular expressions is very useful in programming, especially in dealing with string processing, pattern matching, natural language processing, search engines, XML and other Web-based applications. With the rapid growth of sophisticated Web-based applications, data is increasingly being represented and processed in flexible data languages such as XML, the eXtensible Markup Language. It is a common task in software development today to build good parsers and interpreters for patterns in data using regular expressions.

In this chapter, we learned that the languages of finite automata, namely, regular languages can be very succinctly described using *regular expressions* that are supported in most modern programming languages. We will see in the next chapter that regular expressions are simpler versions of what we call *grammars*.

EXAMPLE 4.20

Finally, let us see if we can deal with the language $a^n b^n$ using regular expressions. A RegEx for this language cannot be just a^*b^* since this would allow unequal numbers of a s and b s. We are forced to construct an endless expression such as

$$\lambda + ab + aabb + aaabbb + \cdots$$

We cannot write a regular expression of finite length for this language, just like how we couldn't construct an automaton to accept it, because regular expressions can repeat patterns an infinite number of times, but are unable to count and repeat a pattern a specified number of times. We will see in later chapters that this language is not a regular language (Chapter 6) and that it requires more powerful grammar formalisms (Chapter 7) and computing machines (Chapter 8) to handle this language. It is important to note, however, that although automata and regular expressions are both finite, they are perfectly capable of handling infinite languages, that is, sets of infinitely many strings where many strings themselves are infinitely long.

4.10 Key Ideas

1. A formal language is a set of strings made up of symbols from an alphabet.
2. The set of all strings accepted by a computing machine is called the language of the machine.

3. Alphabets and computing machines are always finite. Languages are often infinite and include strings of unlimited length. While software programs must be of finite length, usually they must be able to handle an unlimited variety of input strings. Hence, we are usually interested in infinite languages.
4. A language is governed by some rules that determine which strings belong to the language and which others do not. Such rules are specified concisely using regular expressions or grammars.
5. The complement of a language is the set of all the strings over the alphabet that do not belong to the language. The complement of the language of an automaton is the set of all strings rejected by the automaton.
6. A regular expression is a very compact representation of a set of strings. It is constructed from primitive symbols using the +, . and * operators.
7. Regular expressions are equivalent to finite automata.
8. Regular expressions do not have the equivalent of a reject state. Strings to be rejected cannot be shown explicitly in a regular expression.
9. Regular expressions can be readily converted to a non-deterministic finite automaton.
10. An automaton can be converted to a regular expression by the method of state elimination in which transitions in the automaton are labeled with regular expressions. Such automata are also called generalized transition graphs.
11. Two regular expressions can be compared by first converting them to minimal deterministic finite automata and then determining if the two start states are distinguishable.
12. Although there is no algorithm to design a regular expression, the set of mantras discussed in this chapter help us in designing an accurate RegEx for a given problem.
13. Regular expressions are supported in most programming and scripting languages, usually with an enhanced syntax to make it easy to construct non-trivial expressions.
14. Regular expressions are used extensively in scripting, pattern matching and user data validation in Web-based systems.

4.11 Exercises

A. Construct a regular expression (RegEx) for each of the following (assuming that the shortest acceptable strings are long enough to accommodate all the requirements).

1. Binary numbers that are twice an odd number (i.e., binary representations of $2n$ where n is an odd number).
2. Binary strings containing the sequence 011.
3. Binary strings containing at least one 00 and at least one 11.
4. Binary strings with at least two occurrences of at least two consecutive 1 s, the two occurrences not being adjacent (i.e., 011011 is acceptable but 011111 is not).
5. Strings over $\{a, b, c\}$ in which the fourth symbol from the beginning is a c.
6. Strings over $\{a, b\}$ with an even number of a s.
7. Strings over $\{a, b\}$ with at most three a s.

8. Strings over $\{a, b\}$ of the form $a^{2n}b^{2m}$ where n and m are positive integers.
9. Strings over $\{a, b\}$ whose length is divisible by 2 but not by 3.
10. Binary strings such that if they are of even length, then the number they represent is also even and if the length is odd, then the number is also odd.
11. Strings over $\{a, b, c\}$ with a single c before which an even number of as and any number of bs are present in any order; after the c, there must be an odd number of as and any number of bs in any order.
12. Strings of the form $a^nb^mc^k$, where $n + m$ is odd and k is even.
13. Strings of the form $a^nb^mc^k$, where $n \leq 4$, $m \geq 2$, $k \leq 2$.
14. Strings in the infinite sequence: aba, a^5, aba^7, a^{11}, aba^{13}, a^{17}, ...
15. Strings over $\{a, b, c\}$ in which at least one a appears in the first three symbols and at least one b in the last four symbols.
16. Binary strings in which any run is of length at least 3. A run is a sequence of two or more occurrences of the same symbol. For example, 00011110000 has a run of 0s of length 3 followed by a run of 1s of length 4 and a run of 0s of length 4.
17. All valid dates in the year 2012 expressed in the dd/mm format.
18. All valid telephone numbers: just a 7-digit number (e.g., 26721983), or a 10-digit number (e.g., 9900011111), or a 3-digit area code followed by a space and a 7-digit number (e.g., 080 26721983), or a plus sign followed by a 2-digit country code, a space and either a 10-digit number (e.g., +91 9900011111) or a 3-digit area code, a space and a 7-digit number (e.g., +91 80 26721983). Note that of these, only the three-digit area code can begin with a 0 in its first digit.
19. All valid names of people: a first name and an optional last name and any number of middle names or middle initials (e.g., James Bond or James H. H. E. Bond); or any number of initials followed by a single name (e.g., J. H. H. E. Bond). First name, middle name and last name are all in init-caps (i.e., only the first letter is capitalized); initials are a capital letter followed by a period.
20. All valid monetary amounts: starting with the Rupee symbol (shown here as ₹) with an optional decimal point and a two-digit fractional amount; if it is larger than three digits, then commas separate billions, millions and thousands (e.g., ₹ 101.99 or ₹ 10,000 or ₹ 1,670,439,444.49).
21. All valid GPS coordinates: latitude and longitude expressed in degrees and minutes in the DDD MM.MMMM format with N, S, E, W to indicate North or South of the equator for latitudes and East or West of the Greenwich Meridian for longitudes. Note that latitudes range from 0 to 90 and longitudes from 0 to 180. Leading 0s may be omitted in both, for example, 12 30.0123 N 77 66.5843 E.

B. For the following, describe the language of the RegEx as concisely as possible.

22. $1(0+1)(0+1)(0+1)(0+1)*0$
23. $(b+\lambda)(a\,(a+\lambda)*(b+\lambda))*(a+\lambda)*$
24. $(ab+ba)*$
25. $(aa)*(bb)*(b+\lambda)$
26. $(A-Z)(a-z)*(a+e+i+o+u)$

27. $(a-z)(a-z|0-9)*$
28. $(1-9)(0-9)(0-9)?\backslash(1-9)(0-9)(0-9)?\backslash(1-9)(0-9)?\backslash(1-9)(0-9)?$

C. Simplify the following RegEx's:

29. $(\alpha+\beta)*\beta\alpha(\beta+\alpha)*+(\beta+\alpha)*\beta\beta(\alpha+\beta)*$
30. $00*(0+\lambda)(1+\lambda)(1+\lambda)(1+\lambda)*$
31. $(00+11+01+10)*$
32. $abc+a(c+b)(c+b)+(b+c)a(c+b)+(c+b)(c+b)a+(b+c)(b+c)(b+c)$

D. or the following, construct a finite automaton (deterministic finite automaton or non-deterministic finite automaton) first and then convert it to a RegEx.

33. Strings over $\{a, b\}$ with an odd number of as and an odd number of bs.
34. Binary strings not containing the keyword 011.
35. Binary strings in which every 0 is followed by 11.
36. Binary strings representing positive integers divisible by 3.

E. Convert the given RegEx to an equivalent NFA:

37. $(0+\lambda)(1+\lambda)(1+2)*0(2+1)*$
38. $(0+11+10(1+00)*01)*$
39. $((a+b+c)c)*(a+b+c+\lambda)$
40. $(a+b+c)*c(a+b)c(a+b)(c+\lambda)*$

F. Convert the given deterministic finite automaton (DFA)/non-deterministic finite automaton (NFA) to an equivalent RegEx.

41. DFA shown in Fig. 2.5 for even binary numbers without leadings 0s.
42. DFA shown in Fig. 2.12 for same symbols in all odd positions.
43. NFA shown in Fig. 3.5 to search for a keyword.
44. NFA shown in Fig. 3.9 for certain binary strings.
45. NFA shown in Fig. 3.11 for binary numbers divisible by either 4 or 6.
46. DFA shown in Fig. 3.15 for strings with 0, 1 and 2 in that order.

G. Are the following pairs of RegEx's equivalent (i.e., do they represent the same set of strings)?

47. $(0+\lambda)(11*0)*(1+\lambda)$ and $(1+\lambda)(011*)*(0+\lambda)$
48. $(0*1*)*$ and $(0+1)*$
49. $(1+\lambda)(00*1)*0*$ and $(0+\lambda)(11*0)1*$
50. $(0+1)*(0+\lambda)$ and $(1+\lambda)(1+0)*(0+1+\lambda)$
51. Can a RegEx with only union and concatenation operators (i.e., with no * closure) represent an infinite number of strings? Explain.

H. Debug and correct the following incorrect RegEx's.

52. Binary strings with a 0 in every third position: $110(0+1)^*$.

53. Binary strings with alternating 0s and 1s starting with either a 0 or a 1: $(01+10)^*$.

54. Student letter grades represented as strings made up of the symbols $\{A, B, C, D, F\}$ with no more than two F grades:

$$(A+B+C+D)^*F(A+B+C+D)^*F$$

55. Time in 24 hour hh:mm:ss format:

$$(1+2+0)(0-9):(0-5)(0-9):(0-5)(0-9)$$

56. Radio stations from AM 535.0 kHz to AM 1605.0 kHz and FM 88.0 MHz to FM 108.0 MHz:

$$(A+F)M\ (1+\lambda)(0-9)(0-9)(0-9).(0-9)\ (\mathrm{k}+\mathrm{M})\mathrm{Hz}$$

Chapter Five

Grammars

Learning Objectives

After completing this chapter, you will be able to:

- Learn how grammars can be more compact than regular expressions.
- Learn how grammars are used to parse or derive strings in a language.
- Learn how to construct grammars for regular languages.
- Learn to convert automata to regular grammars.
- Learn to convert regular grammars to automata.

The third set of key concepts presented in this book, namely, grammars, are introduced in this chapter by taking examples from both natural languages (e.g., English) and formal languages. The equivalence of right (or left) linear grammars to regular languages and finite automata is demonstrated and the set of mantras running through the chapters are now adapted for the construction of grammars.

5.1 *The Idea of* a Grammar

Regular expressions (RegEx), although quite powerful and practical in representing regular languages, are not always elegant or succinct. For example, consider the language of all binary numbers divisible by 3. We have seen a finite automaton for this language [Fig. 2.8(b)]. A RegEx for this language (which can be obtained by eliminating states as shown in Chapter 4) is

$$(0+11+10(1+00)^*01)^*$$

Now, let us consider the set of all binary numbers divisible by 5. An automaton for this language was shown in Fig. 2.9. A RegEx for this language, however, does not look very elegant:

$$(101+11(01)^*(1+00)1+(100\ +11(01)^*(1+00)0)(1+0(01)^*(1+00)0)^*0(01)^*$$
$$(1+00)1)(0\ +101+11(01)^*(1+00)1+(100\ +11(01)^*(1+00)0)(1+0(01)^*$$
$$(1+00)0)^*0(01)^*(1+\ 00)1)^*\ +0$$

Even if we try to make it look pretty by indenting parts of the expression, we get

```
(101
+ 11(01)*(1 + 00)1
 + (100 + 11(01)*(1 + 00)0) (1 + 0(01)*(1 + 00)0)* 0 (01)*(1 + 00)1
)
(0 + 101 + 11(01)*(1 + 00)1 +
   (100 + 11(01)*(1 + 00)0) (1 + 0(01)*(1 + 00)0)* 0 (01)*(1 + 00) 1
)*
 + 0
```

This is clearly too unwieldy for human comprehension. We observe that a few sub-expressions or patterns repeat in this RegEx. For example, (01)*(1 + 00) occurs several times. What if we could assign a short form to this expression the same way as we use local variables in programming? We can then reduce the above complex RegEx as follows:

$$(101 + 11U1 + (100 + 11U0)(1 + 0U0)^* 0U1)$$
$$(0 + 101 + 11U1 + (100 + 11U0)(1 + 0U0)^* 0U1)^* + 0$$

where

$$U = (01)^*(1 + 00)$$

Continuing with this process, we can further reduce the RegEx to

$$(W + XY^*Z)(0 + W + XY^*Z)^* + 0$$

where

$$\begin{aligned} W &= 101 + 11U1 \\ X &= 100 + 11U0 \\ Y &= 1 + 0U0 \\ Z &= 0U1 \end{aligned}$$

Using variables such as U, W, X, Y and Z to represent "intermediate" expressions eliminates repetitions to give us a concise representation of the formal language. This is the *idea of* a *grammar*. In a grammar, the above expression is further reduced to generate a set of *rules* for replacing variables and expanding the expression until the entire expression is generated.

While appreciating the power of a regular expression to give us a very compact representation of an elaborate automaton, we also realize its limitation in cases such as the above. By assigning sub-expressions to variables, we could get a much more readable representation. This is exactly what a grammar enables us to do: get very compact, machine-processible representations of fairly complex languages and machines without the need for elaborate transition diagrams or transition tables.

On the same lines, consider the RegEx we constructed in Chapter 4 (Example 4.4) for the set of all binary strings containing exactly one pair of consecutive 0s:

$$(1 + (01))^* 00(1 + (10))^*$$

This can be written more elegantly as a grammar:

$$\begin{aligned} S &\rightarrow 1S \mid 01S \\ S &\rightarrow 0A \\ A &\rightarrow 0B \\ B &\rightarrow 1B \mid 10B \mid \lambda \end{aligned}$$

We will learn the details of the above notation in a minute. For now, we can read the above rules as follows:

The start symbol (or start state) S can begin with optional 1s or optional 01s ($S \rightarrow 1S \mid 01S$). This is the same as an automaton looping back to the same state (i.e., a jolly ride). It goes to symbol (or state) A on consuming a 0 ($S \rightarrow 0A$) and then to symbol (or state) B on consuming the second consecutive 0 ($A \rightarrow 0B$). These are the same as an automaton making a transition to another state. B can consume optional 1s or 10s at the end ($B \rightarrow 1B \mid 10B$). These are jolly rides again. Finally, it ends by disappearing into the null symbol λ, ($B \rightarrow \lambda$). This is the same as marking this state as a final state in the automaton.

In addition to *computational grammars* of the above kind, we are familiar with the traditional idea of a grammar for human languages such as English. Let us begin our study of grammars by analyzing the famous English sentence from the movie *My Fair Lady*:

The rain in Spain stays mainly in the plain.

Depending on the method used in our English grammar classes, we may have learned one of the following ways of analyzing this sentence to determine if it is grammatical:

```
{  {The rain}
   {in Spain}
}
{  {stays mainly}
   {in
        {the plain}
   }
}.
```

or

```
Sentence:
     Subject:
          Noun Phrase:
               Article:                The
               Noun:                   rain
          Prepositional Phrase:
               Preposition:            in
               Noun Phrase:
                    Proper name:       Spain
     Predicate:
          Verb Phrase:
               Verb:                   stays
```

```
        Adverb:              mainly
Prepositional Phrase:
        Preposition:         in
        Noun Phrase:
              Article:       the
              Noun:          plain
```

The simplified grammar of English with which we analyzed this sentence says that

a valid English *sentence* is made up of a *subject* and a *predicate*;
the *subject* is a *noun phrase*;
the *predicate* is a *verb phrase* (and if the *verb* is transitive, it has an *object noun phrase*);
noun phrases are made up of either an *article* and a *noun* or just a *proper name*;
verb phrases are made up of a verb (and an optional adverb);
noun and *verb phrases* can also have an optional *prepositional phrase* that in turn is made up of a *preposition* and a *noun phrase*.

Just as in the above example of a formal language, we see how the pattern of a noun phrase repeats three times in our sample sentence. A complete grammar of English is of course much more complex and has many more rules. These simplified rules of English grammar can be written concisely as follows:

Sentence → Subject Predicate
Subject → Noun_Phrase
Predicate → Verb_Phrase [Noun_Phrase]
Noun_Phrase → [Article Noun | Proper_Name] [Prepositional_Phrase]
Verb_Phrase → Verb [Adverb] [Prepositional_Phrase]
Prepositional_Phrase → Preposition Noun_Phrase

Traditionally, the rules of grammar for a natural language such as English are written in the form of descriptive sentences in the same language (with the significant exception of the ancient work on Sanskrit grammar by Pāṇini who systematically compiled a set of cryptic rules called *sutras*). While traditional grammar books are useful for students of the language, they are not suitable for processing by a computing machine.

The idea of a *computational grammar* was the seminal contribution of Noam Chomsky to the field of computer science. His work on creating formal grammars for languages led to entirely new fields of research and development known as *computational linguistics* and *natural language processing*. What is even more important is Chomsky's contribution to programming languages. His work on grammars and parsing enabled the development of high-level programming languages and compilers for the languages and thereby eventually the entire information technology industry, the software revolution, the World Wide Web, and so on. In Chapter 11, we will study a classification of formal languages proposed by Chomsky into what is called the *Chomsky Hierarchy*.
We define a formal or computational grammar G as consisting of these four elements:

1. *G*.variables, denoted by **V**, is the finite set of variable names used in the grammar; they are sometimes also called *non-terminals* or phrases; in the above examples, *S*, *A*, *B*, *W*, *X*, *Y*, *Z*, Sentence, Subject, Predicate, Noun_Phrase, and so on are all variables.

2. *G*.terminals, also denoted by **T**, is the finite set of terminal symbols; this is the alphabet of the language of the grammar; in the grammar for English, the words in English are the terminal symbols; 0 and 1 are the terminal symbols in the grammar for a binary language.
3. *G*.startSymbol, usually denoted by **S**, is a special variable denoting the start symbol of the grammar; this is also called the sentence variable.
4. *G*.productions, usually denoted by **P**, is the set of grammar rules, also known as *production rules* or simply *productions*; each production $x \rightarrow y$ has a left-hand side x and a right-hand side y; x and y are both sequences of variable or terminal symbols.

G.startSymbol S is always the starting point for applying the grammar to a string. It is called the sentence variable since a grammar was originally meant to define the structure of a sentence in a natural language.

5.2 *The Ideas of* Parsing and Derivation

A grammar is used in two ways: to verify that a given string (or sentence) is grammatical and to derive or generate a new string. The process of verifying grammaticality, known as *parsing*, involves ensuring that each part of the string is constructed according to the production rules of the grammar. A compiler for a programming language, for example, parses the given program to verify that it is grammatical and that it does not have any syntax errors. In the process, it usually also constructs a data structure known as a *parse tree* that represents the structure of the program and enables it to convert the program to executable (or intermediate) code in later stages of compilation. Similarly, a parse tree for an English sentence shows the structure of the sentence in terms of its clauses and phrases.

Generation or *derivation* is the process of *expanding* the start symbol S by repeatedly applying the production rules of the grammar until all variables disappear and only a string of terminal symbols from the alphabet of the grammar remains. The grammar is said to have derived the string. A production rule is applied by replacing the variable that occurs on its left-hand side with the contents of the right-hand side of the rule. As such, the first production to be applied must have S on its left-hand side. A string that is derived from a grammar is guaranteed to be grammatical according to that grammar. Every part of the string is of course derived according to the rules of the grammar.

Parsing and derivation are inverses of each other. A derivation can also be represented as a parse tree where the root of the tree is the variable *S*. Each intermediate node of the parse tree is a variable in the grammar. Each leaf node is a terminal symbol. The string that is derived or parsed is represented as the sequence of leaf nodes in the tree read from left to right. Incidentally, the structure of the English sentence shown in Section 5.1 is also nothing but a parse tree.

EXAMPLE 5.1

Consider the language of binary numbers that are even, which is represented by the RegEx (0 + 1)*0. We can construct a grammar G_1 for this formal language as follows:

(a) G_1.variables = {*S*}
(b) G_1.terminals = {0, 1}
(c) G_1.startSymbol = *S*

(d) G_1.productions = $\{S \rightarrow 0S$
$S \rightarrow 1S$
$S \rightarrow 0\}$

Given a string 1010 that belongs to this language, we can parse it as follows by replacing terminal symbols that match right-hand sides of production rules with the left-hand sides of the productions:

$$1010 \Leftarrow 101S \Leftarrow 10S \Leftarrow 1S \Leftarrow S$$

This parse of the given string is represented by the parse tree shown in Fig. 5.1. The tree shows graphically how production rules have been applied to replace parts of the string being parsed. Each node in the tree is the left-hand side of a production; its immediate children, in left-to-right order, constitute the right-hand side of the same production. For example, the production $S \rightarrow 1S$ generates a sub tree where a node labeled S has two children nodes, the left child labeled 1 and the right child labeled S.

The parse tree in Fig. 5.1 also represents the derivation of the string 1010 from the start symbol S:

$$S \Rightarrow 1S \Rightarrow 10S \Rightarrow 101S \Rightarrow 1010$$

Notice that the sequence of derivation is exactly the reverse of the sequence shown above for parsing. Parsing can be seen as the process of going upward in a parse tree from the leaf nodes to the root; derivation is the reverse process of starting from the root node and generating the children nodes to ultimately derive the string represented by the leaf nodes of the tree. For example, in the above derivation sequence, the symbol S is replaced by $1S$ using the production $S \rightarrow 1S$. This results in the root node being expanded to the leftmost leaf node labeled 1 and the second-level child node labeled S. Next, the S in $1S$ is expanded to $0S$ using the production $S \rightarrow 0S$ so that the overall string becomes $10S$. This is represented in the tree by the second-level S node having a left child labeled 0 and a right child labeled S. Two more applications of appropriate productions result in the final string 1010. The last production $S \rightarrow 0$ when applied eliminates S, thereby generating a parse tree where all the variables are in interior nodes (i.e., non-leaf nodes). This completes the process of derivation.

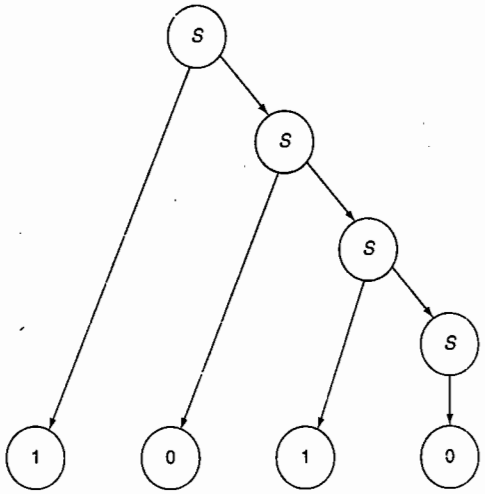


FIGURE 5.1 Parse tree.

Intermediate strings obtained during parsing or derivation are called *sentential forms*. For example, $10S$ is a sentential form. A sentential form contains both variables and terminals. The final sentential form which contains no variables is also called a *sentence*. Derivation can be seen as a process of successively expanding the sentential form until the required sentence is generated. Parsing is the process of successively collapsing the given string by introducing variables until no terminal symbol is left and the sentential form reduces to just the symbol S.

We can observe several properties of a parse tree:

1. It is an ordered tree.
2. The root is S.
3. Leaves are terminal symbols (or the empty symbol λ).
4. Interior nodes are variables.
5. λ, when it occurs as a leaf, cannot have a sibling node.

There are many choices during parsing or derivation. For example, in the first step in a derivation, we can apply any production that has S on its left-hand side. In order to derive a particular sentence or parse a given string, we need to carefully select the production rule to be applied. Because of this choice, the same grammar can derive a set of different strings. The set of all strings that can be derived from the start symbol S of a grammar is called the *language of the grammar*. The language of a grammar can be an infinite set of strings, but given any string of finite length, it takes a finite number of applications of productions to parse or derive the string. The same production may of course be used multiple times in deriving the same string.

EXAMPLE 5.2

Grammars are not unique for a language. Let us construct a different grammar G_2 for the same language of even binary numbers:

1. G_2.variables = $\{S, T\}$
2. G_2.terminals = $\{0, 1\}$
3. G_2.startSymbol = S
4. G_2.productions = $\{ S \rightarrow T0$
$T \rightarrow T0$
$T \rightarrow T1$
$T \rightarrow \lambda \quad \}$

Notice that this grammar requires a second variable. The start symbol, having generated the mandatory 0 at the end of the string, has to change the variable to T before other optional symbols can be generated. The string 1010 can also be derived from this grammar:

$$S \Rightarrow T0 \Rightarrow T10 \Rightarrow T010 \Rightarrow T1010 \Rightarrow \lambda 1010 \text{ (which is the same as 1010)}$$

The parse tree for this derivation, shown in Fig. 5.2, looks quite different from the one in Fig. 5.1.

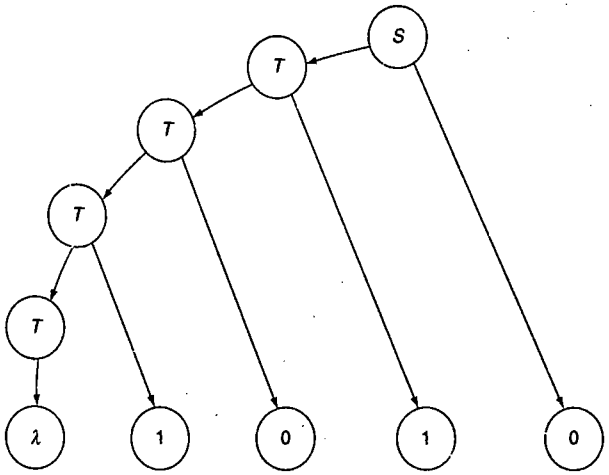


FIGURE 5.2 Parse tree from a different grammar.

5.3 Grammars for Regular Languages

Grammars are very powerful. We don't need the most general types of grammar to represent the kind of formal languages with which we have been dealing using finite automata and regular expressions. For these *regular languages*, a restricted form of grammar known as *regular grammar* is sufficient.

A regular grammar, G_{Reg}, is a grammar in which the production rules $x \rightarrow y$ are constrained in four ways:

1. y should be at least as long as x, that is, the right-hand side cannot be shorter than the left-hand side; sentential forms in a derivation can only expand, they cannot shrink at any point.
2. There can be no terminal symbols on the left-hand side; there can be only one non-terminal symbol on the left-hand side (which makes the grammar *context free* as we will see in Chapter 7).
3. There can be only one non-terminal (i.e., variable) on the right-hand side (which makes the grammar *linear* as we will see in Chapter 7).
4. The single variable on the right-hand side must appear either as the rightmost symbol in every rule, giving us a *right-linear grammar*, or as the leftmost symbol in every production rule, giving us a *left-linear grammar*; the grammar cannot have some rules where the variable is the leftmost symbol on the right-hand side and some others in which the variable is the rightmost symbol on the right-hand side; no rule can have a variable in the middle of the right-hand side, that is, with terminal symbols on either side of a variable.

Right-linear and *left-linear grammars* are called *regular grammars*. The grammar in Example 5.1 was a right-linear grammar and the grammar in Example 5.2 was left linear. The derivation of a string using a

right-linear grammar has a variable at the end of every sentential form; the variable occurs at the beginning of the sentential form for a left-linear grammar. As shown in Fig. 5.1, the parse tree for a right-linear grammar is skewed to the right (it is about to fall off to the left!) and the parse tree for a left-linear grammar, as shown in Fig. 5.2, is skewed to the left (is about to fall off to the right). In both cases, the terminals at the leaves of the parse tree, when read left to right, give us the string being parsed (or being generated from the start symbol S).

5.4 Constructing Regular Grammars

Let us construct several regular grammars to understand how they work.

EXAMPLE 5.3

Let us construct a right-linear grammar for all strings over $\{a, b\}$ that start with two or more as followed by three or more bs (i.e., $a^n b^m$, $n \geq 2$, $m \geq 3$). We begin with the start symbol S. Two mandatory as must be generated at the beginning. An equivalent automaton will be changing states to remember the first a and the second a; an equivalent regular expression will construct concatenations; the grammar changes variables to produce the same effect:

$$S \rightarrow aA$$
$$A \rightarrow aB$$

At this point, we can generate any number of additional as without any need to count or remember how many were generated. Thus, we can add the rule for a loop:

$$B \rightarrow aB$$

We are done with as. Now, similarly, we need to generate three bs:

$$B \rightarrow bC$$
$$C \rightarrow bD$$
$$D \rightarrow bE$$

And then, we can generate any number of bs through a loop back to E:

$$E \rightarrow bE$$

Finally, when the required number of bs is generated, we need to stop and get rid of the variable E:

$$E \rightarrow \lambda$$

It is often not necessary to represent every intermediate expression or variable. For example, in the above grammar, the intermediate variable A that stands for the state of having found one of the two required as need not be shown explicitly. Rather, two terminal symbols can be included together in the same production rule by saying $S \rightarrow aaB$ instead of $S \rightarrow aA$ and $A \rightarrow aB$. You may notice that this is the

same as eliminating states in an automaton to convert it to a generalized transition graph (see Chapter 4). The revised grammar is

S	$\rightarrow$	aaB	Two mandatory as
B	$\rightarrow$	aB	Any number of additional as
B	$\rightarrow$	$bbbE$	Three mandatory bs
E	$\rightarrow$	bE	Any number of additional bs
E	$\rightarrow$	λ	End generation (similar to reaching a final state)

Notice that we have shown only the productions of the grammar since variables, terminals and the start symbol can be seen clearly from the productions themselves. Further, we can shorten the grammar by combining production rules that have the same variable on the left-hand side and separating the right-hand sides of different rules in the combined rule using the vertical bar | as shown below:

$$\begin{aligned} S &\rightarrow aaB \\ B &\rightarrow aB \mid bbbE \\ E &\rightarrow bE \mid \lambda \end{aligned}$$

This is a right-linear grammar for the language; every rule has a variable (if any) as the rightmost element of its right-hand side. Similarly, we can develop a left-linear grammar for this language (see below) thereby demonstrating that grammars are not unique for a given language. However, given a grammar, there is exactly one formal language that it represents.

S	$\rightarrow$	$Sb \mid Bbbb$	Any number of optional bs and then exactly three bs
B	$\rightarrow$	$Ba \mid aa$	Any number of optional as and then exactly two as

The first pair of productions, say, that the start symbol S can append any number of bs at the right end of the string without counting them or without changing states or variables; when it is done doing so, it generates the three required bs and switches to variable B. The productions for variable B are similar to those for S: B can add any number of as at the end before finally generating the two mandatory as.

As seen in this example, *just as the automaton has to change states to remember something about the input string processed thus far, the grammar has to change variables to remember some part of the string that has been already expanded to terminals.*

EXAMPLE 5.4

Our next example is for the language of some number of as followed by some number of bs where the total length of the string is an odd number, that is, for the language $a^n b^m$, where $n+m$ is odd. There are two ways in which the length of such a string can be odd: the number of as is odd or the number of bs is odd (but not both). How can a grammar generate an odd number of symbols? It must generate one symbol and then optionally generate any number of pairs of symbols:

$$\begin{aligned} S &\rightarrow aA \\ A &\rightarrow aaA \end{aligned}$$

Note that a change of variable is necessary here. It would be incorrect to write $S \rightarrow aS \mid aaS$; we would have no control over the number of as generated from such a pair of production rules.
If we generate an odd number of as, we have to ensure that we generate an even number of bs (and vice versa). Hence, a right-linear grammar for the language is

S	$\rightarrow$	$aA \mid C$	Odd number of as or even number of as (two branches)
A	$\rightarrow$	$aaA \mid B$	Continue to maintain odd number of as
B	$\rightarrow$	$bbB \mid \lambda$	Add even number of bs
C	$\rightarrow$	$aaC \mid D$	Even number of as in this branch
D	$\rightarrow$	$bbD \mid E$	First add even number of bs
E	$\rightarrow$	b	Finally an odd b

The following grammar is incorrect for the given language. Why?

S	$\rightarrow$	$A \mid C$	One of the two branches
A	$\rightarrow$	$aaA \mid aB$	Odd number of as
B	$\rightarrow$	$bbB \mid \lambda$	Even number of bs
C	$\rightarrow$	$aaC \mid bbC \mid b$	Even number of as and odd number of bs

In both Examples 5.3 and 5.4, it is useful to observe when a change of variable is needed and how to exit the grammar by expanding the last variable to λ or to one or more terminal symbols.

EXAMPLE 5.5

Consider the language represented by the RegEx 00*(01+0)* where the only non-optional part is the first 0. The following left-linear grammar represents the language:

S	$\rightarrow$	$S0 \mid S01 \mid T$	Generate the optional (01+0)* on the right
T	$\rightarrow$	$T0 \mid 0$	Generate 0* and then finally 0

Notice that an equivalent right-linear grammar needs productions with λ on their right-hand sides:

S	$\rightarrow$	$0T$	Generate the mandatory 0
T	$\rightarrow$	$0T \mid U$	Generate the optional 0s in 0*
U	$\rightarrow$	$01U \mid 0U \mid \lambda$	Generate the optional (01+0)* at the end

Of course, both grammars can be further simplified by realizing that the given language is the same as the one represented by the RegEx 0(01+0)*

EXAMPLE 5.6

We can assign meanings to variables in a grammar the same way as we described the meanings of states in an automaton. To see an example of this, consider all binary strings with at most three 0s:

$$
\begin{aligned}
S &\rightarrow 1S \mid 0T \mid \lambda && S \text{ generates all strings with zero, one, two or three 0s}\\
T &\rightarrow 1T \mid 0U \mid \lambda && T \text{ generates all strings with zero, one or two 0s}\\
U &\rightarrow 1U \mid 0W \mid \lambda && U \text{ generates all strings with zero or one 0}\\
W &\rightarrow 1W \mid \lambda && W \text{ generates all strings with no 0s}
\end{aligned}
$$

Notice that every time a 0 is introduced, the variable changes (from S to T, from T to U and from U to W). This way, there is no room for introducing a fourth 0. Also, this is equivalent to an automaton changing states to remember that it has seen the symbol 0.
A left-linear grammar for this language is quite similar:

$$
\begin{aligned}
S &\rightarrow S1 \mid T0 \mid \lambda\\
T &\rightarrow T1 \mid U0 \mid \lambda\\
U &\rightarrow U1 \mid W0 \mid \lambda\\
W &\rightarrow W1 \mid \lambda
\end{aligned}
$$

5.5 Converting Automata to Regular Grammars

As already noted, regular grammars are equivalent to both finite automata and regular expressions. Converting a finite automaton to a regular grammar is a straightforward process. The start state of the automaton maps to the S variable of the grammar. The alphabet of the automaton is the set of terminal symbols of the grammar. States in the automaton map to variables in the grammar. Every transition in the automaton becomes a production rule in the grammar. A final state is indicated by a production rule that does not have a variable on its right-hand side.

The equivalence of regular grammars to finite automata is easily seen by establishing a bidirectional correspondence between transitions in a deterministic finite automaton (DFA) or a nondeterministic finite automaton (NFA) and productions in a grammar. This is shown in Table 5.1. In addition, if the start state of the automaton is also a final state (or if the null string is accepted through one or more λ transitions from the start state to one of the final states), the production $S \rightarrow \lambda$ must be added to the grammar.

EXAMPLE 5.7

Let us construct a regular grammar for the language of 0s, 1s and 2s in which all 0s should appear before any 1 and all 1s should appear before any 2s. A DFA and an NFA for this language were shown in Figs. 3.15 and 3.16, respectively.

TABLE 5.1 Correspondence between Grammar Productions and Transitions in a Finite Automaton

Transition	Production (right linear)	Remarks
$\delta(q_1, a) = q_2$	$V_1 \rightarrow aV_2$	The production generates the symbol consumed by the transition
$\delta(q_1, a) = q_1$	$V_1 \rightarrow aV_1$	Loop back to the same state or variable
$\delta(q_1, a) = q_f$	$V_1 \rightarrow a$	Final state q_f; no new variable introduced in the grammar
$\delta(q_1, \lambda) = q_2$	$V_1 \rightarrow V_2$	Just a change of variable (or state); such productions are called *unit productions* (see Chapter 7)
$\delta(q_1, \lambda) = q_1$	$V_1 \rightarrow V_1$	There is really no need for such a transition or production!
$\delta(q_1, \lambda) = q_f$	$V_1 \rightarrow \lambda$	Final state q_f; variable V_1 is eliminated; this is a special case and we do not consider this as the right-hand side being shorter than the left-hand side

Let us first convert the NFA in Fig. 3.16 to a right-linear grammar:

$$
\begin{aligned}
S &\rightarrow 0S \mid A\\
A &\rightarrow 1A \mid B\\
B &\rightarrow 2B \mid \lambda
\end{aligned}
$$

The derivation of the string 001222 using this grammar is

$$S \Rightarrow 0S \Rightarrow 00S \Rightarrow 00A \Rightarrow 001A \Rightarrow 001B \Rightarrow 0012B \Rightarrow 00122B \Rightarrow 001222B \Rightarrow 001222\lambda$$

From the DFA in Fig. 3.15, we get

$$
\begin{aligned}
S &\rightarrow 0S \mid 1A \mid 2B \mid \lambda\\
A &\rightarrow 1A \mid 2B \mid \lambda\\
B &\rightarrow 2B \mid \lambda
\end{aligned}
$$

The derivation of the same string 001222 using this grammar is

$$S \Rightarrow 0S \Rightarrow 00S \Rightarrow 001A \Rightarrow 0012B \Rightarrow 00122B \Rightarrow 001222B \Rightarrow 001222\lambda$$

It is clear that the derivation using the grammar obtained from the DFA is shorter (i.e., has fewer applications of productions) for the given string. This is an important consideration in the design of programming languages and their compilers. We will get an introduction to the problem of efficient parsing when we study normal forms of grammars in Chapter 7.

We can also obtain left-linear grammars from the automata. It is useful to think of left-linear grammars as deriving a string from right to left, while a right-linear grammar derives strings from left to right. From the NFA, we get

$$
\begin{aligned}
S &\rightarrow S2 \mid C \mid \lambda\\
C &\rightarrow C1 \mid D \mid \lambda\\
D &\rightarrow D0 \mid \lambda
\end{aligned}
$$

A derivation of the string 001222 using the left-linear grammar is

$$S \Rightarrow S2 \Rightarrow S22 \Rightarrow S222 \Rightarrow C222 \Rightarrow C1222 \Rightarrow D1222 \Rightarrow D01222 \Rightarrow D001222 \Rightarrow \lambda 001222$$

Similarly, from the DFA, we get the left-linear grammar:

$$\begin{aligned} S &\rightarrow S2 \mid C2 \mid D2 \mid \lambda \\ C &\rightarrow C1 \mid D1 \mid \lambda \\ D &\rightarrow D0 \mid \lambda \end{aligned}$$

Finally, the derivation of 001222 from this grammar is

$$S \Rightarrow S2 \Rightarrow S22 \Rightarrow C222 \Rightarrow D1222 \Rightarrow D01222 \Rightarrow D001222 \Rightarrow \lambda 001222$$

Once again, the grammar obtained from the DFA has more productions in it, but it is able to derive the given string in fewer steps.

EXAMPLE 5.8

Even if an automaton is rather complex, deriving a grammar from it is straightforward. For example, consider once again the language of binary numbers divisible by 5. A DFA for this language was shown in Fig. 2.9 (Chapter 2). Remember that we were unable to construct an elegant regular expression for this language (see Sec. 5.1). A right-linear grammar for the set of all binary numbers divisible by 5 is

$$\begin{array}{llllll} S \rightarrow 1A & E \rightarrow 0E & C \rightarrow 0A \\ S \rightarrow 0F & G \rightarrow 1G & A \rightarrow 1C \\ E \rightarrow \lambda & D \rightarrow 1D & E \rightarrow 1A \\ A \rightarrow 0B & F \rightarrow \lambda & C \rightarrow 1B \\ D \rightarrow 0C & F \rightarrow 1G & B \rightarrow 0D \\ G \rightarrow 0G & F \rightarrow 0G & B \rightarrow 1E \end{array}$$

It may be noted that the variable G, which represents the reject state q_7 in the automaton in Fig. 2.9, is really not necessary since once G is introduced, there is no way to get rid of it from the sentential form to derive any string at all.

5.6 Converting Regular Grammars to Automata

By applying the mapping in Table 5.1, we can also convert a regular grammar to a finite automaton.

EXAMPLE 5.9

Consider the language over the binary alphabet where the first two symbols must be the reverse of the last two symbols in the string, that is, the first symbol is the same as the last symbol and the second symbol is the same as the penultimate symbol. A RegEx for this language is

$$0(0(0+1)*0+1(0+1)*\ 1)0+1(0(0+1)*0+1(0+1)*1)1$$

A right-linear grammar for this language is

$$\begin{aligned} S &\rightarrow 0A \mid 1D \\ A &\rightarrow 0B \mid 1C \\ B &\rightarrow 0B \mid 1B \mid 00 \\ C &\rightarrow 0C \mid 1C \mid 10 \\ D &\rightarrow 0E \mid 1F \\ E &\rightarrow 0E \mid 1E \mid 01 \\ F &\rightarrow 0F \mid 1F \mid 11 \end{aligned}$$

An NFA obtained by converting this grammar is shown in Fig. 5.3. We can think of a more elegant grammar for this language. For example,

$$\begin{aligned} S &\rightarrow 0A0 \mid 1A1 \\ A &\rightarrow 0B0 \mid 1B1 \\ B &\rightarrow 0B \mid 1B \mid \lambda \end{aligned}$$

However, this is neither a left-linear grammar nor a right-linear grammar because of productions in which the variable is in the middle of two terminal symbols on the right-hand side of the production. It is not a regular grammar, although its language is still a regular language.

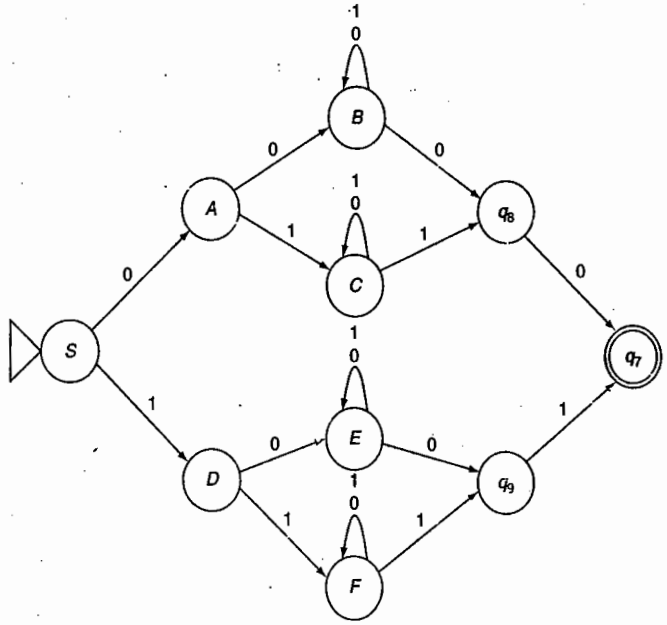


FIGURE 5.3 NFA obtained by converting a grammar.

EXAMPLE 5.10

The above example showed how a non-regular grammar can be constructed for a regular language. Linear grammars such as the one above that are not right linear or left linear are too powerful for regular languages. Such grammars can also be constructed for non-regular languages. For example, we can easily write a linear grammar for our familiar non-regular language $a^n b^n$ as follows:

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

As we know, this is not a regular grammar and the language is not regular; we have failed earlier to construct a DFA, NFA or RegEx for this language. We will see in Chapter 6 precisely why this language is not regular.

5.7 Constructing Regular Grammars: *Mantras*

Given a description of a formal language, it is sometimes easier to construct an NFA first, sometimes a RegEx and at other times a grammar. Since we already know how to convert from one formalism to the others, we are free to choose whichever one is more intuitive to us for a given problem.

Let us write down a set of *mantras* for constructing regular grammars:

1. *What is the simplest string in the language?* Write a production for the start variable S that generates just the simplest string. For example, for the language of all binary strings containing at least two 1 s, the simplest string is 11 and the production rule is $S \rightarrow 11$.
2. *What are some other strings in the language?* List a few other strings in the language and try to include them in the productions. For example, other than the two 1 s, there can be 0 s anywhere in the string. We can include them by adding the rule $S \rightarrow 0S$.
3. *What are all the other strings in the language?* Think of what other symbols can be present in an acceptable string in the language. Enhance the grammar rules to cover all of them. In general, first write the grammar for a simpler language and then extend it to take care of modifications as needed. It is useful to pay particular attention to extreme cases and boundary conditions for inclusion of a string in the language. In our example, more than two 1 s can be present in a valid string. To accommodate additional 1 s, we add the rule $S \rightarrow 1S$.
4. *Is the null string a member of the language?* If yes, it must be possible for the start symbol S to derive the null string λ. This can happen in two ways: either there is a direct rule $S \rightarrow \lambda$ or S derives another variable that is "nullable," that is, $S \rightarrow A$ and $A \rightarrow \lambda$.
5. *With what symbols do strings in the language begin and end?* In a grammar, beginning symbols are derived by the start symbol S; this is especially easy to see in a right-linear grammar as these symbols appear as leftmost symbols on the right-hand side of an S production. Ending symbols in a right-linear grammar appear in "final" productions that contain only terminal symbols on their right-hand sides. In a left-linear grammar, on the other hand, ending symbols are easier to spot; they appear as rightmost symbols in an S production. Beginning symbols appear as terminals only right-hand sides of productions in a left-linear grammar.
6. *What does an equivalent automaton need to remember?* Whenever an automaton needs to remember the occurrence of an input symbol, it changes to a new state. We saw in Chapter 4 that an equivalent RegEx moves on to the next part to account for this (i.e., it introduces a concatenation or . which is often not written explicitly). On the same lines, a grammar changes variables to "remember" some part of the string that has already been expanded to terminals. In our example, there is no need to remember 0 s since they don't matter as far as the grammaticality of an input string is concerned. On the other hand, the machine does need to remember the two 1 s that are essential for a string to be a member of the language. In our earlier production $S \rightarrow 11$, we did not care to remember the occurrence of the two 1 s. As a result, our current set of productions does not allow 0 s in between the two 1 s. We need to change this production to $S \rightarrow 1A$ and $A \rightarrow 1B$.
7. *What are all the productions for each variable in the grammar?* Consider each variable as it is introduced and add all the productions that are necessary or possible with that variable as the left-hand side. In our example, we now need to add $A \rightarrow 0A \mid 1A$ and $B \rightarrow 0B \mid 1B$.
8. When can derivation end? Consider the idea of a final state in an equivalent automaton; whenever the automaton reaches one of the final states, the corresponding regular grammar has a production with no variable on the right-hand side. The right-hand side can have either one or more terminals or just the empty symbol λ. In our example, S or A do not correspond to a final state; only B does since only in B are we sure that two 1 s have already been encountered. Therefore, we add $B \rightarrow \lambda$.
9. Grammars are not constrained to behave deterministically. Like regular expressions, grammars also behave more like non-deterministic finite automata (NFAs) than deterministic finite automata (DFAs). As such, it is often useful to construct or think of an NFA first in writing a regular grammar.
10. In constructing a regular grammar for a language, write a regular expression first; it is often easier to translate a RegEx to a grammar than to conceive the production rules directly.
11. Set-theoretic thinking is often useful. For example, think about what is not in this language or whether it is the complement of a simpler language or the union or intersection or difference of two different simpler languages (see Chapter 6).
12. In particular, try to split the language into two or more sublanguages that are easier to handle; then, combine the grammars mutually exclusively by adding a production of the form $S \rightarrow S_1 \mid S_2 \mid \ldots$ where the variables S_1, S_2, and so on derive the individual sublanguages.
13. Just as in the case of regular expressions, there is no need to introduce the equivalent of a reject state in a grammar. A reject state in an automaton corresponds to a variable that cannot be eliminated from a derivation. Once such a variable is introduced into a sentential form, no complete string of terminal symbols can be generated from it.
14. Finally, just as in the case of automata and regular expressions, do not try to merge branches that were created for mutually exclusive sublanguages. In the case of grammars, usually separate variables denote different branches (e.g., variables A and D in Example 5.9). The productions for such variables should be kept separate throughout to avoid errors of deriving non-member strings.

To complete the examples used to illustrate some of the *mantras*, a right-linear grammar for the language of binary strings containing at least two 1 s, that is, $(0+1)^*1(0+1)^*1(0+1)^*$ is

$$\begin{aligned} S &\rightarrow 0S \mid 1S \mid 1A \\ A &\rightarrow 0A \mid 1A \mid 1B \\ B &\rightarrow 0B \mid 1B \mid \lambda \end{aligned}$$

As stated in Chapter 1, the subject matter of this book concerns sets of machines or automata, formal languages and grammars. At this point, we have completed the study of our first set: finite automata, regular languages and regular expressions as well as regular grammars. These will be extended in the next chapters to context-free, context-sensitive and other higher classes of triads of automata, languages and grammars.

5.8 Key Ideas

1. A grammar is a set of production rules.
2. Grammars are sometimes more compact than regular expressions. They allow patterns to be assigned to variables or non-terminals which can be reused in other production rules.
3. The set of all strings that obey all the rules of a grammar is called the language of the grammar. Each of those strings can be parsed or derived using the grammar.
4. Parsing and derivation are inverse processes. A string in the language of a grammar can be derived from the start symbol S by expanding it repeatedly using some or all of the production rules in the grammar.
5. A string belonging to the language of a grammar can also be parsed by replacing terminal symbols with non-terminals according to the production rules until only the symbol S remains.
6. Strings used during parsing or derivation that contain both terminals and non-terminals are called sentential forms.
7. A regular grammar is a right-linear or left-linear grammar.
8. The language of a regular grammar is a regular language.
9. Regular grammars are equivalent to regular expressions and finite automata.
10. A regular grammar can be converted to an equivalent finite automaton and vice versa.
11. An automaton changes states to remember something. A regular expression introduces a concatenation. A regular grammar changes variables to remember something.
12. Although there is no algorithm to design a grammar, the set of mantras discussed in this chapter help us in designing an accurate regular grammar for a given problem.

5.9 Exercises

A. Construct right-linear or left-linear grammars for the following regular languages:

1. Binary strings in which every 0 is followed by 11. Construct a parse tree for the string 0111011
2. Binary strings in which each string has alternating 0s and 1s and is of even length.

   Also, convert the grammar to an equivalent deterministic finite automaton (DFA). What is the minimum number of states in the equivalent DFA?
3. Binary strings that begin with 11 and end with 11 or begin with 00 and end with 00.
4. Binary strings starting with 000 or ending with 111 (or both). Show the derivation of 00010111
5. Binary strings in which the sum of the last three digits is even (e.g., 00101011 but not 00101001).
6. Strings over $\{a, b, c\}$ that contain at least one a and at least one b.

7. Strings over $\{a, b\}$ that contain at least three as or at least two bs.
8. Strings over $\{a, b\}$ in which some number of as is followed by some number of bs with the total length of the string being divisible by 3. Show the parse tree for *aabbbb*.
9. Strings over $\{a, b\}$ containing at least two as and ending with an even number of bs.
10. Strings over the alphabet $\{a, b\}$ of the form $(ab)^n$, for example, *ababab*.
11. Strings of the form *abwba* over the alphabet $\{a, b\}$ where w is any string over the same alphabet. Show the parse tree for *abbaba*.
12. The set of all binary strings that are palindromes of length 4. The alphabet is $\{a, b, c\}$.
13. Binary strings with no consecutive 0s. Show the derivation of the string $w_1 = 1011101$ and also why $w_2 = 1001$ cannot be derived.
14. Binary strings in which the first part of each string contains at least four 1s and the second part contains at least three 0s.
15. Binary strings with at least two occurrences of at least two consecutive 1s, the two occurrences not being adjacent (i.e., 011011 is acceptable but 011111 is not).
16. Strings over $\{a, b\}$ of the form $aa^*(ab + a)^*$.
17. Strings over $\{a, b\}$ where the last two symbols in each string are a reversal of the first two symbols (i.e., last symbol = first symbol and penultimate symbol = second symbol).
18. Student grades in an examination are represented with the letters $\{A, B, C, D, F\}$. A string such as *ABFCAD* indicates the grades obtained by a student in six different subjects. The grammar must generate only those strings that have at most three Ds and no Fs.

B. For the following, first construct non-deterministic finite automata or deterministic finite automaton and then convert it to a regular grammar.

19. Binary strings with an odd number of 0s and an even number of 1s.
20. Strings over the binary alphabet that do not contain the substring 010.
21. Binary strings in which every pair of adjacent 0s appears before any pair of adjacent 1s.
22. The set of all binary strings having a substring 100 before a substring 111 (both must be present). Show the derivation for the string $w = 00100110111$.
23. A *run* in a string is a sub-string made up of a single symbol. For example, *aabbbcd* has a run of as of length 2 and a run of bs of length 3. The grammar must derive all strings over the alphabet $\{a, b, c, d\}$ with at least one run of as of length 2 and a run of cs of length 3.
24. Strings over $\{a, b\}$ such that the length of the string is a multiple of 3 but not a multiple of 4.
25. Strings of the form ab^3wb^4 where the alphabet is $\{a, b\}$ and w is any string over the same alphabet.

C. Describe the language of the following grammar as concisely as possible.

26. $S \rightarrow aS \mid aaS \mid \lambda$
27. $S \rightarrow aA \mid \lambda, A \rightarrow bS$
28. $S \rightarrow aA \mid \lambda, A \rightarrow aA \mid B, B \rightarrow bB \mid \lambda$

29. $S \to 0S \mid 1S \mid A, A \to 0B, B \to 0C, C \to 0C \mid 1C \mid \lambda$
30. $S \to 01S \mid 10S \mid A, A \to 01A \mid 10A \mid \lambda$

D. Convert the given grammar to an equivalent non-deterministic finite automata.

31. The grammar in Exercise 26.
32. The grammar in Exercise 28.
33. The grammar in Exercise 29.
34. The grammar in Exercise 30.

E. The following grammar is not regular. Convert it to an equivalent regular grammar. What is the language of the grammar?

35. $S \to 0S \mid 1A, A \to A1 \mid \lambda$
36. $S \to abS \mid baS \mid Sab \mid Sba \mid \lambda$

F. The following regular grammars are incorrect. Debug and correct them.

37. Binary numbers divisible by 4:

$$S \to 0S \mid 1S \mid 00S \mid \lambda$$

38. Strings with any numbers of 0, 1, and 2 in that order:

$$S \to 0S \mid 0A, A \to 1A \mid 2B, B \to 2B \mid \lambda$$

39. Strings with at least two consecutive as before an occurrence of two consecutive bs:

$$S \to bS \mid aaA, A \to aA \mid bA \mid bb \mid \lambda$$

40. Binary strings of even length:

$$S \to 0S \mid 1S \mid 0A \mid 1A, A \to 0S \mid 1S \mid \lambda$$

Chapter Six

Nature of Regular Languages

Learning Objectives

After completing this chapter, you will be able to:

- Learn to combine regular languages using their closure properties to obtain more complex regular languages.
- Learn to answer questions such as emptiness and finiteness of regular languages.
- Learn why some languages are not regular.
- Learn why regular languages are called regular.
- Learn to apply the Pumping Lemma in an adversarial game to show that some languages are not regular.

In this chapter we complete our study of the first class of formal languages, grammars and machines. Here we bring out some of the properties of regular languages. Readers learn why they are called regular and how some languages are not regular. The Pumping Lemma used to demonstrate this is stated first in plain English and then introduced as a formal mathematical statement. Mathematical concepts of universal and existential quantifiers are brought in to show why a proof using the Pumping Lemma takes the form of an adversarial game.

6.1 Closure Properties

Regular languages are the most well behaved formal languages. We will see later in the book (e.g., in Chapter 9) that some of the properties of regular languages that we explore in this chapter do not hold good for higher classes of formal languages. These properties are not merely of theoretical interest; they determine the practicality of using such languages whether in the design of programming languages or in data representation and processing.

Having studied computing machines, expressions and grammars for regular languages in the previous four chapters, we are now ready to summarize by defining a *regular language* as a formal language for which we can construct

1. a deterministic finite automaton (DFA), or
2. a non-deterministic finite automaton (NFA), or

3. a regular expression (RegEx), or
4. a right-linear grammar or a left-linear grammar (regular grammar)

to accept, match, parse or generate all the strings in the language and no other string.

If we need to show that a particular set of strings is a regular language, all we need to do is to show that one of the above can be constructed for the set. We use this below to show that the results of various operations on regular languages are also regular languages.

Regular languages remain regular when set-theoretic operations are performed on them. We call such behaviors *closure properties* since any number of such operations keep the result still within the class of regular languages. The closure properties discussed in the following sub-sections apply to regular languages.

6.1.1 Union of Regular Languages

The set union of two regular languages is a regular language.

The result of the union of two regular languages L_1 and L_2 is a regular language because if $RegEx_1$ and $RegEx_2$ are regular expressions for L_1 and L_2, then $(RegEx_1 + RegEx_2)$ is a valid regular expression for $L_1 \cup L_2$. Similarly, if there is a finite automaton for each language, the two automata can be combined into an NFA (by creating a new start state that has λ-transitions to the two original start states) which has all the final states of both automata (see Fig. 6.1). Such an NFA would accept any string that belongs to either language, thereby making its language a regular language that is the union of the two languages. It must be noted that the input string must be supplied to both automata so that they can both independently process the same input.

For example, we know that the set of all binary numbers divisible by 4 is a regular language and that the set of all binary numbers divisible by 6 is also a regular language. Their union – binary numbers divisible by 4 or 6 – is also a regular language, as shown in Chapter 3 (see Example 3.7 and Fig. 3.11).

6.1.2 Concatenation of Regular Languages

The concatenation of two regular languages is a regular language.

The result of the concatenation of two regular languages L_1 and L_2 is also a regular language because if $RegEx_1$ and $RegEx_2$ are regular expressions for L_1 and L_2, then $RegEx_1.RegEx_2$ is a valid

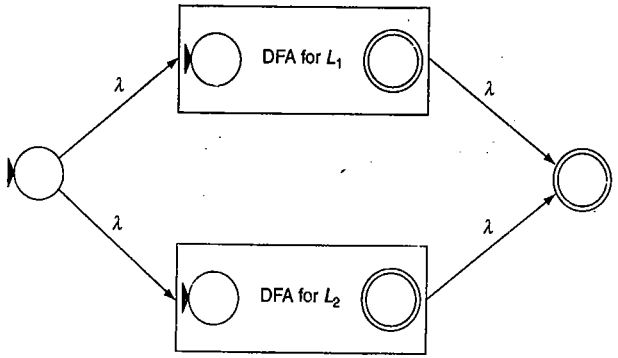


FIGURE 6.1 NFA for the union of two regular languages.

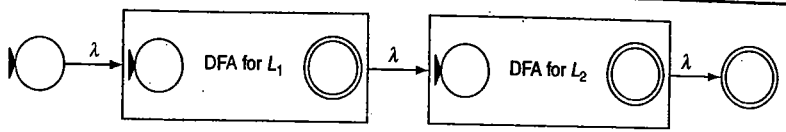


FIGURE 6.2 NFA for the concatenation of two regular languages.

regular expression for the concatenation of L_1 and L_2. Similarly, if there is a finite automaton for each language, the two automata can be combined into an NFA [by creating a chain where the final state(s) of the first automaton has a λ-transition to the start state of the second automaton, as shown in Fig. 6.2]. Such an NFA would accept any string that is a concatenation of a string from the first language and a string from the second language, thereby making the concatenation of the two languages also a regular language.

For example, we know that the set of all strings over $\{a, b\}$ that begin with aa is a regular language: $aa(a + b)^*$. Similarly, strings over the same alphabet that end with bb are also a regular language: $(a + b)^*bb$. The concatenation of the two languages is also a regular language since it is represented by the regular expression: $aa(a + b)^*.(a + b)^*bb$ or just $aa(a + b)^*bb$.

6.1.3 * Closure of Regular Languages

*The * closure of a regular language is a regular language.*

The result of the * closure of a regular language is also a regular language because if $RegEx_1$ is a regular expression for the language, then $(RegEx_1)^*$ is a valid regular expression for the closure. The closure of a language is the set of strings obtained by repeatedly concatenating any number of strings from the language. Similarly, if there is a finite automaton for the language, it can be converted to an NFA (by creating loops with λ-transitions around it as shown in Fig. 6.3). Such an NFA would accept any string that is obtained by repeatedly concatenating strings from the given language, thereby making the closure of the regular language also a regular language.

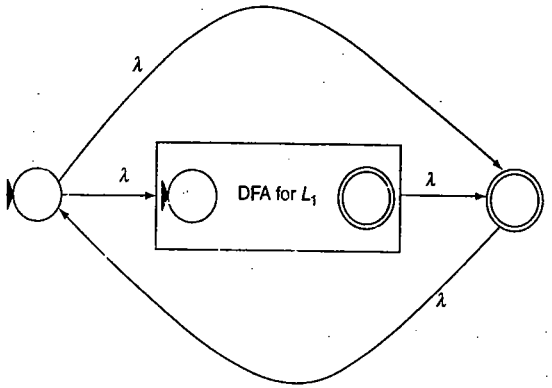


FIGURE 6.3 NFA for the * closure of a regular language.

For example, we know the set of all strings of length three over the alphabet $\{a, b\}$ is a regular language: $(a + b).(a + b).(a + b)$. The set of all strings of length divisible by 3 (as seen in Example 4.7 in Chapter 4) is also a regular language because it is nothing but

$$((a + b).(a + b).(a + b))^*$$

6.1.4 Complement of Regular Languages

The complement of a regular language is a regular language.

The complement of a language is the set of all strings that do not belong to the language. In other words, it is the set of all strings in Σ^* that are not in the given language. Consider an automaton for the given language. Its final states accept all the strings in the language. Whenever an input string does not belong to the language, the automaton halts in a non-final state which might be a reject state or any other non-final state. Let us construct a new automaton which is identical to the original automaton except that every final state is made a non-final state and every non-final state is made a final state. The new automaton rejects every string from the original language and accepts every string that was not in the original language. We have constructed an automaton for the complement of the language which is therefore a regular language.

For example, the set of all even binary numbers (with or without leading zeros) is a regular language: $(0+1)^*0$. The complement of this language, the set of all odd binary numbers, is also a regular language and, in fact, it is represented by the regular expression $(0 + 1)^*1$. Notice how regular expressions do not make it obvious how to obtain the complement of a language. There is no easy way to complement a RegEx.

6.1.5 Reversal of Regular Languages

The reversal of a regular language is a regular language.

The reversal of a language is the set of the reversals of all the strings in the original language. That is, each string is reversed so that its original first symbol becomes the last symbol, the second symbol the penultimate and so on. It is easy to construct an automaton for the reversal, given the automaton for the original language. All we need to do is reverse the direction of every arrow, that is, every transition. The start state becomes the final state and every final state becomes a start state. If there was more than one final state in the original automaton, a new start state can be introduced with λ-transitions leading from it to the original final states. This automaton accepts the reversals of all the strings in the original language, thus making the reversal also a regular language. Another way to show this is by reversing a regular expression at each concatenation. For example, if the original language is

$$(a + b)^*ab(a + b)^*c(ab)^*$$

the reversal of the language is

$$(ba)^*c(a + b)^*ba(a + b)^*$$

6.1.6 Intersection of Regular Languages

The intersection of two regular languages is a regular language.

The intersection has only those strings that are present in both languages. We can combine the automata for the two languages by placing them in parallel in the same way as we did in the case of union (see Fig. 6.1), except, in this case, a string should be accepted only when both automata accept the string by reaching their respective final states. Such a combined automaton accepts the intersection of the two

languages, thereby making it a regular language. An easier way to see this is through the set-theoretic relation between intersection and complement: the intersection of two sets (or languages) is the complement of the union of the complements of the two sets (and we already know that regular languages are closed under the operations of union and complementation):

$$L_1 \cap L_2 = (L_1.\text{Complement} \cup L_2.\text{Complement}).\text{Complement}$$

For example, let L_1 be the regular language of all strings over $\{a, b, c\}$ containing at least one a: $(a + b + c)^*a(a + b + c)^*$. Let L_2 be the regular language of all strings containing at least one b: $(a + b + c)^*b(a + b + c)^*$. Now L_1.Complement is just $(b + c)^*$ and L_2. Complement is just $(a + c)^*$. Their union is $(b + c)^* + (a + c)^*$, strings containing no as or strings containing no bs. The complement of this language is the set of strings that contain both as and bs, or, at least one a and at least one b in whichever order they appear. This is also a regular language (as seen in Example 4.8 in Chapter 4) because it is nothing but

$$(a + b + c)^*a(a + b + c)^*b(a + b + c)^* + (a + b + c)^*b(a + b + c)^*a(a + b + c)^*$$

6.1.7 Difference of Regular Languages

The set difference of two regular languages is a regular language.

The difference of two sets L_1 and L_2 is nothing but the set of strings that are present in L_1 but not present in L_2. This is also the same as the intersection of L_1 with the complement of L_2, that is

$$L_1 - L_2 = L_1 \cap L_2.\text{Complement}.$$

Since we already know that regular languages are closed under both complement and intersection operations, it follows that the difference is also a regular language.

For example, Σ^*, being the closure of the alphabet, is a regular language. The difference between this and any other regular language L, $\Sigma^* - L$, is nothing but the complement of the language L.Complement which we already know is a regular language.

In addition, we can replace the symbols in a regular language with new symbols. The resulting language will still be a regular language. In fact, entire strings can be substituted for each symbol and the language will remain regular. Such a substitution is called a *homomorphism* and regular languages are said to be closed under homomorphism. For example, if we take the language of even binary numbers and replace every occurrence of 0 with "no " and every occurrence of 1 with "yes " then the set of resulting strings (such as "yes no no yes yes no ") is a regular language. This can be seen easily by constructing a regular grammar for the new language:

$$S \rightarrow \text{yes } S \mid \text{no } S \mid \text{no}$$

EXAMPLE 6.1

Having shown that both concatenation and reversing preserve regularity, we propose to show that, given a regular language L, $L.L^R$ – the language L concatenated with its reverse L^R – is a regular language. For example, if L is $(a + b + c)^*$, then what is $L.L^R$? Is it the language of all *even palindromes* over the alphabet? That is, the set of all strings which read the same left-to-right as they do right-to-left? We will see later in this chapter (Example 6.3) that the language of even palindromes (so-called because all strings in the language are of even length) is not a regular language. What is wrong here? Are our conclusions about closure properties of regular languages incorrect?

Not at all! The fact is, the language $L.L^R$ is not the language of even palindromes. It is just a set of strings! This is because in this language, we can take any string from the original language and concatenate it with the reverse of the same string or any other string to generate a valid string in the concatenated language. In the case of even palindromes, every string has in its second half the reverse of the same string as the first half. For example, the strings *abc* and *bbb* belong to L; the strings *abc.cba* and *bbb.bbb* belong to both $L.L^R$ and the language of even palindromes. However, the strings *abc.bbb* and *bbb.cba* also belong to $L.L^R$ but not to even palindromes.

Intuitively, palindromes cannot be regular because the machine will have to remember the first half in order to match it to the second half of the input string which finite automata are unable to do for arbitrarily long palindromes. There is no contradiction here. $L.L^R$ is a superset of even palindromes. There is no need for the machine to remember the first part of the string in this larger language. Often, larger languages are simpler to handle than their subsets.

6.2 Answering Questions About Regular Languages

Next, we address the following elementary questions about regular languages and find simple methods for answering all of them (which will not be possible in the case of other classes of languages, as we will see in Chapter 9):

1. *Is a given regular language empty?* That is, given a representation of the language such as a finite automaton, a RegEx or a regular grammar, is there any string that is accepted by them or is the language a null set?
   Method: The language is empty if there is no path from the start state of its automaton to any of the final states. It is also empty if the start symbol of the grammar S is non-generating, that is, the production rules of the grammar are such that it can never generate a complete string of terminal symbols. Sentential forms in such a grammar can never get rid of all the non-terminals to derive any string (see Chapter 7 for further details about such problems in grammars). Note that in the case of a RegEx, an empty language has no regular expression at all. If the RegEx is λ, we consider it as a language containing just one string, namely, the empty string λ (as such, the language itself is not an empty set; rather it is a singleton set containing the empty string λ). The language is not empty if there is a path from the start state of the automaton to any of its final states or if the start symbol of the grammar is generating or if there is a valid RegEx for the language.
2. *Is a given regular language finite or infinite?*
   Method: A regular language is finite if its automaton (or transition graph) is acyclic. It is infinite if there is a cycle among the states in its automaton that can be traversed in going from the start state to one of the final states of the automaton. It may be noted that even if the language is infinite, the automaton is still finite, that is, it has a finite number of states. Similarly, the language is infinite only if its RegEx has a * closure. For example, $ab + ab^*c$ represents an infinite language while $ab(b + c)a$ is a finite language. A regular grammar represents an infinite language only if at least one loop occurs among its variables, that is, either there is a useful production rule of the form $A \rightarrow aA$ where the same variable occurs on either side or there is a chain of productions that derive the same variable again. For example, the grammar

$$A \rightarrow aB \mid a$$
$$B \rightarrow bC \mid b$$
$$C \rightarrow aA \mid c$$

generates an infinite language.

3. *Does a given string w belong to a regular language L?* This is called the *membership question.* Given that we have been building finite automata to do exactly this, it might seem like a trivial question to ask. As with some of the other questions above, this will turn out to be a rather important and tricky question for larger classes of languages (see Chapter 11).
   Method: Yes, if an automaton for the language accepts the string. No otherwise. Similarly, if a regular grammar for the language derives the string or a RegEx for the language matches the string, then it belongs to the language, else it does not.
4. *Are two regular languages the same?*
   Method: If the two languages are specified in the form of regular expressions or grammars, we first convert them to DFAs (via NFAs if necessary). We then compare the two DFAs using the method outlined in Chapter 4 (Section 4.7) for establishing the equivalence of two regular languages. Essentially, we try to determine if the start states of the two automata are distinguishable from each other. If they are, the languages are different; otherwise they are the same language.

6.3 Why are Some Languages Not Regular?

In answering the question of finiteness of a regular language, we may observe that:

1. Its machine must still be finite even if the language is infinite;
2. the machine for any infinite language must have a loop in it, that is, any software program (that is necessarily finite) must have at least one loop if it must be able to process an infinite language – an infinite variety of inputs; and
3. all finite languages are regular (because we can easily construct an NFA for a finite language that has a separate "branch" or path from the start to a final state for accepting each of the finite number of strings in the language.

This observation about the necessity of loops for processing infinite regular languages is formalized in a famous result known as the *Pumping Lemma for regular languages.* We begin its study by first asking why regular languages are called regular. What is *regular* about them? They are called regular because they have a pattern which repeats in the strings that belong to the language. These patterns make the finite automata for regular languages compact, that is, finite. Without such a pattern, the machine would need an infinite number of states to handle the infinite language. We are really interested only in *finite* machines that handle *infinite* languages because our objective is to build compact programs (or algorithms) that can process an infinite variety of inputs, since only finite things can ever be built, whether in software or hardware. Let us understand exactly how these repeating patterns work in regular languages.

Languages which are not regular do not have a simple repeating pattern. Either they have no pattern at all or they have more than one pattern, with the multiple patterns being correlated with each other. All of

our mechanisms for dealing with regular languages – finite automata, RegEx and regular grammars – are unable to deal with such languages. We see exactly why this is so in the following section.

6.4 The Pigeonhole Principle and the Pumping Lemma

To understand the limitation of a finite number of states being able to handle only regular languages with their simple repeating patterns, we first look at the *Pigeonhole Principle.* This says that only n things at most can fit into n slots or holes or bins. If there are n pigeonholes in a building, each big enough for just one pigeon to sit in, then at most n pigeons can sit in the pigeonholes. If the $(n+1)$th pigeon comes flying in, it will not be able to find a hole to sit in. There is no way that more than n pigeons can be accommodated in n pigeonholes.

We use the pigeonhole principle to show that certain languages are not regular. We also use the idea to outline a proof for the *Pumping Lemma.* As already noted, repeating patterns mean loops in the program or cycles in the automaton. A finite automaton with no cycle cannot handle an infinite language. How can we formally state all of the above ideas? The *Pumping Lemma for regular languages* which is rather complex when expressed formally and mathematically, can be stated in plain English as follows:

> *Regular languages are regular because every non-trivial string in every infinite regular language has a repeating pattern that is not empty and is not too far from the beginning of the string; this pattern can be repeated any number of times to generate infinitely many strings in the language.*

The repeating pattern is handled by the cycle or loop in the finite automaton for the language. How do we say things like "repeating pattern" and "not too far from the start" formally? The *Pumping Lemma for regular languages* says:

> *Every infinite regular language L has a constant m, specific to that language, such that all strings w, $|w| \geq m$ belonging to L can be split into $w = xyz$ where $|xy| \leq m$ and $|y| \geq 1$ and for all $i = 0, 1, 2, \ldots$ the strings xy^iz belong to L.*

The constant m is the minimum size of strings for which the lemma applies. Strings shorter than that need not have any repeating pattern. Why? Because, there can only be a finite number of such short strings and an automaton can handle them with a finite number of states.

Any longer string w with its length $|w| \geq m$, according to the pumping lemma can be split into three parts – x, y and z – and this partitioning of the string must follow two rules:

1. The middle part y which is the repeating pattern cannot be empty or of length 0, that is,
$$|y| \geq 1$$
2. The repeating pattern y cannot be too far from the beginning of the string, that is, the length of the first two parts xy cannot be greater than the constant m, that is,
$$|xy| \leq m$$

The non-repeating pattern x can be handled by a finite number of states since its length is limited. If a repeating pattern exists but it can be infinitely far away from the start of the string, it does not help our cause: the corresponding automaton would need an infinite number of states to handle the part of the string before it encounters the repeating pattern. If xy could be of arbitrary length, then it would take an unlimited number of states to process x and then y before any pattern can be repeated (i.e., pumped), thereby making the automaton infinite and the language not regular.

Once we have the repeating pattern that is not too far from the beginning of the string, the Pumping Lemma says that we can *pump* the repeating pattern y (which is not null), that is, we can repeat it zero times ("pump it down") or any number of times ("pump it up"). We write this by saying, for all $i = 0, 1, 2, \ldots$, the strings xy^iz, where the repeating pattern y is repeated i times, belong to the regular language L.

6.5 Using Pumping Lemma: An Adversarial Game

We do not use the Pumping Lemma to show that a given language is regular; we have easier ways of doing that: we can construct a DFA, NFA, RegEx, or right-linear or left-linear grammar for the language to prove that it is a regular language. Rather, we use the Pumping Lemma to show that a given language is *not* regular.

We use a proof by contradiction to show that a given language is not regular. We begin by assuming that the language is regular. Therefore, the Pumping Lemma must be applicable to it. We apply it, but we cleverly choose a particular string w belonging to the language to set up a contradiction. The contradiction is shown by pumping y (down to $i = 0$ or up to $i = 2, 3, \ldots$) to create an imbalance in the string, that is, to disturb the repeating pattern in the string. We arrive at an xz (for $i = 0$) or an $xyyz$ (for $i = 2$) or an $xyyyz$ (for $i = 3$) which no longer belongs to the language. This is a contradiction. Either the Pumping Lemma is false or our assumption is false. The rest is mere argumentation for the sake of completeness to show that our assumption must have been wrong – that the given language must not have been regular.

Before we apply such a proof by contradiction, we need to be sure that the Pumping Lemma is true for all regular languages. Consider an infinite regular language L and a DFA that accepts the language (i.e., the automaton accepts every string in the language and no other string). The automaton has a finite number of states. Let this number be m. Consider now a string w belonging to L that is at least as long as m, $|w| \geq m$. There must be such a string since the language is infinite; if there is no string longer than a finite number such as m, the language must be finite and is, therefore, anyway regular.

Consider the state transitions that occur as the automaton accepts the string w. In each transition, beginning from the start state of the automaton, there are two possibilities:

1. The automaton goes to a state that was not visited before or
2. the automaton goes to a previously visited state.

Since there are only m states and more than m symbols in the input string w, the second case must occur at least once. How do we know this? From the pigeonhole principle! The m states are like m pigeonholes; the incoming symbols from the input are the pigeons. There are more pigeons than pigeonholes. At least one input symbol cannot find a previously unvisited state; it must try to squeeze into the same hole where another pigeon is already sitting. The situation is shown in Fig. 6.4.

Since at least one state is revisited in the chain of transitions that accepts the input string w, it follows that there is a loop in the path of transitions. It may be as trivial as a loop from a state back to the same state or a much longer loop going through all the m states. In any case, a loop is a jolly ride and may be traversed any number of times. If we skip the loop and go directly towards the final state, we are pumping down the string w (i.e., $i = 0$). If we ride the loop more than once, we are pumping it up (i.e., $i > 1$). All of these strings for any value of i, that is, any number of repetitions of the pattern y, must belong to the

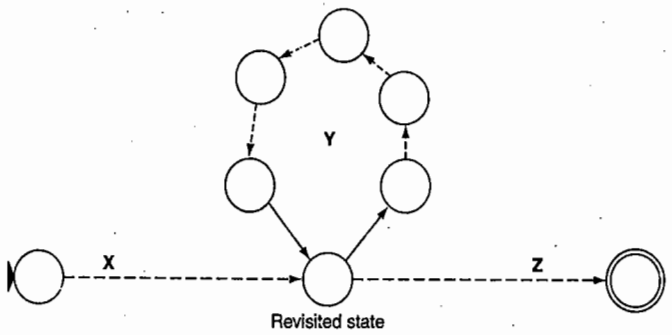


FIGURE 6.4 Mandatory loop for pumping of patterns in regular languages.

language since they are all accepted by the automaton. They all traverse the same path from the start state to a final state of the automaton, the only difference being in the number of times they go through the loop of "intermediate" states. Thus, the Pumping Lemma is true for all the long strings in every infinite regular language.

In the partitioning of the string w into xyz, x is in fact the label on the path from the start state to the revisited state; y is the label on the path along the loop from the revisited state back to itself; and z is the label on the path from the revisited state to the final state (see Fig. 6.4). Of course, there may be more than one revisited state. If so, there are more opportunities to take jolly rides along any combination of multiple loops, thereby generating even more strings all of which belong to the regular language. There is one important point to be noted though for later use: When there are multiple loops, the number of times the first loop is traversed is independent of the number of times any other loop may be traversed in generating a string, and so on for all the other loops. If there are indeed dependencies or correlations, such as for example saying that there are two loops and both loops must be traversed an equal number of times, the language is no longer regular (as we will see in Chapter 9).

Let us first apply the Pumping Lemma in a proof by contradiction to show that a given language is not regular and then outline a general method that is applicable to any non-regular language.

EXAMPLE 6.2

We have seen previously that we cannot construct a finite automaton, regular expression or a regular grammar for a language such as $a^n b^n$: a certain number of as followed by an equal number of bs. This formal language is significant from the point of view of the practical problem of matching nested parentheses in programming languages. We replace the symbol a by the opening parenthesis "(" and the symbol b by the closing parenthesis ")" to get strings of the kind ((()())) which is the same as a^4b^4. This language is known as the language of *simple nesting*. We will now use the Pumping Lemma to prove that the language of simple nesting is not regular.

Since this is a proof by contradiction, we assume that the opposite of what we want to prove (i.e., the "*contra*"-"*diction*") is true. In this case, we assume that the language *is* regular. If it is regular, the Pumping Lemma must be applicable to it. Accordingly, if we consider any string w that is sufficiently

long, which means, $|w| \geq m$ for some unknown constant m, it must be possible to split the string into three parts $x.y.z$ while satisfying the remaining requirements of the Pumping Lemma.

Our objective is to arrive at a contradiction, that is, to find some value for i such that when the repeating pattern y in the string w is pumped that many times, the resulting string no longer belongs to the given language. The clever part of this particular proof by contradiction is in choosing the right w. In this example we choose $w = a^m b^m$ (i.e., $n = m$), where m is the Pumping Lemma constant for the given language (whose value we do not know). The string we have chosen is twice as long as the minimum length for which the Pumping Lemma is applicable. In fact, it is crucial that its first m symbols are all just as.

What can the Pumping Lemma tell us about the repeating pattern y in this language apart from the fact that it cannot be empty?

We know that it must start and end before m symbols from the beginning of the string w, that is, $|xy| \leq m$. This means, for the string we have chosen, xy must all be made up of only as! Thus, we are sure that y contains at least one a and that it cannot contain any bs.

What happens when we pump y up or down?

Consider, for example, $i = 0$: the resulting string xz will have at least one less a (which was lost with the pumping down of y) than b. It is like having at least one extra unmatched closing parenthesis than opening parentheses; the string does not belong to the given language at all. Similarly, when we pump up, say for $i = 2$, we get $xyyz$ which has at least one more a (which was added with the pumping up of y) than b; it is like having at least one extra unmatched opening parenthesis than closing parentheses; the string once again does not belong to the given language.

Either the Pumping Lemma is false or our assumption was incorrect. Since we have already proven the truth of the Pumping Lemma for all regular languages, our assumption that the given language was regular does not hold. Therefore, the language of simple nesting is not a regular language.

We will see in the next chapter how we can construct a more powerful kind of grammar known as *context-free grammar* for this language.

If we had chosen a string that was inconvenient for us, we would not have been able to set up a contradiction. For example, if we had chosen $w = a^k b^k$, where $k = m/2$ [if m was even, or, for the sake of completeness, $k = 1 + (m - 1)/2$ if m was odd], then upon partitioning w into xyz, the pattern xy could cross over between the as and the bs. In such a case, y could contain both as and bs and pumping it up or down may still keep the numbers of as and bs equal. There would be no contradiction. This, however, does not mean that the language is regular; rather, it means that we failed to prove that it is not regular.

6.5.1 Adversarial Game

The key to a successful application of the Pumping Lemma lies in the choice of the string w. Here are some tips for choosing the right string:

1. Since we don't know what m is, and since w must be at least as long as m, the string we choose must somehow contain m in its expression.

2. Our objective is to gain total control of what goes into the parts x and y; to do this, we need to ensure that the first part is at least as long as m and we need to put whatever symbol or pattern is convenient to us in the first part.
3. We need to control the boundary between the first part that is in our control and the second part that is not; in particular, we do not want to allow the opponent (see Table 6.1) to choose a symmetrical y (e.g., ab) that can cross the boundaries between the two parts to ensure symmetry or correlation and thereby prevent any contradiction from occurring.
4. We need to establish a contradiction by creating an imbalance between the first part that is in our control and the second part (which should not change when we pump y).

Why do we have the freedom to make the above choices but not to choose a convenient value of m? It is because of the difference between *universal* and *existential quantifications.* A universally quantified statement is true *for all* (or *every*) values of the quantified variable; an existentially quantified statement is true *for at least one* (or some or *there exists* one) value of the variable. The Pumping Lemma alternates between universal and existential quantification and says:

For all infinite regular languages,
There exists a constant m that is specific to that language, such that
For all strings w longer than m in the language,
There exists a partition of w into xyz, such that
y is not empty and xy is not longer than m and
For all values of i,
y can be pumped up or down i times to generate an infinite number of strings all of which belong to the regular language.

Whenever there is a universal quantification, that is, when a statement is to be true for all values, we are free to choose any value we please. The proof is valid irrespective of which value we choose since the statement is true for every single possible value. Whenever there is an existential quantification, on the other hand, we do not know for which particular value the particular statement is going to be true. As such, we do not have the freedom to choose some convenient value for that variable. If we violate this and choose a value to our liking the proof is no longer valid.

In the case of the Pumping Lemma:

1. The lemma is true for all regular languages and, therefore, we can assume that a particular language is regular and apply the Pumping Lemma to it.
2. The lemma is true for only some value of the constant m for that language and thus the value of m remains unknown to us; we cannot arbitrarily choose $m = 100$ for example.
3. The lemma is true for all strings w that are longer than m and therefore we are free to choose whichever string is convenient to us, as long as it belongs to the language.
4. The lemma is true for some partitioning of w into xyz; we don't know which one it is and we cannot partition the string to our convenience; for example, we cannot say let x be null.

TABLE 6.1 The Adversarial Game of the Pumping Lemma

Move #	Player	Move	Quantification	Intention
1	We	Choose language	Universal	We choose whichever language we want
2	Opponent	Choose m	Existential	Chooses value of m most difficult for us
3	We	Choose w	Universal	We choose the most convenient string
4	Opponent	Choose xyz	Existential	Chooses the most difficult partition for us
5	We	Choose i	Universal	We choose a convenient value of i

5. The lemma is true for all values of i and, therefore, it is sufficient for us to show that one particular value of i (say 0 or 2) results in a string that does not belong to the language, thereby establishing the contradiction.

This entire process for applying the Pumping Lemma can be seen as an *adversarial game* where two opponents try to beat each other by taking turns in playing a game such as chess. In every move we make, we try to maximize our chances of winning the game; in every move that the opponent makes, he/she tries to minimize our chances of winning the game. Table 6.1 summarizes the alternating moves in applying the pumping lemma.

As noted earlier, the most crucial part of the game is choosing a good string w that results in an immediate *checkmate* for the opponent. No matter what move the opponent makes, we should be certain of a win from there on. Let us apply the method to a few more interesting examples.

EXAMPLE 6.3

Consider again the language of all *even palindromes* over an alphabet such as $\{a, b\}$. This is the set of all strings which read the same from left-to-right as they do right-to-left. In other words, their second halves are the reverses of the corresponding first halves and thus the language is often represented as ww^R. Let us assume that even palindromes are regular. We choose the string $w = a^m bba^m$ to satisfy the requirements of the Pumping Lemma (where we do not know what m is). Neither do we know exactly how the string w will be partitioned into xyz by our opponent but since the Pumping Lemma guarantees that the first two parts xy cannot be longer than m, we know that the repeating pattern y contains only as in the string w. Therefore, when we pump it up to $i = 2$, we get a string with at least one extra a, which is no longer an even palindrome. This is a contradiction and therefore *even palindromes* are not a regular language.

EXAMPLE 6.4

Let us consider the language ww, that is, the set of all strings where the second half is a repetition of the first half. To show that it is not regular, as usual we assume that it is regular. We choose the string $w = a^m ba^m b$. The rest of the proof is very similar to the one in the previous example for even palindromes. This is an interesting language and we will encounter it again in the next chapter and in later chapters as well.

EXAMPLE 6.5

Let us show that the formal language of *addition is not a regular language.* The language of addition of positive integers is the set of all strings which have three numbers in them and the third number is the sum of the first two numbers. Let us consider the simplest possible number system for representing the language of addition, namely, the *unary number system.* There is only one symbol {1} in the alphabet of this system. Any number is represented by a sequence of as many 1s. For example, the number 3 is 111 and the number 7 is 1111111. If we use the unary system as is, we run into trouble in separating the first number from the second and the second from the third in strings that belong to our language of addition. To overcome this, we use three different symbols $\{a, b, c\}$ one for each of the three numbers. With this notation, the language of addition is the set of all strings of the form $a^pb^qc^r$, that is, p number of as, followed q number of bs, followed by r number of cs, where $r = p + q$. If in any string $r \neq p + q$, then it does not belong to the language of addition.

We assume that the language of addition is regular and choose $w = a^mb^mc^{2m}$. When we pump this up to $i = 2$, we see that the resulting string no longer belongs to the language because the new value of the first exponent p will be more than m by at least one, and the sum of the first two number will be greater than $2m$. The way we chose the string w, the opponent could not partition it so that there could have been a corresponding increase in the sum (i.e., the third exponent r of the string) when the string was pumped. We will see in the next chapter that addition in fact belongs to the larger class of context-free languages. Of course, as shown in Section 3.6, a finite-state transducer such as a Moore or Mealy machine which corresponds to regular languages can perform serial addition. This is possible only if corresponding bits of the two numbers are presented together in the input so that the transducer can immediately output the resulting bit in the sum, remembering only the carry bit. The above language of addition requires counting of input symbols to perform the addition which is not possible in a finite automaton or regular language.

Before we conclude this chapter, let us see what happens if we try to use the Pumping Lemma to show that the binary language where the total number of 0s and 1s is even is not a regular language. Let us assume that this is regular and choose as our w the string 0^m1^m. It clearly belongs to the language since it has an even number of symbols. In this case, it is quite possible that the opponent partitions it such that $y = 00$. Now, no matter which value of i we choose, no matter how much we pump it up or down, the resulting string still belongs to the language since its total length remains even. Even if we try any other string w, we get the same result for this language. We are unable to establish a contradiction to prove that the language is not regular. This is because, in fact, the language is regular. It is just

$$((0 + 1).(0 + 1))^*$$

Thus, the Pumping Lemma does not help us if we try to show that a language that is actually regular is not regular.

6.6 Key Ideas

1. Regular languages are well-behaved under various operations. They can easily be combined using union, concatenation, * closure, complementation, reversal, intersection and set difference operations.
2. A regular language is empty if there is no path in any automaton for the language from its start state to any accepting state. It is also empty if in any equivalent regular grammar, the start symbol S is non-generating.
3. A regular language is finite if there is no loop in its finite automaton along any path from the start state to any accepting state. A corresponding regular expression has no * closure and the corresponding regular grammar has no recursive production rule or cycle among any of its non-terminals. Any software program that is able to handle an unlimited variety of inputs must have at least one loop (or an equivalent recursive function) in it.
4. Every finite language is regular.
5. All infinite regular languages have a repeating pattern that is not too far from the beginning of its strings.
6. The repeating pattern can be repeated any number of times to generate an infinite set of strings each of which must belong to the regular language. This is called the Pumping Lemma.
7. The Pumping Lemma is used only to show that a given language is not regular in a proof by contradiction.
8. The Pumping Lemma is used in an adversarial game where the opponent chooses an unknown value for the constant m and an unknown partition of the string w into xyz. We can choose the string w and the value of i to our convenience.
9. Application of the Pumping Lemma works like an adversarial game because of alternating universal and existential quantifiers. We are free to choose any value when a variable is universally quantified but must ensure that our arguments work for any unknown value when under an existential quantifier.
10. The key to a successful application of the Pumping Lemma is to choose a string that includes the constant m and takes full control over the crucial substring of length m.

6.7 Exercises

1. If the alphabet is $\{a\}$, what is the complement of a^*? Of a^+?
2. We know that the concatenation of two regular languages is a regular language. Consider the language $L = 0^n1^n$ over $\{0, 1\}$; L is not regular. Now consider, the language $L_1 = \{0^n\} = 0^*$ and $L_2 = \{1^n\} = 1^*$. L_1 and L_2 are obviously regular. Explain why although L_1 and L_2 are regular, L which could be seen as a concatenation of L_1 and L_2 is not regular.
3. What happens if we apply the Pumping Lemma to show that a formal language such as $((a + b)\,(a + b)\,(a + b))^*$ that is actually regular is not regular? Explain.

A. Using closure properties of regular languages, construct a finite automaton (non-deterministic finite automaton or deterministic finite automaton) for:

4. Binary strings which when interpreted as positive integers are not divisible by 4.
5. Strings over $\{a, b\}$ that do not contain two consecutive as.

B. Using closure properties of regular languages, show that the following languages are regular.

6. Binary strings that do not contain the substring 101.
7. Binary strings made up of two parts; the first part begins with a 1 and ends with a 0; the second part begins and ends with a 1.
8. Binary strings which represent positive integers that are not divisible by 5.
9. Binary strings which represent positive integers that are divisible by 3 but not by 5.
10. Binary strings which when reversed represent positive integers that are divisible by 3.
11. Strings over $\{a, b, c\}$ that either have the same symbol in all odd positions or have the same symbol in all even positions.
12. Strings over $\{a, b, c\}$ whose length is neither an even number nor divisible by 3 or 5.
13. The set of encrypted strings in a rather simple encryption scheme that uses the set of prime numbers less than 1000. A given message is encrypted by repeated concatenations of prime numbers from the set.

C. Show that the following languages are not regular.

14. Binary strings of even length having the same number of 1s in its two halves.
15. The language 0^n1^m, $n <= m$.
16. Binary strings containing more 1s than 0s.
17. Odd palindromes over $\{a, b\}$, that is, wcw^R, where c is a special symbol marking the midpoint of the string.
18. Binary strings whose length is a perfect square.
19. The language of subtraction (in the unary number system). For convenience, use the sublanguage in which the result of subtraction is always zero or positive.
20. The language of unary multiplication.

Chapter Seven

Context-Free Languages and Grammars

Learning Objectives

After completing this chapter, you will be able to:

- Learn what context-free behavior is.
- Learn how context-free grammars can derive languages that are not regular.
- Learn to construct linear context-free grammars.
- Learn to construct non-linear context-free grammars.
- Learn to construct leftmost and rightmost derivation trees for strings.
- Learn the relationship between parsing and ambiguity in grammars.
- Learn to eliminate ambiguity in grammars.
- Learn to convert a context-free grammar to Chomsky Normal Form.
- Learn to convert a context-free grammar to Greibach Normal Form.
- Learn to parse a string using the CYK algorithm.

This chapter begins the study of the second class of formal languages, namely, context-free languages. It tells readers why they are called context-free and explains the nature of context-free grammars (which will be completed in Chapter 9). The rest of the chapter shows how to construct context-free grammars methodically and how they are used in the parsing and compilation of high-level computer programs. The need for normal forms is established and the methods for converting grammars to the Chomsky and Greibach Normal Forms are demonstrated. An algorithm for parsing strings in context-free languages is presented.

7.1 *The Idea of* Context-Free Behavior

In the last chapter, we saw how the Pumping Lemma could be used to show that some formal languages are not regular. If they are not regular languages, what are they? We introduce here the next bigger class of formal languages known as *context-free languages* (CFLs) and the corresponding class of grammars known as *context-free grammars* (CFGs). In the next chapter, we will study computing machines known as *pushdown automata* that are powerful enough to handle context-free languages.

In *linguistics*, the study of languages, *context* refers to the relevant constraints of the communicative situation, both verbal and social, that influence language use, language variation and discourse. In the

study of formal languages, context usually means the surrounding text, that is, what appears before and what appears after the current (set of) symbol(s) in the input string. Context-free means being able to take an action irrespective of the context, that is, irrespective of the surroundings. Life would be really simple if it was context-free: if our past actions had no consequence on the present and our present actions had no consequence on the future! But it is usually not so. Much the same way, in the case of formal and programming languages, we would like their behaviors and hence their grammars to be mostly context-free to simplify their parsing, derivation or other kinds of processing. However, there are a few elements in the syntax of any modern programming language that are not context-free.

As seen in Chapter 5, the action that we consider in a grammar is parsing a symbol or a sequence of symbols in the input or, in the opposite direction, expanding a variable in the present sentential form by replacing it with the right-hand side of a production for that variable to derive the next (usually longer) sentential form. A grammar is called context-free if the application of a production rule for parsing or derivation is independent of the context in the sentential form where it is applied. That is, if a variable A occurs in the sentential form, it can be expanded using any production whose left-hand side is A irrespective of what terminal or variable is to the left or right of A in the present sentential form (and, indirectly, in the input string).

For example, consider a sentential form: *aaaSbbb* and the production $S \rightarrow aSb$. If we can apply the production rule to the sentential form to derive the next sentential form *aaaaSbbbb*, the action is said to be context-free. The S was replaced by aSb without looking at what was to the left or right of S in the sentential form. If all production rules are of this nature, then we call the grammar a context-free grammar, often abbreviated as CFG.

How can a grammar not be context-free?

If the expansion of a variable using one of its productions is meant to be dependent on the context, then the context must be somehow encoded in the production itself. This can be done only by including more than just a variable on the left-hand side of a production. Let us say that a production rule has on its left-hand side one or more terminals or other variables to the left or right of a variable S. The production can be applied and S can be expanded using it only if the entire pattern on the left-hand side of the production appears as is in the present sentential form. Such an action is not context-free.

For example, consider the same sentential form *aaaSbbb* and the productions $aS \rightarrow aaSb$ and $Sa \rightarrow aSbb$. The first production is applicable but the second is not because in the context of the given sentential form, the variable S is preceded by a but not followed by a. Thus, from the current sentential form, we can derive only the sentential form *aaaaSbbbb* by replacing aS by $aaSb$ as specified by the first production. A grammar that is not context-free is called *context-sensitive*. In Chapter 9, we consider examples of languages that are not context-free and see why they are not context-free. We study context-sensitive grammars later in Chapter 11 (Sec. 11.9).

7.2 Nature of Context-Free Grammars

In our study of regular languages, we learned in Chapter 5 that regular grammars were either left-linear or right-linear. Let us now look at the nature of the next bigger class of grammars which we call context-free grammars. As seen in Chapter 5, a grammar G has four components to it with additional

constraints on the structure of the production rules. We now relax some of the constraints so that a *CFG* is as follows:

1. *G.variables*: denoted by V, is the finite set of variables or non-terminals usually represented by uppercase letters such as S, A, B, W, X, Y, Z, etc.
2. *G.terminals*: denoted by T, is the finite set of terminal symbols usually denoted by lower-case letters and digits. It is also known as the alphabet of the grammar. In addition, the special null symbol λ is also a terminal.
3. *G.startSymbol*: usually denoted by S, is a special variable denoting the start symbol of the grammar.
4. *G.productions*: usually denoted by P, is the set of grammar rules, also known as *production rules* or simply *productions*. In each production $x \rightarrow y$:
   - x, the left-hand side, is a single variable from the set V.
   - y, the right-hand side, is any sequence of terminals or variables. y can consist of only variables, only terminals or a combination of both.

In the case of regular grammars, as you remember, the right-hand side y had to contain a single variable that always had to be the rightmost symbol in every production in a right-linear grammar (and the leftmost symbol in every production in the case of a left-linear grammar). These restrictions are no longer applied to CFGs where productions can have terminals and variables anywhere and in any order on their right-hand sides.

Why are these grammars called context-free?

The productions in a CFG are such that their left-hand sides are free of any context, that is, the left-hand side x is just a single variable. There is no context for the application of any production rule during parsing or derivation. The variable on the left-hand side of a production can be expanded using the production without regard to what is to the left or right of it in the current sentential form.

All regular grammars are context-free but not all CFGs are regular. In other words, regular grammars are a proper subset of the set of CFGs. *Regular grammars, in addition to being context-free, must also be linear and either right-linear or left-linear.* In other words, it is possible to write a CFG that is not a regular grammar for a regular language but it is impossible to write a regular grammar for a language that is not regular.

EXAMPLE 7.1

Consider the regular language where every string has an even number of as followed by an even number of bs, that is, $(aa)^*(bb)^*$. A CFG for this language is:

$$S \rightarrow aaS \mid Sbb \mid aaSbb \mid \lambda$$

As you can see, this is a linear grammar but is neither left-linear nor right-linear. The language is still regular since we can construct a right-linear grammar for it (or even the regular expression above):

$$S \rightarrow aaS \mid T \mid \lambda$$

$$T \rightarrow bbT \mid \lambda$$

What is the language of a CFG that is not regular?

You will remember from Chapter 5 that

1. The language of a grammar is the set of all strings over its alphabet which can be derived or generated from the start symbol S of the grammar.

   OR

2. The same set of grammatically correct strings which can be parsed using the grammar to construct a parse tree with S as its root and the input string along the leaves of the parse tree (which is also called the *yield* of the parse tree).

The language of a CFG that is not a regular language is called a *context-free language* (CFL for short). CFLs are a superset of the set of regular languages, that is, *there are CFLs that are not regular* as we saw in Chapter 6. One cannot construct a finite automaton or regular expression for a CFL that is not regular.

In the case of regular languages, we first studied the class of computing machines, namely, finite automata, and then the corresponding regular expressions and grammars, and finally the properties of the languages themselves. In the case of CFLs, our study is in a slightly different order: we have first begun our study of CFGs; we will get adept at constructing them before looking at the equivalent class of computing machines (in Chapter 8); finally, we study interesting properties of CFLs (in Chapter 9).

7.3 Constructing CFGs: Linear Grammars

Let us construct linear CFGs for several languages to derive a set of *mantras* that constitute a reliable method for constructing a CFG for any CFL. Recall that a linear grammar is one which has only one variable on the right-hand side of any production rule.

EXAMPLE 7.2

Let us consider the formal language of *simple nesting* which, as we showed in Chapter 6, is not a regular language. As noted there, this language is useful in matching parentheses or brackets in programming languages. An example string in this language is ((((()))). Let us, for convenience, represent the opening parenthesis or bracket by the symbol a and the closing parenthesis or bracket by the symbol b. With this notation, the sample string becomes *aaaabbbb* or a^4b^4. In other words, the simple nesting language is nothing but our familiar language a^nb^n.

How do we construct a CFG for this language?

We first observe that the number of as must be equal to the number of bs that appear in the second half of a string in this language. Our experience with regular languages tells us that it is not a good idea to count the number of as and try to remember it. Instead, is there a way to add a b every time we add an a? This would ensure that the numbers of as and bs are equal. Whenever an a is added to the first half of the string, a b must be added to the second half. Moreover, if we do this, we need not remember how many as and bs have been added.

A grammar which can do this must have a production rule of the form:

$$S \rightarrow a...b$$

What should be there on the right-hand side of the production between the a and the b?

We observe that in a^nb^n, what we would have in the middle is $a^{n-1}b^{n-1}$. In other words, we can define strings in this language recursively; the start variable S itself can derive whatever is in the middle and therefore we can simply write:

$$S \rightarrow aSb$$

Of course, we need a way to terminate the recursion. This can be accomplished by adding:

$$S \rightarrow \lambda$$

These two simple productions constitute of our grammar for the simple nesting language:

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

No matter how hard we try, it is not possible to make this grammar right-linear or left-linear. There is no equivalent regular expression or finite automaton for this language. It is a CFG since both productions have just S, a single variable, on their left-hand sides. Let us see how it can derive the above string where $n = 4$:

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaaaSbbbb \Rightarrow aaaa\lambda bbbb = aaaabbbb$$

Figure 7.1 shows the parse tree corresponding to this derivation. Unlike in the case of regular grammars, this parse tree is not skewed to the left or right.

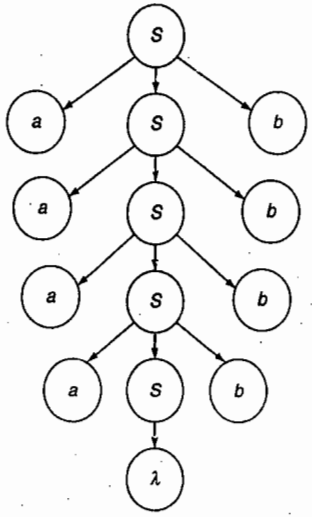


FIGURE 7.1 Parse tree for a derivation in the simple nesting language.

EXAMPLE 7.3

Now let us consider a similar example: the set of all *even palindromes* over the alphabet $\{a, b\}$, that is, the set of all strings of even length where the reverse of the string is identical to the string. We observe that in such a palindrome, the second half of any string is the reverse of the first half of the string. In other words, the last symbol is the same as the first symbol, the second the same as the last but one, and so on. This language is sometimes also represented as ww^R to indicate that the second half is a reverse of the first half of the string.

To generate such strings in a CFG, we can introduce the productions

$$S \to aSa \mid bSb \mid \lambda$$

so that every time an a is introduced at the beginning of the string, a corresponding a is introduced at the end of the string and similarly for b. Since these productions are applied recursively to expand S, and in each recursive step, either of the productions can be applied, it follows that the productions can generate every possible even palindrome over the given alphabet.

For example, the string *abbabbabba* can be derived from S as follows:

$$S \Rightarrow aSa \Rightarrow abSba \Rightarrow abbSbba \Rightarrow abbaSabba \Rightarrow$$

$$abbabSbabba \Rightarrow abbabλbabba = abbabbabba$$

From the derivation above, you can see that every sentential form in the derivation is also a valid even palindrome but has a variable S in the middle of it. When we finally replace the S with λ, we get the final form which is now a sentence, not a sentential form because it has no variables in it. We need the third production $S \to \lambda$ for this purpose.

From these two examples, we can see that CFGs can handle the two languages of simple nesting and even palindromes even though they are not regular languages. We see that the resulting grammars are not right-linear or left-linear. The variable S had to occur in the middle of the right-hand side of the productions in these grammars. One way to understand this is to observe that CFGs, unlike finite automata, are not constrained by having to process the input string left to right. Finite automata could not handle languages such as palindromes because, in having to process the string left to right, they had to try to remember every symbol that occurred in the first half of the string; this they clearly could not do since there was no limit on the size of these strings. They did not have the ability to go to the end of the string, match or otherwise control the symbol(s) there and come back to an earlier part of the string. CFGs are not bound by left-to-right processing. In being able to place the variable S in the middle of the right-hand side of a production, they have the ability to simultaneously control two parts of the string. This makes CFGs more powerful than the class of regular languages and grammars.

EXAMPLE 7.4

Suppose we want to write a CFG for the set of all odd palindromes over $\{a, b\}$. A string in this language is of an odd length because it has an extra symbol exactly in the middle. Hence, this language is often represented as wcw^R, where c is any symbol (or string) from the alphabet. The grammar for odd palindromes can be easily derived from the one above for even palindromes:

$$S \to aSa \mid bSb \mid c$$

Notice that the variable S terminates the derivation by generating a c instead of the λ in the previous grammar. This is fine since λ does not belong to the language of odd palindromes.

EXAMPLE 7.5

Consider the language $a^n b^{n+1}$ with one more b than a. The same technique that we used for odd palindromes works here too. A CFG for this language is:

$$S \to aSb \mid b$$

The first production generates an equal number of as and bs but it cannot eliminate S from the sentential form. The second production, in eliminating S, introduces the extra b to derive a valid string in the language.

EXAMPLE 7.6

What if the strings had twice as many bs as as, that is, the language $a^n b^{2n}$. In this case, the grammar needs to generate two bs every time it generates an a:

$$S \to aSbb \mid \lambda$$

EXAMPLE 7.7

Strings begin with as and end with bs but have more as than bs, that is, $a^n b^m$, where $n > m$.

The grammar needs to generate at least one more a than b, but it should be able to generate any number of additional as. This can be accomplished by introducing another production $S \to aS$ for the extra as, while ensuring that at least one extra a is generated by S itself before it disappears:

$$S \to aSb \mid aS \mid a$$

EXAMPLE 7.8

What about a language where the numbers of as and bs are unequal? Let us say the as still come before the bs, that is, $a^n b^m$, $n \neq m$. How do we deal with unknown numbers of as and bs?

It is not difficult to see that if the two numbers are unequal, there are really only two distinct possibilities: the first one is greater than the second or vice versa. We can extend the grammar from the previous example for each case to obtain a CFG for this language:

$S \to T \mid U$	Either case 1 or case 2
$T \to aTb \mid aT \mid a$	Case 1: more as than bs
$U \to aUb \mid Ub \mid b$	Case 2: more bs than as

Notice the production $U \to Ub$ in which the b appears after the U. This is important, for, if instead we had $U \to bU$, the grammar would generate strings that have bs before as and hence do not belong to the language. For example, here is such an incorrect derivation:

$$(S \Rightarrow U \Rightarrow bU \Rightarrow baUb \Rightarrow babb)$$

EXAMPLE 7.9

Our next language is the set of strings where the number of as is not equal to twice the number of bs that follow the as. In other words, the language is $a^n b^m$, $n \neq 2m$.

We begin by observing that if the numbers have to be unequal, then either the number of as should be more than twice the number of bs or the number has to be less than it. We treat this as two distinct cases:

$S \rightarrow T \mid U$ Either case 1 or case 2

$T \rightarrow aT \mid aW$ Case 1: at least one extra a

$U \rightarrow Ub \mid Wb$ Case 2: at least one extra b

$W \rightarrow aaWb \mid \lambda$ Generate all strings with twice as many as as bs

This covers almost the entire language except for those that contain just one a such as ab, abb, $abbb$, etc. To include these, we add the following additional rules:

$S \rightarrow aX$ Generate one a

$X \rightarrow Xb \mid \lambda$ Generate as many bs as needed

EXAMPLE 7.10

Consider the language $a^n b^m c^k$ where $k = n + m$. This is the *language of addition,* which, as we showed in Chapter 6, is not a regular language. We can easily construct a CFG for this language the moment we realize that a string in this language, such as *aaabbccccc*, has two parts: an inner part *bbcc* and an outer part *aaa...ccc*, that is,

$$a^n b^m c^k = a^n b^m c^m c^n$$

Hence, a CFG for this language is

$S \rightarrow aSc \mid T \mid \lambda$ Generate outer string and switch to variable T

$T \rightarrow bTc \mid \lambda$ Generate inner string

Thus, addition is a linear CFL. In other words, the set of all problems of adding two numbers can be modeled as a formal language that is linear context-free.

7.4 Constructing CFGs: Non-Linear Grammars

In all the examples above, the grammar was linear: there was always just one variable on the right-hand side of any production rule. Let us now look at examples that require non-linear productions.

EXAMPLE 7.11

Consider the way parentheses or brackets are nested in programming languages. As a formal language, we call this the language of *proper nesting*. Matching of nested parentheses is a practical problem in parsing and compilers. This problem is similar to that of simple nesting, $a^n b^n$, but with an

additional complexity: there is no longer a requirement that all as must precede any b. Instead, the requirements are as follows:

1. The string must start with an a, that is, an opening parenthesis.
2. At any point, counting from left to right (i.e., in any prefix), the number of closing parentheses (i.e., b) encountered thus far cannot be greater than the number of opening parentheses (i.e., a).
3. The number of opening and closing parentheses must be equal at the end of the string.

Let us consider an example of properly nested parentheses:

((((())()()))()), that is, *aaaaabbababbbabb*

How do we deal with the tree structure of this string?

It is not possible to construct a linear grammar for this language. Its structure is more complex: there can be many embedded $a^n b^n$s in this string. For example, the string has an *aabb* and three *ab*s in its interior parts. In order to visualize the structure of this string, let us re-write it as a tree structure as shown in Fig. 7.2.

Consider the lowest level of the structure shown. There appear to be three instances of $a^n b^n$ at this level: an $a^2 b^2$ followed by two $a^1 b^1$s. Each one of these can be modeled by the previous grammar but not a sequence of them. We, therefore, enhance the grammar by adding the non-linear rule:

$$S \rightarrow SS$$

The resulting grammar for properly nested parentheses is as follows:

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

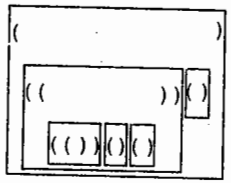


FIGURE 7.2 Tree structure of properly nested parentheses.

EXAMPLE 7.12

To continue this sequence of examples, let us consider the language of strings with equal numbers of a and b, where the as and bs can come in any order. This language is often represented as

$$n_a = n_b$$

It is true even in this language that every time we introduce an a, we must introduce a b as well. However, the b can come before the a or anywhere else for that matter. This language is similar to the language of properly nested parentheses but no longer requires that the number of bs cannot exceed

the number of as encountered in any prefix (i.e., up to any point in the string counting from the left). A suitable grammar is as follows:

$$S \to aSb \mid bSa \mid SS \mid \lambda$$

Any production in this grammar that adds an a also adds a b, thereby guaranteeing that their total numbers will always be equal. Also, there is no need to add productions of the kind

$$S \to abS$$

or

$$S \to baS$$

or

$$S \to Sab$$

or

$$S \to Sba$$

since these can be generated by $S \to SS$. For example, the string *babbabaaaababb* is derived as follows:

$$S \Rightarrow SS \Rightarrow bSaS \Rightarrow b\lambda aS \Rightarrow baSS \Rightarrow babSaS \Rightarrow babbSaaS \Rightarrow babbaSbaaS \Rightarrow$$
$$babba\lambda baaS \Rightarrow babbabaaaSb \Rightarrow babbabaaaaSbb \Rightarrow babbabaaaabSabb \Rightarrow$$
$$babbabaaaab\lambda abb = babbabaaaababb$$

Also, this is not the only possible derivation for this string in the grammar. Another way to understand this language is to count the difference between the number of as and the number of bs as we process the string from left to right. This difference must be zero both at the start and at the end of the string.

Does it ever become zero in between?

If it does, we have a non-linear, that is SS, situation. By including the difference within parentheses at each point in the string, we get:

$$(0)b(-1)a(0)b(-1)b(-2)a(-1)b(-2)a(-1)a(0)a(1)a(2)b(1)a(2)b(1)b(0)$$

The corresponding parse tree is shown in Fig. 7.3.

EXAMPLE 7.13

What about the complement of the above language? That is, those strings that have an unequal number of as and bs appearing in any order ($n_a \neq n_b$)? Appiying the technique used in Example 7.8 above, we can modify the grammar from Example 7.12 to obtain the required grammar:

$S \to T \mid U$	Either case 1 or case 2
$T \to aTb \mid bTa \mid TT \mid a \mid aT \mid Ta$	Case 1: at least one extra a
$U \to aUb \mid bUa \mid UU \mid b \mid bU \mid Ub$	Case 2: at least one extra b

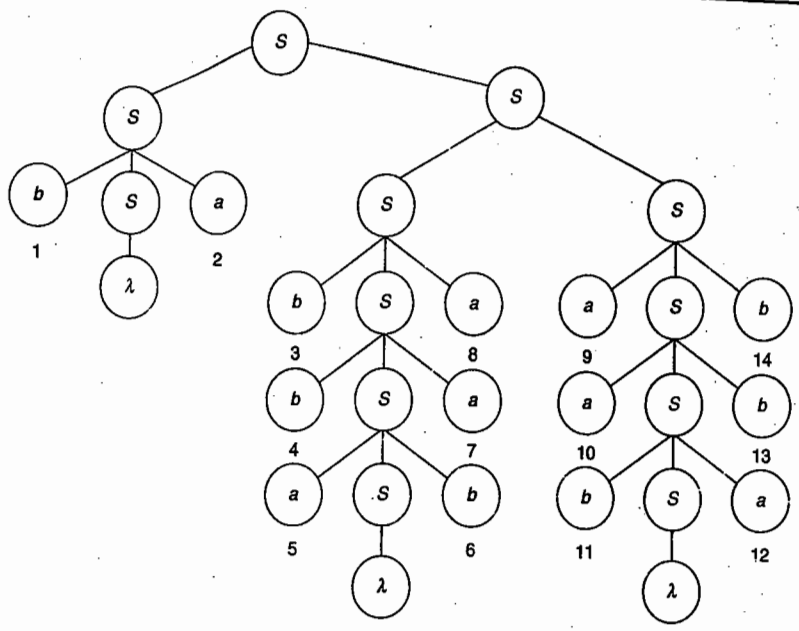


FIGURE 7.3 Parse tree for a non-linear grammar. Leaf nodes are numbered left to right.

EXAMPLE 7.14

We have seen how CFGs represent some aspects of the syntax of programming languages in our examples of simple nesting and proper nesting of parentheses and brackets. Let us consider the language of nested if-then-else statements whose alphabet includes **if, condition, then** and **else** as atomic symbols in addition to any other **statement** in the programming language and flower braces to demarcate proper nesting as desired. That is

$$\Sigma = \{\text{if, condition, then, else, statement, \{, \}}\}$$

An example string in the language is as follows:

If condition then statement else { if condition then statement else statement }

This language is very similar to proper nesting with one important difference: there is no requirement that every **if** should have an **else** clause and, as such, the number of **else** clauses can be less than the number of **if** s. Other characteristics remain the same: the string must begin with an **if**; and in any prefix, the number of **if**s should be greater than or equal to the number of **else**s. A grammar for this language is as follows:

$S \to$ if condition then S	Generate an if without an else clause
$S \to$ if condition then S else S	Generate an if with an else clause
$S \to \{\ S\ \}$	Optional flower braces
$S \to$ statement	Terminate with any other statement

EXAMPLE 7.15

Another significant part of the structure of a programming language is arithmetic expressions and the way in which they are nested and composed. If we consider the operators for addition, subtraction, multiplication and division along with optional parentheses while treating all variables and constants as atomic symbols, we get the alphabet

$$\{\text{variable, constant, } +, -, *, /, (,)\ \}$$

A CFG for this language is as follows:

$$S \rightarrow S + S$$

$$S \rightarrow S * S$$

$$S \rightarrow S - S$$

$$S \rightarrow S / S$$

$$S \rightarrow (S)$$

$$S \rightarrow \text{variable} \mid \text{constant}$$

A string such as $(x + 2.0) * y / (z - 6.0)$ can be derived as follows (assuming that any variable such as x, y or z and any constant such as 2.0 or 6.0 can be derived directly from S):

$$S \Rightarrow S * S \Rightarrow (S) * S \Rightarrow (S + S) * S \Rightarrow (x + S) * S \Rightarrow (x + 2.0) * S \Rightarrow$$

$$(x + 2.0) * S / S \Rightarrow (x + 2.0) * y / S \Rightarrow (x + 2.0) * y / (S) \Rightarrow (x + 2.0) * y / (S - S) \Rightarrow$$

$$(x + 2.0) * y / (z - S) \Rightarrow (x + 2.0) * y / (z - 6.0)$$

Before leaving the topic of constructing CFGs, it may be noted that just as we could construct a CFG (i.e., a non-regular grammar) for a regular language in Example 7.1, we can also construct a non-context-free grammar for a CFL. For example, here is a *context-sensitive grammar* for the CFL of simple nesting, $a^n b^n$:

$S \rightarrow aTb \mid \lambda$	T is an unnecessary variable; it behaves the same way as S
$aT \rightarrow aaTb$	An unnecessarily context-sensitive rule
$Tb \rightarrow aTbb$	An unnecessarily context-sensitive rule
$T \rightarrow \lambda$	

A formal language does not become a context-sensitive language merely because one can construct a context-sensitive grammar for it. As we will see in Chapter 9, a language is context-sensitive only if there can be no CFG for it.

7.5 Constructing CFGs: *Mantras*

In the examples so far, we practiced the art of constructing CFGs for a variety of languages. Although there is no mechanical procedure for constructing grammars, we were able to identify a set of *mantras* for

writing accurate grammars in Chapter 5. The same *mantras* are also applicable to CFGs. The following additional tips may be useful in the case of CFGs:

1. Since CFGs have the ability to generate strings with a tree structure, we need not be constrained by left-to-right processing of strings. Thinking in terms of parse trees and actually constructing parse trees for a few sample strings in the language will help in writing a suitable grammar.
2. As we will see in Chapter 9, CFLs have two parts in their strings that are correlated with one another. For example, the number of opening parentheses must match the number of closing parentheses or the number of **if** s must match or exceed the number of **else** s, and so on. Figure out what is the correlation in the given language. To model such correlated behaviors, CFGs can add corresponding elements in two places in the string being generated by having non-regular productions (i.e., rules of the form $S \rightarrow \ldots S \ldots$ where the variable is in the middle of the right-hand side instead of being rightmost or leftmost).
3. CFGs can also be non-linear, that is, they can have more than one variable on the right-hand side of a production rule. This gives them the ability to concatenate sub-strings (or join sub-trees) each of which can be from its own CFL. A single non-linear production of the form $S \rightarrow SS$ is often sufficient. If you determine that the given language is non-linear, apply the counting technique used in Example 7.12 to mark the boundaries of the sub-strings to be concatenated. This will help in separating the linear and non-linear parts of the grammar.
4. It is often useful to compare the given language to one of the known CFLs such as the languages of simple nesting, proper nesting, addition, if-then-else and arithmetic expressions. Differences, if any, can be accommodated by modifying the production rules appropriately.
5. CFLs usually have some parts that are *regular* as well, that is, in addition to their correlated context-free behaviors, strings in the language contain parts that can be modeled using regular expressions. For example, one or more extra symbols (i.e., $a^n b^{n+1}$), some symbols in the middle (i.e., odd palindromes) and so on. Figuring out these will ease the construction of suitable productions (which by themselves tend to be right or left-linear) for modeling the *regular* parts of the language.

7.6 Introduction to Parsing

We saw in Chapter 5 how a grammar can be applied to parse a given string to verify its grammaticality and to construct a parse tree. When we try to apply a simple parsing algorithm such as the following *top-down* (deriving or generating) algorithm to a CFG, two problems arise:

Algorithm

Start with just the variable S as the initial sentential form;
Find a production whose left-hand side matches one of the variables in the sentential form
    Replace the variable by the right-hand side of the production;
    If the resulting sentential form does not match the given string,
    try a different production.
Until the given string is derived completely.

First, it results in an exhaustive search of all possible derivations in the grammar and is highly inefficient. Even if we invert the algorithm to try a *bottom-up* parsing algorithm, it still searches among many different ways of parsing the given sentence only some of which complete the parsing by reducing the sentential form to the start symbol S. The theory and practice of *compiler design* have produced a variety of techniques and efficient algorithms for making parsing efficient so that a compiler can parse a long program rather quickly. For our purposes here, it suffices to note that grammars themselves can often be re-designed to make parsing efficient, as we will see in the following sections.

Apart from being highly inefficient and impractical, an exhaustive-search parser that generates all possible strings in the language to match the given input would never halt when given a string that is *not in* the language; it will go on generating strings that are *in* the language forever. Imagine a compiler that goes into an infinite loop every time it is given a program with errors! This is similar to the *halting problem* that we will encounter later in Chapter 12 for undecidable problems, but, fortunately for us, there are better algorithms for parsing with CFGs. But first, we must get the grammar into a cleaner shape and then find an efficient parsing algorithm.

Second, often there is no unique parse for a given string. We will see in the following section on ambiguity how the same string can result in two or more different parse trees. This is highly undesirable from the point of view of compiler design: *if there are two or more ways of interpreting a given program, how should the compiler translate it to executable code which might implement a different semantics than what was intended by the programmer?* The design of programming languages addresses this problem by ensuring that valid programs in the language are unambiguous, that is, they result in one and only one parse tree.

To address both issues, in the remaining sections of this chapter, we explore how to recognize and eliminate ambiguity in a grammar and how to convert a grammar to better *normal forms* so that parsing can be efficient.

7.7 Ambiguity

Consider the parse tree shown in Fig. 7.3 obtained for the string *babbabaaaababb* in Example 7.12. There is another possible derivation for the same string:

$$S \Rightarrow SS \Rightarrow SaSb \Rightarrow SaaSbb \Rightarrow SaabSabb \Rightarrow Saab\lambda abb \Rightarrow SSaababb \Rightarrow SbSaaababb \Rightarrow$$

$$SbbSaaaababb \Rightarrow SbbaSbaaaababb \Rightarrow Sbba\lambda baaaababb \Rightarrow bSabbabaaaababb \Rightarrow$$

$$b\lambda abbabaaaababb = babbabaaaababb$$

Figure 7.4 shows the corresponding parse tree for the string. Although the two parse trees look rather similar, the way in which the three sub-trees are attached towards the top of the tree is different between Figs. 7.3 and 7.4. This may be significant in terms of the semantics of the language being parsed. Such a difference can be illustrated by using a well-known example from natural language:

I saw the man on the hill with the telescope.

Various interpretations are possible for this sentence and a couple of them along with the corresponding parses are shown in Fig. 7.5. As speakers of a language such as English, we usually have a "normal" interpretation for such sentences and thus we are not bothered by parsing or syntactic ambiguities.

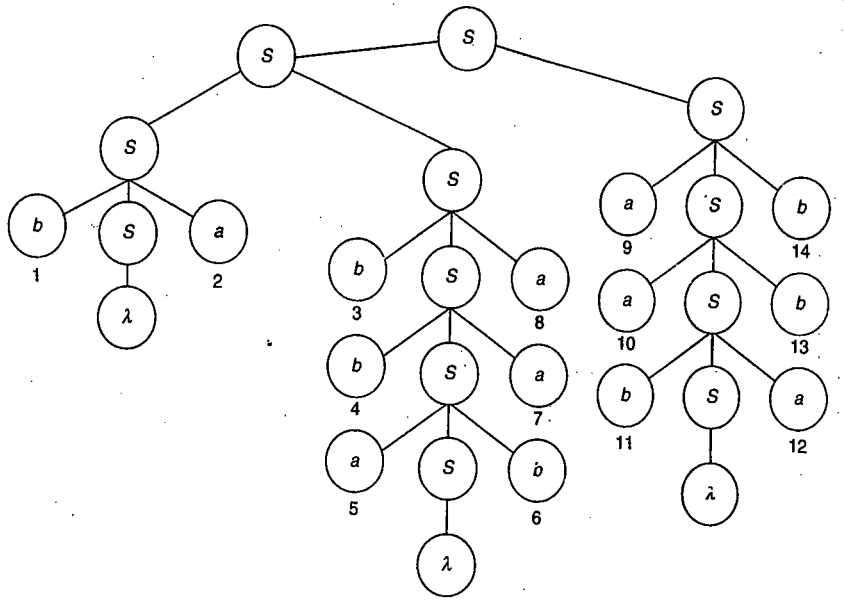


FIGURE 7.4 Another parse tree with a rightmost derivation.

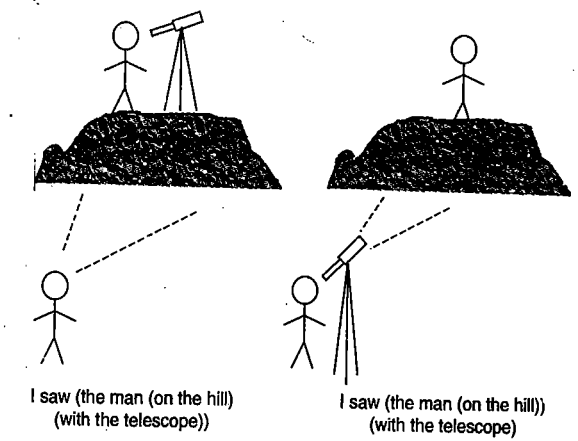


FIGURE 7.5 Two of the multiple interpretations for an ambiguous sentence.

Compilers, in particular, and computers, in general, have no such intuition and therefore we need to make our grammars as unambiguous as possible.

Whenever it is possible to obtain more than one parse tree for the same string, we say that the grammar is *ambiguous*. In such cases, we try to differentiate between the different parses (or parse trees) by looking at which variable in a sentential form (when more than one existed) was chosen for

expansion during derivation. In the parse tree shown in Fig. 7.3, the leftmost variable was always chosen for expansion. Such a derivation is called a *leftmost derivation*. Similarly, the parse tree shown in Fig. 7.4 was obtained from a *rightmost derivation* in which the rightmost variable in the sentential form is always expanded first.

EXAMPLE 7.16

As another example of ambiguity in a CFG, consider the arithmetic expression:

$$a + b / c$$

There are two possible leftmost derivations of this string using the grammar in Example 7.15:

$$S \Rightarrow S + S \Rightarrow a + S \Rightarrow a + S / S \Rightarrow a + b / S \Rightarrow a + b / c$$

or

$$S \Rightarrow S / S \Rightarrow S + S / S \Rightarrow a + S / S \Rightarrow a + b / S \Rightarrow a + b / c$$

By following a convention to always attempt a leftmost (or a rightmost) derivation, it is sometimes possible to obtain a consistent parse even from an ambiguous grammar. However, for some languages and grammars, even such a convention results in ambiguity, that is, we get more than one leftmost derivation (or more than one rightmost derivation) for the same string in the language. Such a grammar is said to be ambiguous. An ambiguous grammar provides more than one leftmost derivation (or more than one rightmost derivation) for the same string. A formal language is said to be *inherently ambiguous* if every grammar for the language is ambiguous. Fortunately, not all languages are inherently ambiguous and many ambiguous grammars can be converted to equivalent unambiguous grammars.

7.8 Eliminating Ambiguity

Ambiguity is of several kinds. *Lexical ambiguity* or *word-sense ambiguity* arises when a word in a natural language or a token or symbol in a programming language has multiple meanings. Lexical ambiguity has been pretty much eliminated in programming languages using reserved and key words and allowing overloading of operators and method names only where the context helps us to disambiguate the meaning. *Structural ambiguity* or *syntactic ambiguity* arises when there are multiple parse trees, as in the examples from the previous section. Structural ambiguities exist in programming languages, for example, in nesting of if-then-else statements and in precedence of operators in arithmetic expressions.

In general, there are three ways of working around such ambiguities:

1. Relying on the programmer to always disambiguate by providing parentheses or brackets appropriately (usually not a good choice).
2. Using operator precedence rules outside of the grammar, that is, in later stages of the compiler.
3. Re-writing the grammar by introducing new non-terminals to encode the rules of operator precedence in the grammar itself.

EXAMPLE 7.17

Let us see how to re-write the expression grammar from Example 7.15 to make it unambiguous. The ambiguity arose in the grammar because the productions for different arithmetic operators could be applied in any order. Rules of precedence which are applied conventionally in programming languages tell us that * and / have a higher precedence than + and –. These rules can be incorporated into the grammar itself by introducing new variables for intermediate nodes in the parse tree as follows:

$S \rightarrow (S) \mid S + T \mid S - T \mid T$	An expression has a + or – operator or is just a term
$T \rightarrow T * F \mid T / F \mid F$	A term has a * or / operator or is just a factor
$F \rightarrow$ variable | constant	A factor is just a variable or a constant

Notice that no variable occurs twice on the right-hand side of any production rule, thereby eliminating any structural ambiguity. The string $a + b / c$ from Example 7.16 has only one leftmost derivation now:

$$S \Rightarrow S + T \Rightarrow T + T \Rightarrow F + T \Rightarrow a + T \Rightarrow a + T / F \Rightarrow a + F / F \Rightarrow a + b / F \Rightarrow a + b / c$$

From the parse tree for this derivation, shown in Fig. 7.6, it is clear that the division must be performed before the addition (if a leftmost derivation is the convention in the given programming language), thereby making the expression unambiguous even in the absence of parentheses in the expression.

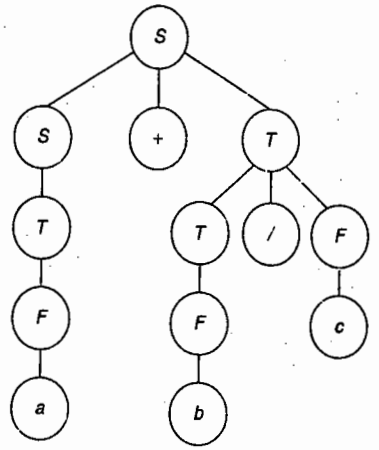


FIGURE 7.6 Parse tree for leftmost derivation using unambiguous grammar.

7.9 *The Idea of* Chomsky Normal Form

We have learned that grammars are not unique; the same formal language can have multiple grammars. Some grammars are simpler and more elegant than others; some lead to better parsers and faster compilers. We will now see how to convert a grammar to a *normal form* that gives us cleaner grammars with good algorithms for parsing.

The most well known normal form for CFGs is the *Chomsky Normal Form* (CNF). The idea of CNF is very simple: each production in the grammar either generates a terminal symbol (i.e., a symbol in the input string without further expanding the sentential form) or expands the sentential form by exactly 1 (i.e., increases its length by 1). Therefore, productions can only be of two forms:

$A \rightarrow a$ Replaces a variable in the sentential form by a terminal symbol

$A \rightarrow BC$ Replaces a variable by two variables to grow the sentential form

If every production in the grammar is in one of the above two forms, we say that the grammar is in Chomsky Normal Form.

Another way to understand the CNF is that *it makes the parse tree a binary tree.* In a binary tree, a node may have either one or two child nodes. In a parse tree obtained from a CNF grammar, if a node has just one child, that child is a terminal symbol (from a production of the form $A \rightarrow a$); if a node has two children, they are both internal nodes representing non-terminals (from a production of the form $A \rightarrow BC$).

In addition, a grammar in CNF must be clean, that is, it must not have any:

1. *Lambda productions* of the form $A \rightarrow \lambda$. Variables having such productions are called *nullable variables*. They are wasteful in the sense that a variable is introduced in the sentential form only to be replaced by null at a later time. There is no need to introduce such productions in the grammar.
2. *Unit productions* of the form $A \rightarrow B$ which merely replace one variable by another without making any progress in the derivation. The length of the sentential form remains the same and no terminal symbol is generated when such a production is applied in a derivation.
3. *Useless variables* which are of two kinds:
   - *Non-generating* ones which never lead to terminal symbols. If a non-generating variable is introduced into the sentential form, it is a dead-end; the sentential form can never generate any string in the language since there is no way to get rid of the non-generating variable.
   - Productions for *unreachable variables.* An unreachable variable, on the other hand, does not appear on the right-hand side of any production (to make it reachable from the start symbol S) and hence will never appear in a sentential form. Productions with unreachable variables on their left-hand sides are useless and can be safely removed from the grammar.

We will now present a method for converting a CFG to the CNF and use it to show how a parsing algorithm can be applied to the grammar.

7.10 Converting to Chomsky Normal Form

The algorithm for converting a CFG to the CNF is based on *the idea of substitution.* If a grammar has a set of productions $A \rightarrow aBc$ and $B \rightarrow x \mid y \mid z$, where x, y, z stand for any piece of a sentential form, that is, any sequence of terminal or non-terminal symbols, then we can eliminate the B productions by substituting the right-hand sides of those productions wherever B occurs in other productions. In the A production, we can substitute for B to obtain a new set of productions:

$$A \rightarrow axc \mid ayc \mid azc$$

The above substitution rule gives us a method for chaining production rules to derive a sentential form. This can be applied to eliminate undesirable elements in a grammar as outlined below.

The following algorithm (or method) converts any CFG to its equivalent CNF.

Algorithm

1. *Eliminate lambda productions:* If there is a lambda production $A \rightarrow \lambda$, delete the production after changing all other productions where A occurs on the right-hand side as follows:
   In a production $B \rightarrow xAy$, remove the nullable A by replacing it with:
   $B \rightarrow xy \mid xAy$ If there is some other production that expands A or just
   $B \rightarrow xy$ If there is no other production for A
2. *Eliminate unit productions:* For every unit production $A \rightarrow B$, delete the production by replacing any occurrence of A with B on the right-hand sides of all other productions, that is, replace
   $C \rightarrow xAy$ *by* $C \rightarrow xAy \mid xBy$
   (or just $C \rightarrow xBy$ if there is no other production for A).
3. *Eliminate useless variables:* Simply delete all productions containing unreachable and non-generating variables. A variable is useful if it is a part of the derivation of at least one string in the language of the grammar; useless ones are those do not take part in any derivation as a result of one of two things: (a) they are unreachable from S (i.e., S cannot derive the variable in any sentential form) or they are non-generating (i.e., they cannot get rid of themselves to introduce only terminal symbols).
4. *Re-write productions in CNF form* as follows:
   Productions which are already of the form $A \rightarrow a$ or $A \rightarrow BC$ are fine;
   If a production $A \rightarrow BCD$ has more than two symbols on the right-hand side,
   introduce new, intermediate variables and productions of the form:
   $A \rightarrow BE$ and $E \rightarrow CD$
   If a production $A \rightarrow bC$ has a terminal and a non-terminal,
   re-write it in the form:
   $A \rightarrow BC$ and $B \rightarrow b$ both of which are in CNF.

In eliminating lambda productions, if there is more than one nullable variable on the right-hand side of a production, we have to be careful in using the substitution rule to include all subsets of nullable variables. For example, if $A \rightarrow xByCzDw$, where lambda productions $B \rightarrow \lambda$, $C \rightarrow \lambda$, $D \rightarrow \lambda$ exist (where other non-null productions also exist for each of them), the substitution should result in the following:

$A \rightarrow xyzw$	All three are nulled
$\mid xByzw \mid xyCzw \mid xyzDw$	Only one of them is not null
$\mid xByCzw \mid xByzDw \mid xyCzDw$	Any two of them are not null
$\mid xByCzDw$	All are non-null

EXAMPLE 7.18

Let us remove lambda productions from the grammar:

$$S \to ABaC$$
$$A \to BC$$
$$B \to b \mid \lambda$$
$$C \to D \mid \lambda$$
$$D \to d$$

Here B and C are nullable. However, since $A \to BC$, both of which are nullable, so A is also nullable! Considering all the subsets of $\{A, B, C\}$, we get

$$S \to ABaC \mid BaC \mid AaC \mid ABa \mid Aa \mid Ba \mid aC \mid a$$
$$A \to BC \mid B \mid C$$
$$B \to b$$
$$C \to D$$
$$D \to d$$

EXAMPLE 7.19

Let us get rid of unit productions in the grammar:

$$S \to A \mid Bb$$
$$A \to B \mid Bc$$
$$B \to a \mid aa$$

Here, $S \to A$ and $A \to B$ are the unit productions. Substituting in place of A everything to which A can expand, and similarly for B everything to which B can expand, we get

$B \to a \mid aa$	No change
$A \to a \mid aa \mid Bc$	Substitute for B in the unit production
$S \to a \mid aa \mid Bc \mid Bb$	Substitute for A in the unit production

In the worst case, there may be circular unit productions, for example, $A \to B$ and $B \to C$ and $C \to A$. In such a case (see Exercise 51 for example), we have to order the variables in the circular unit productions by constructing a *dependency graph* which is a graph with the variables in the grammar as its nodes. There is an edge between any two nodes A and B whenever there is a production with A on its left-hand side and B on its right-hand side. If the dependency graph is acyclic, then there is no circular dependency among the variables in the grammar. If it has a cycle, then unit productions can still be eliminated by substituting for the variables on the right-hand sides, the same way that we eliminated λ-productions. For example, to eliminate $A \to B$, we replace it with $A \to x \mid y \mid z$, where the only productions for B are: $B \to x \mid y \mid z$ (none of which is a unit production containing A).

EXAMPLE 7.20

Let us eliminate useless variables from the grammar:

$$S \to ABaC \mid Db$$
$$A \to AC \mid ab$$
$$B \to bc$$
$$C \to c$$
$$E \to e$$

In this grammar, A, B, C and D are reachable variables but not E. The variables A, B, C and E are generating variables but not D. As such, any production that contains either D or E can be deleted safely. The resulting clean grammar is as follows:

$$S \to ABaC$$
$$A \to AC \mid ab$$
$$B \to bc$$
$$C \to c$$

It is very important to perform the above three operations in exactly the same order. Eliminating lambda productions can introduce new unit productions. Similarly, eliminating unit productions can result in useless productions. A good way to remember this order is to learn the *mantra*

Lambda units are useless.

This means, first lambda productions are to be removed, then unit productions and finaliy useless productions.

Let us work through a couple of complete examples to see how all of the above kinds of undesirable elements of a grammar can be eliminated in order to convert the resulting grammar to CNF.

EXAMPLE 7.21

Consider a CFG for all palindromes over the alphabet $\{a, b\}$, that is, the union of all even palindromes and all odd palindromes:

$$S \to aSa \mid bSb \mid A \mid \lambda$$
$$A \to a \mid b \mid \lambda$$

Let us convert this grammar to CNF:

Step 1: Eliminating lambda productions – S and A both are nullable.

$S \to aSa \mid aa \mid bSb \mid bb \mid A$	S is nullable
$A \to a \mid b$	No need for the lambda production

It may be noted that the modified grammar cannot generate the null string λ. If it is necessary to include λ in the language, a separate rule $S \to \lambda$ will have to be retained.

Step 2: Eliminating unit productions – $S \rightarrow A$ is the only unit production.

$$S \rightarrow aSa \mid aa \mid bSb \mid bb \mid a \mid b$$
$$A \rightarrow a \mid b$$

As a result of removing lambda and unit productions, A became an unreachable variable and will be removed in the next step.

Step 3: Eliminating the useless variable A, we get

$$S \rightarrow aSa \mid aa \mid bSb \mid bb \mid a \mid b$$

Step 4: Finally, we need to re-write the first four productions above in CNF by introducing new intermediate variables:

$S \rightarrow XZ$	For aSa
$X \rightarrow a$	
$Z \rightarrow SX$	i.e., Sa
$S \rightarrow XX$	i.e., aa
$S \rightarrow YW$	For bSb
$Y \rightarrow b$	
$W \rightarrow SY$	i.e., Sb
$S \rightarrow YY$	i.e., bb
$S \rightarrow a$	
$S \rightarrow b$	

EXAMPLE 7.22

Let us convert the unambiguous expression grammar from Example 7.17 to CNF:

$S \rightarrow (S) \mid S+T \mid S-T \mid T$	An expression has a + or – operator or is just a term
$T \rightarrow T*F \mid T/F \mid F$	A term has a * or / operator or is just a factor
$F \rightarrow$ variable $\mid$ constant	A factor is just a variable or a constant

Note that **variable** and **constant** are treated as terminal symbols in this grammar. A complete grammar for a programming language might have a separate grammar or regular expression for handling the set of valid variable names and the set of valid constants.

Step 1: There are no nullable variables and no lambda productions to eliminate.

Step 2: Eliminate unit productions – $S \rightarrow T$ and $T \rightarrow F$ are the unit productions. Substituting, we get

$F \rightarrow$ variable $\mid$ constant	
$T \rightarrow T*F \mid T/F \mid$ variable $\mid$ constant	Substituting for F
$S \rightarrow (S) \mid S+T \mid S-T \mid T*F \mid T/F \mid$ variable $\mid$ constant	Substituting for T

Step 3: There are no unreachable or non-generating variables. All variables are useful.

Step 4: Re-writing in CNF, we get

$F \rightarrow$ variable $\mid$ constant	
$T \rightarrow TP$	For $T * F$
$P \rightarrow MF$	
$M \rightarrow *$	
$T \rightarrow TQ$	For T / F
$Q \rightarrow DF$	
$D \rightarrow /$	
$T \rightarrow$ variable $\mid$ constant	
$S \rightarrow OR$	For (S)
$R \rightarrow SC$	
$O \rightarrow ($	Open parenthesis
$C \rightarrow)$	Close parenthesis
$S \rightarrow SA$	For $S + T$
$A \rightarrow UT$	
$U \rightarrow +$	
$S \rightarrow SL$	For $S - T$
$L \rightarrow BT$	
$B \rightarrow -$	
$S \rightarrow TP \mid TQ \mid$ variable $\mid$ constant	For $T * F$ and T / F

7.10.1 Parsing with Chomsky Normal Form

CNF grammars, being clean and normalized so as to result in a binary parse tree, allow a neat parsing algorithm to be applied. The idea of the algorithm is to enumerate all the variables that can generate any sub-string of the string to be parsed. We begin with the smallest sub-strings, namely, the individual terminal symbols $w[1]$, $w[2]$, ..., $w[n]$ in the string w. Next, we enumerate variables, if any, which can generate all the substrings of length 2:

$$w[1..2], w[2..3], w[3..4], \ldots, w[n-1..n]$$

then length 3:

$$w[1..3], w[2..4], w[3..5], \ldots, w[n-2..n]$$

and so on, until at the end, we figure out which variables, if any, can generate the entire string $w[1..n]$. If S is one of the variables that can generate the entire string, then we have found a derivation for the given string and it is grammatical. If S is not in the set, then the given string is not grammatical and does not belong to the language of the grammar. This algorithm is known as the *CYK Algorithm* named after the three scientists who invented it: J. Cocke, D. H. Younger and T. Kasami.

What is nice about the algorithm is that we can directly read the enumerations for a longer string by examining the enumerated variables for shorter substrings. This is made possible by the CNF which ensures that the parse tree is a binary tree. As a consequence, we need to find ways of composing longer strings using only two sub-strings at a time. No string can be composed of three or more sub-strings in a CNF grammar since no production has more than two symbols on its right-hand side. The algorithm works as follows:

Algorithm

For each individual symbol w[i] in the given string w,
List all the variables A which have a production A → w[i]
For length l = 2 to length n of the given string,
For each substring w[i..j] of length l,
List all the variables A which have a production A → BC
where for some k between i and j,
B is already enumerated for the substring w[i..k] and
C is already enumerated for the substring w[k+1..j]
If S is enumerated for the entire string w[1..n], accept the string w;
else, reject the string w.

The enumeration of variables as required by the algorithm can be carried out elegantly by filling a matrix or table, the same way we used table-filling in minimizing an automaton in Chapter 3. An example will serve to illustrate the method.

EXAMPLE 7.23

Parse the string *abbaabab* with the grammar from Example 7.12 for the language of strings with equal numbers of *a* and *b*, where the *a*s and *b*s can come in any order (i.e., $n_a = n_b$):

$$S \rightarrow aSb \mid bSa \mid SS \mid \lambda$$

Converting this grammar to CNF, we get (ignoring the null string λ):

$S \rightarrow AB \mid BA$	*aSb* and *bSa* with *S* nulled
$S \rightarrow AC$	For *aSb*
$S \rightarrow BD$	For *bSa*
$S \rightarrow SS$	No change required
$A \rightarrow a$	
$B \rightarrow b$	
$C \rightarrow SB$	
$D \rightarrow SA$	

We now enumerate all the variables that can generate sub-strings of length one. There are only two such sub-strings – *a* and *b* – and they are generated only by *A* and *B*, respectively. These entries are shown in the bottom row of Table 7.1.

Next we consider sub-strings of length two in the given string:

$w[1..2] = ab = a.b \Leftarrow AB \Leftarrow S$	
$w[2..3] = bb$	Not derivable
$w[3..4] = ba = b.a \Leftarrow BA \Leftarrow S$	
...	
$w[7..8] = ab = a.b \Leftarrow AB \Leftarrow S$	

TABLE 7.1 CYK Algorithm for Parsing with a Chomsky Normal Form Grammar

8	*S*							
7	*D*	*C*						
6	*S*	*S*	*S*					
5	*D*	*C*	*D*					
4	*S*	*S*	*S*		*S*			
3	*C*		*D*		*D*	*C*		
2	*S*		*S*		*S*	*S*	*S*	
1	*A*	*B*	*B*	*A*	*A*	*B*	*A*	*B*
Length	***a***	***b***	***b***	***a***	***a***	***b***	***a***	***b***

A few other entries are illustrated here:

$w[1..3] = abb = a.bb$ or $ab.b \Leftarrow ABB \Leftarrow SB \Leftarrow C$
$w[3..5] = baa = b.aa$ or $ba.a \Leftarrow BAA \Leftarrow SA \Leftarrow D$
$w[1..4] = abba = ab.ba \Leftarrow ABBA \Leftarrow SS \Leftarrow S$
$w[2..5] = bbaa = b.ba.a \Leftarrow BBAA \Leftarrow BSA \Leftarrow BD \Leftarrow S$
$w[1..6] = abbaab = abba.ab \Leftarrow SS \Leftarrow S$
$w[1..7] = abbaaba = abbaab.a \Leftarrow SA \Leftarrow D$
$w[2..8] = bbaabab = bbaaba.b \Leftarrow SB \Leftarrow C$
$w[1..8] = abbaabab = abbaab.ab \Leftarrow SS \Leftarrow S$

In filling the entries in the table, a given string must be broken down into concatenations of two sub-strings in all possible ways. For example,

$w[1..6] = w[1..5].w[6]$ or $w[1..4].w[5..6]$ or $w[1..3].w[4..6]$ or $w[1..2].w[3..6]$ or $w[1].w[2..6]$

At the time we are computing the table entry for $w[1..6]$, we will have already computed the entries for all of the above sub-strings. We simply need to read the corresponding entries and see if there is any production of the form $X \rightarrow YZ$, where *Y* and *Z* are the entries found; if so, we enter the variable *X* in the cell for $w[1..6]$. This process is illustrated in Fig. 7.7. In our present example, to fill up the entry for $w[1..6]$ (shown circled in the figure), we have to find a production having *DB* or *SS* or *C*? or *SS* or *AC* on its right-hand side. Of these, we have $S \rightarrow AC$ and $S \rightarrow SS$ in this grammar. Hence we enter *S* for $w[1..6]$.

As always, it helps to note the meanings of each of the variables in the grammar:

1. S generates the set of all strings with equal numbers of as and bs.
2. C generates all strings with one extra b.
3. D generates all strings with one extra a.

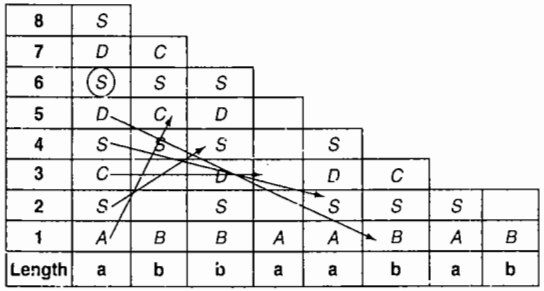


FIGURE 7.7 Making entries in the table in the CYK algorithm for parsing.

Although all entries had at most one variable in this example, the entries are sets of variables in the general case. Also, it is useful to note that if the entry for a sub-string contains the start variable S, then and only then that sub-string, by itself, is a member of the language of the grammar (i.e., the variable S can actually generate that sub-string). If an entry is blank, no variable in the grammar can generate it, not just S, and clearly it does not belong to the language.

7.11 *The Idea of* Greibach Normal Form

Another normal form that is well-known is the *Greibach Normal Form* (GNF). The idea of GNF is to make parsing or derivation linear, that is, every step in the derivation should introduce (or parse) exactly one terminal symbol from the input string. With a grammar in GNF and a leftmost derivation, in each step, one more symbol, from left to right, is generated so that exactly n steps later, a string of length n is derived. Every production in GNF looks like either of the following:

$$A \rightarrow aBCD\ldots$$
$$A \rightarrow a$$

The leftmost symbol on the right-hand side of every production is a terminal symbol. There cannot be any further terminal symbols in the production. Instead, there can be zero or more non-terminals to the right of the terminal symbol. There is no particular algorithm for converting a given CFG to GNF. The following example serves to illustrate the method.

EXAMPLE 7.24

Let us convert our grammar for equal numbers of a and b from Examples 7.12 and 7.23 to GNF. The given grammar is as follows:

$$S \rightarrow aSb \mid bSa \mid SS \mid \lambda$$

Converting to GNF, we get:

$S \rightarrow aSB \mid bSA$ Introducing variables A for a and B for b
$A \rightarrow a$
$B \rightarrow b$

What do we do with $S \rightarrow SS$?

We need to somehow introduce a terminal at the beginning. This can be done by applying the same kind of substitution that we employed in converting to CNF. We need to find a substitute for S which begins with a terminal symbol to obtain a complete GNF grammar with the following additional productions:

$S \rightarrow aSBS \mid bSAS$ Substituting for the first S from the above rules

7.12 Simple, Linear and Other Grammars

We have already seen linear CFGs. All regular grammars are of course linear. A good characteristic of linear grammars is that, since there is only one variable on the right-hand side of any production, the sentential form always contains just one variable (except when the entire string is generated and there are no variables). The parser needs to consider only productions for that one variable. This reduces the number of choices for the parser, thereby making parsing more efficient.

In general, inefficiency in parsing is caused by the number of production rules that can be applied to the present sentential form. Each choice leads the parser in a particular direction and with a compounding effect of a sequence of choices, in natural languages or non-trivial formal languages with non-trivial grammars, there is usually a combinatorial explosion in the number of paths. To successfully parse or derive a string, the parser has to search for the right path that leads to the given string. Even in a linear grammar, there are still multiple choices since there can be more than one production with the same variable on the left-hand side.

An effective way to reduce the number of choices and thereby the complexity of parsing or derivation is to ensure that each production rule being applied matches the "next" terminal symbol in the input string being generated or parsed. For example, GNF ensured that every production began with a terminal symbol. A production can be applied in parsing with GNF only if its terminal symbol matches the next symbol in the input string. For example, if the current sentential form is $aaSBB$ and the complete input string is $aabbaab$ and the available productions for S are as follows:

$S \rightarrow aSB$ Next input symbol must be a
$S \rightarrow bSA$ Next input symbol must be b

only the second production is applicable. The first one generates a sentential form $aaaSBBB$ which does not match the input string (in its third symbol from the left).

In general, in GNF, there can still be more than one production rule that matches a given sentential form (even in a leftmost derivation). This choice is eliminated in a stricter type of grammar known as *simple grammar* or *s-grammar* in which, for a given variable T and given terminal symbol a, there can be only one production rule of the type $T \rightarrow aXYZ$.

There is also a sub-class of CFGs known as *deterministic context-free grammars*. We will study deterministic context-free languages and grammars later in Chapter 11 (Sec. 11.12) when we look at a complete hierarchy of all classes of formal languages.

Programming languages are designed to be mostly context-free. In fact, most parts of the syntax of modern programming languages can be specified using simple grammars. This makes compiling of programs written in those languages rather efficient. Modern compilers employ even more efficient methods by carrying out a *look ahead* in the input string. That is, before deciding which production rule to apply, they study what the next k symbols in the input are, so that they can efficiently choose the right production for parsing the given string. The resulting grammar types such as $LL(k)$ grammars are studied extensively in compiler design.

The syntax of programming languages is usually specified using a grammar notation known as *Backus–Naur Form* (BNF) which is pretty much the same as the notation that we have been using. However, non-terminals are enclosed in angle brackets and the symbol ::= is used instead of the arrow to separate the two sides of a production rule in BNF. For example, a BNF grammar for nested if-then-else statements is as follows:

```
<statement> ::= if <condition> then <statement> |
      if <condition> then <statement> else <statement>
```

Most programming languages do contain some structures that are not even context-free. For example, whenever we declare a sequence of variables or function or method signatures and later re-use them in the same order, for example, by providing the function implementation or the method body, we have a pattern such as $w.w$, where w is any string of symbols. This, unfortunately, is not a CFL, as we will see in Chapter 9. Later in Chapters 10 and 11 we will see how to construct a computing machine or define a grammar for such context-sensitive languages.

As a final example, let us try extending the simple nesting language by including a third symbol c: $a^n b^n c^n$. Surprisingly, it turns out that we cannot construct a CFG for this language (which we will prove in Chapter 9). We know that, unlike regular grammars and finite automata, CFGs are capable of counting – remembering the number of as in $a^n b^n$ and matching the count to the number of bs in another part of the input string. Yet, they are unable to handle this example where all three parts of the input must have equal lengths. It is as though CFGs, while being much more powerful than their regular counterparts, have a use-and-throw kind of memory. If they use the count of as to match with bs, they lose it! If they recall what they remembered, they also erase the memory. We will see in the next chapter why this is so when we look at computing machines called pushdown automata which use a stack to count, that are equivalent to CFGs.

Such a behavior of memory is typical of a stack data structure. If we use a stack to count a number by pushing elements onto the stack, to make use of that number (e.g., to match it to a count of further symbols in the input string), we must pop the stack. At the end of this use of the memory, the stack is empty; the machine forgot what the original number was. It cannot use it a second time to match the number of cs in this example. We will see later in Chapter 11 that this particular language is context-sensitive (see Example 11.1). We will also see that context-sensitive languages need pushdown automata with two separate stacks or another type of computing machine known as *linear-bounded automata*.

7.13 Key Ideas

1. Context-free behavior arises when an action is taken irrespective of the surrounding context. A context-free language is one in which the occurrence or behavior of a symbol or pattern in a string is independent of other symbols or patterns to its left or right.
2. A context-free language is the language of a context-free grammar.
3. A context-free grammar is one in which every production rule has a single non-terminal and no other symbol on its left-hand side.
4. A production in a context-free grammar can be applied during parsing or derivation to a non-terminal in the sentential form without regard to other symbols in the sentential form.
5. Context-free grammars are more powerful than regular grammars. They can derive languages that are not regular.
6. A linear context-free grammar has only zero or one non-terminal on the right-hand side of every production rule. However, such a grammar may not be right-linear or left-linear. Non-regular linear grammars can derive languages such as palindromes and the simple nesting language.
7. Other context-free languages require non-linear grammars with production rules that have more than one non-terminal on their right-hand sides. The typical pattern SS is used to generate languages such as proper nesting, nested if-then-else statements and arithmetic expressions in programming languages.
8. A context-free grammar is said to be ambiguous if it produces two different leftmost (or two different rightmost) derivations for the same string.
9. Often, an ambiguous grammar can be converted to an unambiguous grammar by introducing additional variables.
10. A language is said to be inherently ambiguous if every grammar for the language is ambiguous.
11. Grammars can be cleaned up by eliminating λ-productions, unit productions and useless (i.e., non-generating or unreachable) variables.
12. A context-free grammar can be converted to Chomsky Normal Form in which every production is in one of two forms: $A \to a$ or $A \to BC$. The idea is to ensure that the application of a production during derivation either eliminates a non-terminal by replacing it by a terminal symbol or grows the sentential form by exactly one symbol. This results in a binary parse tree.
13. The CYK algorithm can be applied to a grammar in Chomsky Normal Form to determine if a given string is grammatical, that is, belongs to the language of the grammar. Book-keeping in the CYK algorithm is made easy by applying a table-filling technique.
14. The idea of a Greibach Normal Form is to ensure that every step in parsing consumes exactly one terminal symbol. Productions in this form have exactly one terminal symbol followed by zero or more non-terminals on their right-hand sides.

15. Simple grammars and other variations are used to reduce ambiguity and make parsing more efficient. Such context-free grammars are used extensively in the design of programming languages and compilers.

16. Although there is no algorithm to design a context-free grammar, the set of mantras discussed in the chapter help us in designing an accurate grammar for a given problem.

7.14 Exercises

1. Consider the grammar $S \rightarrow 0S0 \mid 1S1 \mid SS \mid \lambda$. Given the string 0101101110, find a leftmost derivation and a rightmost derivation with corresponding parse trees.
2. For the above grammar, how many different parse trees can you construct for the string 00111100?
3. Consider the grammar $S \rightarrow 0B \mid 1A, A \rightarrow 0 \mid 0S \mid 1AA, B \rightarrow 1 \mid 1S \mid 0BB$. Given the string 1100, find a leftmost derivation and/or a rightmost derivation with corresponding parse trees. Repeat for the strings 001110 and 001101.
4. Given the CFG $S \rightarrow AB \mid \lambda, A \rightarrow aB, B \rightarrow Sb$, construct a derivation tree for *aaabbbbbb*.
5. Prove that there exists a RegEx for every GNF grammar that is also linear.
6. Consider the set of all strings that begin with a certain number of xs followed by a number of ys such that the number of xs is seven more than twice the number of ys. Is this a context-free language?
7. Show that $L = \{w \mid \text{length of } w = 3 \times \text{number of } a\text{s in } w\}$ is a context-free language. The alphabet is $\{a, b, c\}$.
8. Consider the simple nesting language $a^n b^n$ for matching parentheses. Given that in no real program the nesting of parentheses can be to an infinite depth, let us limit nesting to, say, a depth of 10. If all other symbols are ignored, the language becomes finite. Since we know that a finite language is regular, should we model this language as a regular language or is there still some advantage in modeling it using a context-free grammar? Justify your answer by constructing both models.
9. Repeat the above exercise for the proper nesting language.

A. Construct a context-free grammar for each of the following languages:

10. $a^{2n}b^n$
11. $a^n b^n c^m d^m$
12. $a^n b^m c^m d^n$
13. $a^n b^m c^k$, where $2n = m$ and $k \geq 2$.
14. $a^n b^m$, where $n = 2 + (m \bmod 3)$. Is this a regular language? If so, make sure that the grammar is regular.
15. All strings over $\{a, b\}$ with either equal numbers of a and b or twice as many bs as as.

16. $a^n w w^R b^n$, where w is any string in $(a + b)^*$. Show the derivation for the string *aaaabbbbabbb*. Is this grammar ambiguous? How can it derive the string *aaabbbbbbb*?
17. $uvwv^R$, where v is of length at least 1 and u and w are of length 2. The alphabet is $\{a, b\}$.
18. ww^R, where $w = (ab)^* + (ba)^*$ the alphabet being $\{a, b\}$.
19. wcw^R, where $w = (ab)^* + (ba)^*$ the alphabet being $\{a, b, c\}$.
20. All strings of length 5 over the alphabet $\{a, b, c\}$. Can you make sure that the grammar is in Greibach Normal Form?
21. Arithmetic expressions that contain identifiers defined by the RegEx $(a + b).(a + b + 0 + 1)^*$. Expressions support the + and × operators and properly nested parentheses. For the CFG constructed, show derivations and parse trees for the following strings:

    (a) $(a1+b1)\times b0$

    (b) $(ab0)+b1\times bb1$
22. Strings over $\{a, b\}$ with either more as than bs or less as than bs. Is this language regular or context-free? Explain.
23. Proper nesting of a combination of parentheses, square brackets and flower braces. For example,

    { [( ) ( )] { { [ ]) [ ] } }

B. Describe the language of the following context-free grammars as concisely as possible:

24. $S \rightarrow pA \mid qB \mid rC, A \rightarrow Sp, B \rightarrow Sq, C \rightarrow \lambda$
25. $S \rightarrow AB \mid \lambda, A \rightarrow aB, B \rightarrow Bb \mid b$
26. $S \rightarrow aS \mid bS \mid SS \mid \lambda$
27. $S \rightarrow aS \mid bS \mid SSS \mid \lambda$
28. $S \rightarrow aS \mid Sb \mid bS \mid Sa \mid SS \mid \lambda$
29. $S \rightarrow aA \mid Bb \mid \lambda, A \rightarrow Aa \mid a, B \rightarrow bB \mid b$
30. $S \rightarrow aAb \mid aBb \mid \lambda, A \rightarrow aBb \mid a, B \rightarrow aAb \mid b$
31. $S \rightarrow aSa \mid A, A \rightarrow bAb \mid \lambda$
32. $S \rightarrow aS \mid AB, A \rightarrow bA \mid B, B \rightarrow AA \mid \lambda$
33. $S \rightarrow a \mid bB \mid ccC, B \rightarrow bB \mid \lambda, C \rightarrow cC \mid \lambda, D \rightarrow d$
34. $S \rightarrow AaB \mid aaB, A \rightarrow \lambda, B \rightarrow bb \mid \lambda$
35. Consider the grammar called G_{mid}: $S \rightarrow aSa \mid \lambda$. Is G_{mid} right-linear or left-linear? Is the language a regular language? Convert the grammar to a regular grammar.
36. Consider $S \rightarrow aSb \mid \lambda$. Can this be converted to a regular grammar?

C. The following problems are concerned with ambiguity in grammars:

37. Show that $S \rightarrow 1S \mid 11S \mid \lambda$ is ambiguous.
38. Is the following grammar ambiguous? $S \rightarrow AB \mid aaB, A \rightarrow a \mid Aa, B \rightarrow b$

39. Show that $S \to SaS \mid b$ is ambiguous. Construct an unambiguous equivalent of the grammar.
40. Show that the union of $a^n b^n c^m$ and $a^n b^m c^m$ is inherently ambiguous.
41. Consider the language of the grammar: $S \to aaS \mid aaaS \mid \lambda$. Is the language inherently ambiguous? Can you construct an unambiguous grammar for it?
42. Consider the language of the grammar:

    $S \to 0A \mid 1A \mid SS \mid \lambda, A \to 0B \mid 1B \mid 00S \mid 01S \mid 10S \mid 11S, B \to 0S \mid 1S \mid 00A \mid 01A \mid 10A \mid 11A$

    Can you construct an unambiguous grammar for the same language?

D. Convert the following grammars to Chomsky Normal Form:

43. $S \to abS \mid baS \mid \lambda$
44. $A \to aB \mid aAA, A \to bB \mid bbC, B \to aaB \mid \lambda, C \to A$
45. $S \to BAB, B \to bba, A \to Bc$
46. G_{mid} from Exercise 35 above.
47. $S \to a \mid aA \mid B \mid C, A \to aB \mid \lambda, B \to Aa, C \to cCD, D \to add$
48. $S \to AS \mid AAAS, A \to SA \mid aa \mid b$
49. $S \to Aa \mid B \mid Ca, B \to aB \mid b, C \to Db \mid D, D \to E \mid d, E \to ab$
50. $S \to aAa \mid bBb \mid BB, A \to C, B \to S \mid A, C \to S \mid \lambda$
51. $S \to A \mid aB \mid \lambda, A \to aA \mid B, B \to bB \mid C, C \to c \mid A$

E. Convert the following grammars to Greibach Normal Form:

52. $S \to Sab \mid Sba \mid \lambda$
53. $S \to AB, A \to aA \mid bB \mid b, B \to b$
54. $S \to abSb \mid aa$
55. $S \to aSb \mid ab$

F. Apply the CYK-algorithm to verify if the given string(s) can be derived by the grammar:

56. $S \to 00S11 \mid 11S00 \mid \lambda$, $w = 00111100$
57. $S \to 0S1 \mid 1S0 \mid SS \mid \lambda$, $w = 00011110$
58. $S \to AB, A \to BB \mid a, B \to AB \mid b$, $w = aabbb, aabb, aabba, abbbb$
59. $S \to AB, A \to BB \mid a, B \to AB \mid b$, $w = aabbb$

G. Debug and correct the following context-free grammars:

60. For the language $a^n b^m$ where $m = 3n$:

    $S \to aS \mid B, B \to Sbbb \mid \lambda$

61. For two more as than bs (in any order):

    $S \to aaA \mid Aaa \mid \lambda, A \to aAb \mid bAa \mid AA \mid \lambda$

62. For $a^n b^m c^k$ where $k = n - m$ (the language of subtraction):

    $S \to aSb \mid bSc \mid \lambda$

63. For binary string in which every run of 1 s is of odd length:

    $S \to 0S \mid 1A \mid \lambda, A \to 1A \mid 0A \mid \lambda$

H. Open-ended exercises:

64. Write a context-free grammar for the structure of nested loops of various kinds in a modern programming language such as Java™.
65. Consider a modern programming language such as Java™ and find out what elements of it, if any, are not context-free (i.e., context-sensitive).
66. Write down a comprehensive set of production rules for the grammar of English considering most types of simple, assertive sentences (no questions, imperatives, etc.).
67. Consider HTML, HyperText Markup Language, the language of the Web. A valid HTML string has a set of matching start and end tags. Both are enclosed in angle brackets with the end tag having an extra symbol, the backslash at the beginning. For example, <P> is a start tag and </P> is the matching end tag. Tags must be properly nested with every start tag matching a corresponding end tag (although in reality most web browsers are rather forgiving and do not enforce this strictly). For example, a valid HTML string is

    <HTML><HEAD><TITLE></TITLE></HEAD>

    <BODY><P><UL><LI></LI><LI></LI></UL></P></BODY></HTML>

    ignoring of course the actual contents of a web page which appear in between start and end tags. Assuming that the only possible tags are {HTML, HEAD, BODY, TITLE, P, UL, LI}, construct a CFG for valid HTML tag strings.
68. Consider a slightly more accurate specification of HTML: the HTML tag must be the outermost and can occur only once. The HEAD tag must be the first tag inside the HTML tag and it must be followed by exactly one occurrence of the BODY tag. All other tags can only appear inside the BODY tag except the TITLE tag which may (or may not) appear once inside the HEAD tag. Modify your grammar to handle these requirements.
69. Consider a more complete version of HTML where {TABLE, TH, TR, TD} are the additional tags for handling tables, table-headers, table-rows and table-data (or the individual cells in a table). These tags have the additional constraints that TH can only appear inside a TABLE; TR can appear any number of times but only inside a TABLE; TD can only appear inside a TH or TR. Extend the CFG to handle the table tags. (Notice that in the previous exercise, there should have been a similar constraint: LI can only appear inside a UL and a few other similar list structures in HTML.)
70. Consider a further extension whereby the tables should have the same number of columns in each row, that is, the number of TDs in any TR cannot exceed the number of TDs in the TH of the table (while there being no limit on the number of columns in a table). Can your context-free grammar handle this? Explain.
71. Consider tags in XML, the eXtensible Markup Language. This is similar to HTML except that the tags themselves are user-extensible, that is, they no longer are limited to a fixed

number of tags from a given set. Users can make up any tag on their own. For example, a valid XML string is

<a><a><a>

where a and b are tag names. Assuming that tags are any alphabetical string of length less than 10 characters from the alphabet $\{a, b, c, \ldots, z\}$, can you construct a context-free grammar for all valid XML strings (ignoring once again any content that appears between start and end tags)?

Chapter Eight

Pushdown Automata

Learning Objectives

After completing this chapter, you will be able to:

- Learn how to add memory to a finite automaton.
- Learn the equivalence of stack memory and context-free behavior.
- Learn to construct a pushdown automaton for a context-free language.
- Learn to convert a context-free grammar to a pushdown automaton.
- Learn to convert a pushdown automaton to a context-free grammar.
- Understand why non-determinism is required in pushdown automata.

This chapter on computing machines for context-free languages corresponds to Chapter 2 for the class of regular languages. Having shown earlier why finite automata are inadequate for non-regular languages, we now add a memory element to the automaton in the form of a stack and show how the resulting pushdown automaton can handle context-free languages. The equivalence of pushdown automata to context-free grammars is shown through the methods of conversion between them. The important idea of why pushdown automata have to be non-deterministic while finite automata need not be is explained here. Suitable mantras are also presented for the construction of pushdown automata.

8.1 Machines for Context-Free Languages

Finite automata were unable to handle context-free languages (CFLs) primarily because they did not have sufficient memory to remember parts of the input. For example, in languages such as *simple nesting, proper nesting* or *nested if-then-else,* the automaton needs to count an unlimited number of symbols and remember the count to be able to match it to the count of symbols in later parts of the input string. In other languages such as *palindromes,* the automaton needs to remember all the symbols in the first part of the input so that they can be matched to symbols that appear later in the input string.

We saw in the last chapter that context-free grammars (CFGs) were able to handle such languages because they could generate terminal symbols simultaneously in two parts of the input string by placing

a variable in the middle of the right-hand side of a production rule (e.g., $S \rightarrow aSb$). How can we build computing machines which can behave similarly?

What is missing from finite automata is a memory unit for remembering or storing (significant characteristics of) input symbols. Equivalently, what is missing in regular expressions is the use of variables in defining and reusing patterns. And, in the case of regular grammars, the trouble is the restriction that they be linear and that the variable on the right-hand sides of their productions be rightmost (or leftmost) uniformly across the grammar. How can we overcome such limitations and obtain more powerful computing machines by adding memory to finite automata?

We are not right away going to add the kind of random-access memory (RAM) that we find in modern digital computers to our automata. The reason is such memory makes the automaton as powerful as possible (see Chapters 10, 11 and 12 for more precise explanations of the power and limitations of computing machines). In the process, we would miss out on important classes of formal languages and their grammars and behaviors. Intermediate classes such as context-free and context-sensitive languages are significant from both theoretical as well as practical aspects of the design and implementation of high-level programming languages and data representations.

8.2 Adding Memory: Why Stack Behavior?

What is the simplest kind of storage unit, or memory that will do the job of counting or remembering input symbols without making the automaton grow infinitely by adding more and more states? How about a place where the automaton can simply push one or more symbols? A symbol pushed there remains in place until the automaton decides to take it out. There is no need to process the symbols that are stored: no need to look for symbols stored below other symbols, no need to seek, traverse, reverse, sort, index or otherwise change the contents of the memory. All that the automaton needs to do is place a symbol; place more symbols on top of earlier symbols; and pick up the symbol that is currently on the top and match it to the current input symbol. In other words, the automaton needs a memory unit that behaves like a stack data structure.

A finite automaton with an additional stack which can store an unlimited number of symbols is called a *pushdown automaton* (PDA). It is called so because it can push input symbols one by one down the stack. Like in any stack data structure, it can also pop the symbol that is at the top. All other behaviors are the same as in finite automata that handle regular languages. In each transition that the automaton makes, it can pop the stack and push one symbol onto the stack. Moreover, the choice of transition itself can depend not only on the current state and the input symbol but also the current symbol at the top of the stack. Figure 8.1 shows the parts and behaviors of a PDA.

A pushdown automaton M has 7 elements:

1. M.alphabet: Denoted by Σ, the input alphabet is the set of symbols present in input strings.
2. M.stackAlphabet: Denoted by T, the stack alphabet is the set of symbols used for storing on the stack. This set may or may not be the same as the input alphabet.
3. M.states: Also denoted by Q, it is the finite set of all states in the automaton.
4. M.startState: Usually denoted by q_0, it is the start state of the automaton.
5. M.finalStates: Denoted by Q_F, it is the subset of M.states that are final or accepting states of the automaton.

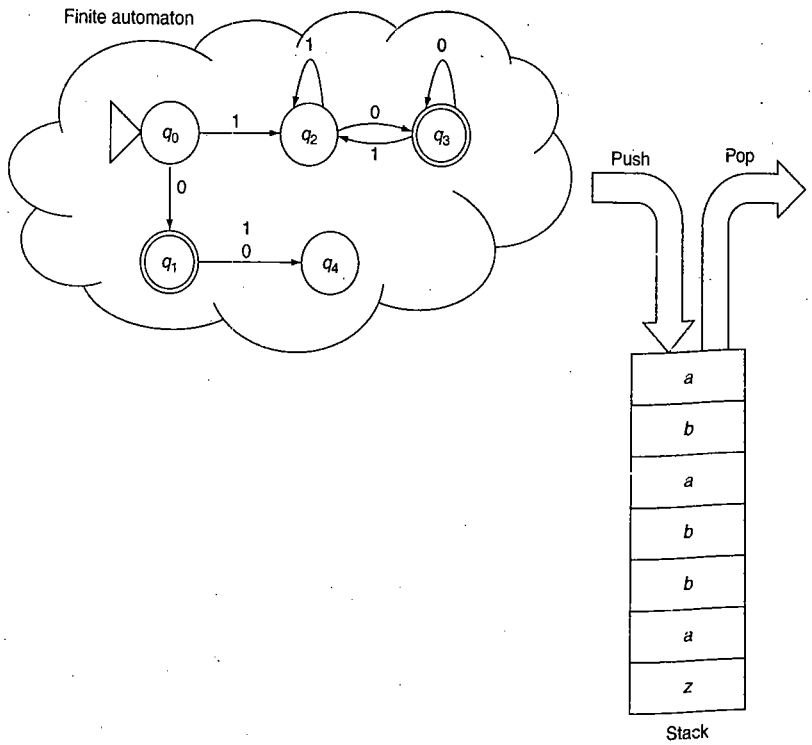


FIGURE 8.1 A pushdown automaton.

6. M.transitionFunction: This is a function δ from $Q \times \Sigma \times T$ to $Q \times (\text{POP}) \times (T)$, that is, a mapping from the current state, the current input symbol and the current symbol at the top of the stack, to a new state, an optional pop operation on the stack and a stack symbol that is optionally pushed onto the stack.
7. M.bottomOfStack: Usually denoted by z, it is a special symbol that is assumed to be always present at the bottom of the stack.

It is possible for the stack alphabet to be the same as the input alphabet. For convenience, we assume that the bottom of stack symbol z can never be popped off. The only operations that a PDA is allowed to perform are as follows:

1. Start with q_0 as the current state and z as the only symbol on the stack.
2. Examine the next input symbol reading it from left-to-right in a single pass.
3. Examine the current symbol at the top of the stack.
4. Based on the input symbol, the current state and the stack symbol, find one or more matching transitions.

5. Execute the transition by:
   (a) consuming the input symbol,
   (b) moving to the new state as specified by the transition function,
   (c) popping the symbol on the top of the stack if so specified by the transition function and
   (d) pushing the symbol specified in the transition (if any) onto the stack.

When the input symbols are exhausted, if the machine is in a final state and if the stack is empty (i.e., if the symbol at the top of the stack is z), then we say the machine has accepted the input string. Otherwise, the machine rejects the input string.

The set of all strings over the alphabet that a PDA accepts is called the language of the machine (denoted by M.language). We will demonstrate later in this chapter that the language of every PDA is a CFL and that pushdown automata (PDAs) correspond to CFGs.

You may have observed above that we talked about finding "one or more" matching transitions. When there is more than one matching transition, a PDA behaves non-deterministically, the same way a non-deterministic finite automaton (NFA) behaves (see Chapter 3). However, unlike in the case of non-deterministic finite automata (NFAs) which are equivalent to deterministic finite automata (DFAs), we will see that non-determinism is necessary to model CFLs! Deterministic PDAs cannot handle all CFLs.

The *snapshot* or *instantaneous description* or *configuration* of a PDA at a particular step during a computation can be captured by including the contents of its stack, the current state of the automaton, and the remaining part of the input string. For example the configuration

$$(q_3, abb, ABB)$$

means that the current state is q_3, input symbols remaining to be processed are abb and the current stack contents are ABB, A being the symbol on the top of the stack. Based on our convention, it is understood that the bottom-of-stack symbol z is present below these three stack symbols.

Why provide the machine with a stack of infinite size, especially after insisting the whole time that computing machines should have a finite set of states?

There are two important reasons why the stack should be able to hold an unlimited number of symbols although no practical implementation of stack memory can ever be of infinite size. The first reason is that we cannot impose an arbitrary limit on the size of the stack. That would exclude any string that required just a few more symbols to be stored, from being accepted by the machine. It would also make every language accepted by the machine a finite language and therefore a regular language. There would be no need to have a stack in the first place, at least when we are concerned only about theoretical models of computing machines. The second reason is that it is much more important for the set of states to be finite than the memory unit. Software programs that we write have to be finite and compact; we can always add more memory if needed to handle a given problem or language.

Why not some other kind of memory? Why not a queue instead of a stack?

A variety of such computing machines with different kinds of memory units have been tried and their properties analyzed and documented. However, from the point of view of studying classes of formal languages, grammars and computing machines in a systematic way, moving from the simplest class of regular languages to more and more powerful, larger and expressive languages, the stack is the most appropriate kind of memory unit. In fact, we will see later in this chapter that automata with stacks correspond exactly to the class of CFLs for which we studied the class of CFGs in the previous chapter.

8.3 Constructing PDAs

Let us construct a few simple PDAs to understand how the stack memory enables them to process languages that finite automata could not. We use a labeling notation on the transitions in the diagrams for the PDAs that involve triples. Each transition is a triple made up of the input symbol that is consumed, the symbol at the top of the stack before the transition and the symbol(s) at the top of the stack after the transition is performed. When a symbol is pushed onto the stack, it is shown along with the previous top of the stack. For example, the triple $(a, z ; 0z)$ means "on consuming the symbol a when the stack was previously empty (because z was the symbol on the top of the stack), the new stack contents are $0z$, that is, 0 on the top and z below it." Similarly, the triple $(a, 0 ; 00)$ means "consume a and push a second 0 onto the existing 0 on the stack" and $(b, 0 ; \lambda)$ means "consume b and pop a 0 from the stack." In this last one, the empty symbol λ on the top of stack denotes that the previous symbol was popped, but not that the symbol λ was actually pushed onto the stack or that the stack became empty; there could very well be other symbols below.

EXAMPLE 8.1

Our first PDA is for the language of *simple nesting* or $a^n b^n$ (see Example 7.2 in Chapter 7 for a CFG for this language). Figure 8.2 shows the PDA. In state q_0, the automaton consumes all the a symbols and counts them (or remembers them) by pushing a 0 onto the stack for every a. It changes state to q_1 upon seeing the first b where it consumes the remaining bs while at the same time popping off a 0 from the stack for every b. Thus the machine matches every b with a corresponding a which was seen before. At the end, when the input is empty and the stack is empty [denoted by the triple $(\lambda, z ; z)$], it reaches the final state q_2 where the input is accepted. As a special case when $n = 0$, that is, the input is the null string λ, the machine goes directly from q_0 to q_2 to accept the null input.

Unlike finite automata, this machine does not change states to remember input symbols. As you can see, it remains in the same state, looping back to it on a jolly ride while consuming symbols. However, the ride is no longer free; there is a toll to be paid every time the ride back to the same state is taken, the toll being a symbol to be pushed onto the stack or popped off the stack. The stack is used to count and remember symbols. States are changed only to remember what stage of processing the machine is at: in this example, in q_0, it is expecting more as; in q_1, it is expecting bs; and it reaches q_2 only if a valid string has already been completely processed.

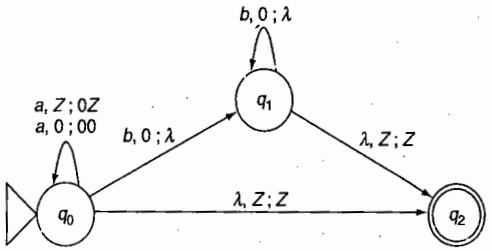


FIGURE 8.2 A pushdown automaton for the simple nesting language $a^n b^n$.

How does the PDA reject a string?

If the given input has more as than bs, the stack will not become empty and the machine gets stuck (or halts) in the non-final state q_1. If the string has more bs than as, then the machine again gets stuck in state q_1 because the stack is empty and there is no 0 on it to match to the extra b. In either case, it never reaches the final state q_2.

EXAMPLE 8.2

Our next example is for validating nested *if-else* statements. As we saw in Example 7.14 in Chapter 7, this language requires a non-linear CFG. A PDA for the language is shown in Fig. 8.3. In the start state q_0 itself, this machine consumes both *if* and *else* clauses, pushing and popping 0s. It changes state to reach the final state q_1 only when the input is empty, whether the stack is empty or not. This is because in this language, there is no need to have as many *else*s as *if*s. If we want the stack to become empty, we can simply add a loop or a jolly ride from the final state q_1 to itself just popping off all the 0s from the stack without needing any input symbol [i.e., a loop labeled $(\lambda, 0; \lambda)$].

This PDA rejects an input string only when it encounters more *else*s than *if*s at any point in the input string. In such a configuration, the stack will be empty and the machine will have no transition from q_0 to accept the extra *else*.

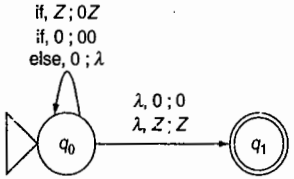


FIGURE 8.3 A pushdown automaton for nested *if-else* statements.

EXAMPLE 8.3

Let us consider even palindromes (a language which we saw earlier in Example 7.3 in Chapter 7), that is, strings whose second halves are the reverses of their corresponding first halves. Figure 8.4 shows a PDA that accepts even palindromes over the alphabet $\{a, b\}$. A CFG for this language could generate even palindromes by simultaneously generating corresponding symbols at either end of the string (with productions of the kind $S \rightarrow aSa \mid bSb$). The PDA, however, is required to process the string left-to-right in a single pass. As such, a difficulty arises in processing an input that is a potential even palindrome: where is the midpoint of the input? The automaton cannot look ahead till the end of the input to determine the midpoint. We will see later in Chapter 10 a kind of computing machine which can actually find the midpoint by looking ahead this way.

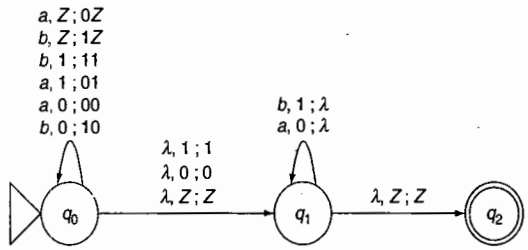


FIGURE 8.4 A pushdown automaton for even palindromes ww^R.

The PDA solves this problem by appealing to non-determinism the same way as NFAs (see Chapter 3). In its start state q_0, it consumes the first half of the input string, remembering all of it by pushing a 0 onto the stack for an a and a 1 for a b, no matter in which order they appear. It guesses the midpoint non-deterministically and jumps to state q_1 without consuming any input symbol and irrespective of what is at the top of the stack at that point. Once it is in state q_1, it processes the second half of the string by matching an incoming a to a 0 on the stack and a b to a 1, while popping off any matching symbol from the stack. When the input is finished, if the stack is empty, the PDA reaches its final state q_2 and accepts the string.

At any point in the second half of the string, if the input symbol does not match the symbol on the top of the stack, the PDA halts in the non-final state q_1 and rejects the string. Even if the non-deterministic guess was incorrect, the machine rejects the input for that particular guess. Of course, other guesses are executed in parallel and if any one of them is accepted, then the machine accepts the input.

Figure 8.5 shows five different configurations of the PDA after processing the prefix *aba* of the input string *abaaba*. As you can see, one of them, configuration (d) will lead to eventual acceptance of the input string.

Configuration (a) is obtained when the PDA incorrectly guesses that the input string is empty and jumps to state q_1 without processing any input symbol; it gets stuck in state q_1 and rejects the input. Configuration (b) results from incorrectly guessing, after processing the first symbol, that the midpoint was found. Similarly configuration (c) is for guessing that the length of the input string was four and thus the midpoint must be after two symbols. Configuration (d) is obtained with the correct guess that the midpoint was after three symbols. The first three symbols have been remembered by pushing onto the stack a 0, a 1 and another 0 for the input symbols *aba*. The PDA has transitioned to state q_1 in this configuration and will successfully match the remaining three symbols to the contents of the stack to accept the string. Finally, configuration (e) is reached when the PDA guesses that the input is still longer and therefore the midpoint is still further away; it has chosen to remain in state q_0 and will eventually reject the input string.

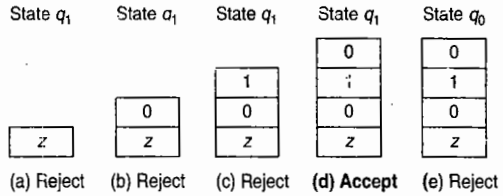


FIGURE 8.5 Non-deterministic configurations after processing *aba* in the input *abaaba*.

Table 8.1 shows the complete sequence of configurations that leads to acceptance of the given string.

TABLE 8.1 Accepting Configuration Sequence of PDA for Even Palindromes

Configuration	State	Input Consumed	Input Waiting	Stack Contents
1	q_0	λ	*abaaba*	z
2	q_0	*a*	*baaba*	0z
3	q_0	*ab*	*aaba*	10z
4	q_0	*aba*	*aba*	010z
5	q_1	*aba*	*aba*	010z
6	q_1	*abaa*	*ba*	10z
7	q_1	*abaab*	*a*	0z
8	q_1	*abaaba*	λ	z
9	q_2	*abaaba*	λ	z

The longer the even palindrome in the input, the more guesses and more configurations there will be in the non-deterministic PDA. There is no way to avoid this in the case of a PDA. We see at the end of this chapter that a deterministic PDA can handle odd palindromes that have a distinct symbol marking the midpoint (see Example 8.6), but non-determinism is essential for the present problem of even palindromes.

8.4 Constructing PDAs: *Mantras*

The mantras listed in Chapter 7 for constructing CFGs are also applicable to PDAs with the following modifications and additions:

1. *What is a context-free grammar for the language?*
   Constructing a grammar first is often helpful in understanding how the language behaves. From this, we can figure out how an equivalent PDA should behave. Sometimes, it is even advisable to directly convert a grammar to a PDA as outlined in the next section. However, the behaviors of such PDAs obtained by converting from a CFG are often not intuitive or easy to comprehend.
2. *What parts of strings in the language are correlated?*
   For example, the count of one symbol must match the count of another or the sequence of symbols in one part must match in reverse another sequence of symbols, and so on. Figuring out what parts are correlated gives us the desired behavior of the stack in the PDA being constructed. It tells us what symbols need to be pushed onto the stack and when they need to be popped off and matched to further symbols in the input string.
3. *When do we introduce a new state?*
   In general, PDAs do not need to change states to remember (the count of) an input symbol. PDAs change states only to remember what part of the input string is being processed. Introducing a new state in a PDA is sometimes equivalent to a regular expression introducing a concatenation in the case of regular languages. In general, a change in the behavior of the stack indicates a new state in the PDA. For example, symbols are pushed onto the stack in one state and when it is time to pop them off, the PDA moves to a new state.
4. *How is it different from known languages?*
   It is often useful to compare the given language to one of the known CFLs such as the languages of simple nesting, proper nesting, addition, if-then-else and arithmetic expressions. Differences, if any, can be accommodated by modifying states and transitions appropriately.
5. *Are any parts of strings in the language regular?*
   CFLs usually have some parts that are *regular* as well, that is, in addition to their correlated context-free behaviors, strings in the language contain parts that can be modeled using regular expressions. For example, one or more extra symbols (e.g., $a^n b^{n+1}$), some symbols in the middle (e.g., odd palindromes), and so on. The PDA has no use for its stack in modeling the *regular* parts of the language; they can be handled by states and state transitions. In fact, if we try to construct a PDA for a regular language, we get one which never uses its stack or, in other words, is just a finite automaton.
6. *Do we need non-determinism?*
   If the language is such that guessing is necessary to process it left-to-right in a single pass, then non-deterministic transitions can be introduced in the PDA. Non-deterministic transitions are multiple transitions from the same state with the same symbol on the top of the stack and the same input symbol (or an input symbol and the null symbol λ).
7. All the other mantras that we have been using from Chapter 2 are useful in constructing PDAs as well:
   (a) What is the simplest string in the language?
   (b) Is the null string a member of the language?
   (c) What are the patterns at the beginnings and ends of strings in the language?
   (d) What does the machine need to remember (through states and through the stack, in the case of a PDA)?
   (e) Do we have all the possible transitions from all the states for all the inputs symbols?
   (f) What is the complement of the language? What does the machine need to reject?
   (g) Do we need an explicit reject state?
   (h) Is the language the union of multiple sub-languages so that they can be handled in parallel branches?

8.5 Converting CFGs to PDAs

Given a CFG, a production rule of the form $T \rightarrow aU$ corresponds to a transition in a PDA where the input symbol is a, the previous top of the stack is T and the new top of the stack is U. The symbol T is popped off and the symbol U pushed onto the stack. If it is a terminal production of the form $T \rightarrow a$, then

the input symbol a is consumed and the stack is just popped. There is, thus, a correspondence between production rules in a CFG and transitions in a PDA. However, not all productions in a CFG are of the above two forms. What if there is more than one variable on the right-hand side?

For example, $T \rightarrow aUXYZ$? We can then extend the capabilities and conventions of a PDA by saying that more than one symbol can be pushed onto the stack: first Z, then Y, then X and finally U will be pushed onto the stack. It can also be shown that such PDAs are equivalent to regular PDAs.

Although not every production rule begins with a terminal symbol on its right-hand side, we do know from Chapter 7 that any CFG can be converted to Greibach Normal Form, in which this requirement is satisfied. This gives us a general method for constructing a PDA from a CFG:

1. Convert the given CFG to Greibach Normal Form.
2. From the start state q_0 of the PDA,
   (a) Add a transition to a new state q_1 with the label $(\lambda, Z; SZ)$.
3. For each production rule of the form $T \rightarrow aUXYZ\ldots$
   (a) Add a transition from q_1 to q_1 with the label $(a, T; UXYZ\ldots)$.
4. From state q_1, add a transition to a new final state q_2 with the label $(\lambda, Z; Z)$.

The PDA begins by artificially pushing the symbol S onto the stack, corresponding to a grammar starting with the variable S in its sentential form. From then on, without the need to change any state, the PDA adds transitions that correspond to each of the production rules in Greibach Normal Form. Finally, it adds a transition to the final state, the requirements for which are that the input should be fully consumed and the stack should be empty.

Why will such a PDA accept the same language as that of the grammar, especially since it does all its computation remaining in a single state? As we will see in the example below, every sequence of transitions that leads to an input string getting accepted by the PDA corresponds to a successful (leftmost) derivation of the string by the CFG.

The stack initially contains just the variable S. In any intermediate configuration, the stack contents are nothing but the variables in the sentential form of the corresponding derivation by the grammar. In each transition, we apply a production rule by expanding the top symbol on the stack and replacing it with the right-hand side of the production rule, except for the only terminal symbol which is matched to the input string. At the end, when all the variables are gone from the sentential form, the stack is empty and every input symbol has been matched. Let us work out an example to see how the method gives us a working PDA.

EXAMPLE 8.4

Consider once again the PDA constructed in Example 8.3 for even palindromes. This non-deterministic automaton performed its computation in two states: storing the first half of the palindrome in state q_0 and matching it to the second half in state q_1 (see Fig. 8.4). How can a single state PDA obtained by the above algorithm be equivalent to it?

Before we apply the algorithm to construct a PDA, let us understand the role of Greibach Normal Form in the correspondence between CFGs and PDAs. Our grammar for even palindromes which corresponds to the earlier PDA in Fig. 8.4 is

$$S \rightarrow aSa \mid bSb \mid \lambda$$

Converting this to Greibach Normal Form (as outlined in Chapter 7), we get

$$S \rightarrow aSA \mid bSB \mid \lambda$$
$$A \rightarrow a$$
$$B \rightarrow b$$

This grammar expands the start symbol S using one of its two non-null productions while matching a or b to the input symbols. In doing so, it keeps pushing variables A and B onto the stack, thereby making note of what symbols to expect in the second half of the input. Upon correctly guessing the midpoint of the input string non-deterministically, it erases the variable S by making it expand to λ. From then on, it keeps popping off A and B while matching with corresponding input symbols, by applying the productions for A and B, to finally end up with an empty stack when the input is exhausted.

Notice that unlike the original grammar which had only one variable S in any sentential form, the GNF grammar introduces many more occurrences of variables in its sentential form. In particular, each non-null production for S introduces two variables now, which can be SA or SB. The same behavior can be modeled in a PDA as shown in Fig. 8.6.

This PDA works along the lines of the grammar in Greibach Normal Form. It is very similar to the one in Fig. 8.4 in splitting its work between two states and jumping from one to the other non-deterministically at the midpoint of the palindrome. The only significant difference is that it also pushes the variable S onto the stack, so that each time, two variables are pushed onto the stack. Table 8.2 shows the complete sequence of configurations that leads this PDA to accept the example string *abaaba*.

Further, it turns out that we do not gain anything by splitting the handling of the palindrome into two states. Since the machine has the power of non-determinism, it is possible to combine the two states into a single state as shown in the new PDA in Fig. 8.7. This is exactly the PDA that we would have obtained if we had directly applied the above method for obtaining a PDA from a CFG to the grammar in Greibach Normal Form (except for the final state to be named q_3 instead of q_2 in order to be similar to the previous PDA in Fig. 8.6).

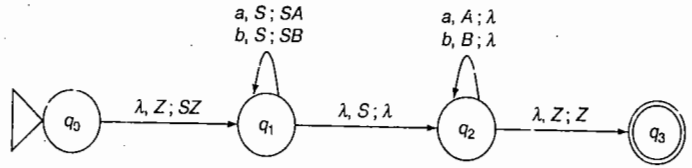


FIGURE 8.6 Another pushdown automaton for even palindromes ww^R.

TABLE 8.2 Accepting Configuration Sequence of PDA for Even Palindromes

Configuration	State	Input Consumed	Input Waiting	Stack Contents
1	q_0	λ	*abaaba*	z
2	q_1	λ	*abaaba*	Sz
3	q_1	*a*	*baaba*	SAz
4	q_1	*ab*	*aaba*	$SBAz$
5	q_1	*aba*	*aba*	$SABAz$
6	q_2	*aba*	*aba*	$ABAz$
7	q_2	*abaa*	*ba*	BAz
8	q_2	*abaab*	*a*	Az
9	q_2	*abaaba*	λ	z
10	q_3	*abaaba*	λ	z

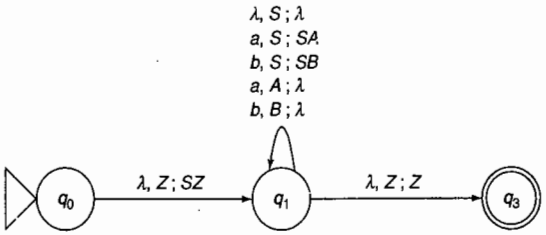


FIGURE 8.7 Yet another pushdown automaton for even palindromes ww^R.

How does this automaton know whether it is processing the first part or the second part of the palindrome in state q_1? The answer is: it need not know; it has the power to guess when to be processing the first part and when the second part, non-deterministically. If the variable S is present on the top of the stack, the PDA is still in the first part of the palindrome. The moment S is popped off using the production $S \rightarrow \lambda$, the PDA has no way of re-introducing it. It can only pop off the As and Bs present on the stack by matching them to corresponding input symbols using the production $A \rightarrow a$ and $B \rightarrow b$. All these transitions happen while the PDA continues to remain in state q_1. Table 8.3 shows the sequence of configurations that leads to acceptance of the given string *abaaba* (among many others that do not lead to acceptance).

TABLE 8.3 Accepting Configuration Sequence of PDA for Even Palindromes

Configuration	State	Input Consumed	Input Waiting	Stack Contents
1	q_0	λ	*abaaba*	z
2	q_1	λ	*abaaba*	Sz
3	q_1	*a*	*baaba*	SAz
4	q_1	*ab*	*aaba*	$SBAz$
5	q_1	*aba*	*aba*	$SABAz$
6	q_1	*aba*	*aba*	$ABAz$
7	q_1	*abaa*	*ba*	BAz
8	q_1	*abaab*	*a*	Az
9	q_1	*abaaba*	λ	z
10	q_3	*abaaba*	λ	z

8.6 Converting PDAs to CFGs

In converting a CFG to a PDA, we saw a correspondence between production rules in the grammar and transitions in the resulting PDA. A production rule of the form $T \rightarrow aUXYZ\ldots$ results in a transition in the PDA where the states do not matter much, but the symbol T must be on the top of the stack and the input symbol must be a for the transition to take place. The result of the transition is to pop T off the stack and push the sequence of symbols $UXYZ\ldots$, U being the topmost symbol on the stack. In other words, the left-hand side of a production rule maps to the symbol on the top of the stack; the terminal symbol on the right-hand side to the input symbol consumed in the transition; and the rest of the right-hand side to the symbols on or near the top of the stack.

Given an arbitrary PDA with several states, in which the states, in addition to its stack, are actually being used to remember things, it is not always convenient to reduce it to the kind of "single-state" PDA that we obtained above. The method for converting a PDA to a CFG must therefore somehow deal with state information in addition to just what is on the stack, in translating the transitions of the PDA to production rules in the CFG. We accomplish this by introducing a variable in the grammar for every combination of:

1. Any pair of states (between which there is a path or a sequence of transitions) and
2. The top of the stack symbol involved in the transitions of the automaton.

Each variable V_{1T2} in the new grammar represents all the (sub)strings that are consumed by the PDA in moving from a state q_1 to a state q_2 while also popping off a symbol T from the stack (along with any other symbols pushed onto the stack in moving from state q_1 to state q_2).

EXAMPLE 8.5

Let us consider as an example the PDA shown in Fig. 8.8 for the language of equal number of as and bs (in any order), that is, $n_a = n_b$. A CFG for this language was constructed in Example 7.12 in Chapter 7.

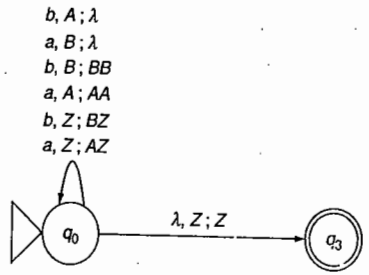


FIGURE 8.8 A PDA for equal numbers of a and b.

This PDA pushes any extra as or bs in the input onto the stack (in the form of variables A and B, respectively) and pops them off when matching symbols are found. In any configuration, the stack contains only As or only Bs; never both (because the number of as so far in the input can be higher than the number bs or vice versa, but never both at the same time). This can also be seen from the

numbering scheme shown in Example 7.12 in Chapter 7 where the number at any point in the input string can be zero or positive or negative but not both positive and negative at the same time.

Let us first work out the conversion of this PDA to a CFG intuitively and then outline the general method for the conversion. Consider the sequence of transition(s) from the start state q_0 to the final state q_3 which empties the stack. This is the set of strings accepted by the PDA. We denote this set of strings by the variable V_{0Z3} and later rename it to S, the start symbol of the grammar we are generating from the PDA. The middle symbol Z denotes the empty stack – the desired stack contents when the final state is reached. All other transitions are from q_0 to q_0 itself. The stack symbols involved are A and B. Thus, the remaining useful variables in the grammar will be: V_{0A0} and V_{0B0}.

What are the productions for these three variables? Consider V_{0A0}. What is the set of strings that the PDA processes in going from state q_0 to q_0 while popping off the symbol A from the top of the stack? These strings represent the right-hand sides of productions for the variable V_{0A0}. We have two such transitions where A is on the top of the stack: $(b, A; \lambda)$ and $(a, A; AA)$. The first one simply removes A, thereby giving us the production:

$$V_{0A0} \rightarrow b$$

The second one pops A but pushes two As back. What does it take to get rid of those two As from the stack? Two more occurrences of the variable V_{0A0} will do! Thus, we get the other production for the variable V_{0A0}:

$$V_{0A0} \rightarrow a\, V_{0A0}\, V_{0A0}$$

The productions for V_{0B0} are very similar. What about the sentence variable V_{0Z3}? There are two transitions in the PDA: $(a, Z; AZ)$ and $(b, Z; BZ)$. They introduce two variables each. What does it take to get rid of them from the stack? Reasoning on the same lines as for the other two variables, we get the productions:

$$V_{0Z3} \rightarrow a\, V_{0A0}\, V_{0Z3}$$
$$V_{0Z3} \rightarrow b\, V_{0B0}\, V_{0Z3}$$

We also have the special transition from the start state q_0 to the final state q_3 which results in the production:

$$V_{0Z3} \rightarrow \lambda$$

Thus, we get the complete grammar:

$$V_{0Z3} \rightarrow a\, V_{0A0}\, V_{0Z3}$$
$$V_{0Z3} \rightarrow b\, V_{0B0}\, V_{0Z3}$$
$$V_{0Z3} \rightarrow \lambda$$
$$V_{0A0} \rightarrow a\, V_{0A0}\, V_{0A0}$$
$$V_{0B0} \rightarrow b\, V_{0B0}\, V_{0B0}$$
$$V_{0A0} \rightarrow b$$
$$V_{0B0} \rightarrow a$$

Replacing V_{0Z3} by S, V_{0A0} by T and V_{0B0} by U, we get

$$S \rightarrow aTS \mid bUS \mid \lambda$$
$$T \rightarrow aTT \mid b$$
$$U \rightarrow bUU \mid a$$

This is the CFG equivalent to the given PDA. We notice that it is in fact in Greibach Normal Form. How does this grammar work? It says a valid sentence in the language can start with an a or a b or it can be altogether null:

$$S \rightarrow aTS \mid bUS \mid \lambda$$

If it starts with a, there must be a point in the string at which the number of as and bs become equal. This part is represented by the sentential form aT. From then on, more such sequences can appear. Therefore we say $S \rightarrow aTS$. Here T represents all strings with one extra b (so that aT has equal numbers).

What can T be?

It too can start with a or b. If it starts with b, it does not need anything else. Thus, we get $T \rightarrow b$. If it starts with a, we need two bs to balance the string now. Which variable can provide us one extra b? T itself. Thus, we get $T \rightarrow aTT$. Similarly we get the productions for U.

Table 8.4 shows the sequence of configurations of the PDA along with corresponding sentential forms in the grammar deriving the sample string *babbaaab*.

TABLE 8.4 Accepting Configuration Sequence of PDA for Equal as and bs

Configuration	State	Input Consumed	Input Waiting	Stack Contents	Sentential Form	Production Used
1	q_0	λ	*babbaaab*	z	S	
2	q_0	b	*abbaaab*	Bz	bUS	$S \rightarrow bUS$
3	q_0	*ba*	*bbaaab*	z	*baS*	$U \rightarrow a$
4	q_0	*bab*	*baaab*	Bz	*babUS*	$S \rightarrow bUS$
5	q_0	*babb*	*aaab*	BBz	*babbUUS*	$U \rightarrow bUU$
6	q_0	*babba*	*aab*	Bz	*babbaUS*	$U \rightarrow a$
7	q_0	*babbaa*	*ab*	z	*babbaaS*	$U \rightarrow a$
8	q_0	*babbaaa*	*a*	Az	*babbaaaTS*	$S \rightarrow aTS$
9	q_0	*babbaaab*	λ	z	*babbaaabS*	$T \rightarrow b$
10	q_3	*babbaaab*	λ	z	*babbaaab*	$S \rightarrow \lambda$

In general, it is not desirable to have to push more than two symbols onto the stack in a single transition. Before converting to a CFG, we need to convert the PDA to a form where every transition either pops the stack (i.e., shrinks the stack by exactly 1) or pops it and pushes two symbols onto it (i.e., grows the stack by exactly 1). Such a PDA results in a grammar in which every production rule is of one of the two following forms:

$T \rightarrow a$ Pops T off the stack (shrinks by 1)
$T \rightarrow aXY$ Pops T off the stack and pushes Y first and then X (grows by 1)

In a sense, this is a type of grammar that is a combination of the ideas of Chomsky Normal Form and Greibach Normal Form (see Chapter 7). The first production above is in both normal forms. The second one is in GNF but has a restriction similar to CNF that there can't be more than two variables on the right-hand side.

We now generalize from the above example and provide the method for converting a PDA to a CFG:

Algorithm

1. *Redesign the PDA so that every transition either pops one symbol from the stack (shrinks by 1) or replaces the symbol on the top of the stack with two symbols (i.e., grows the stack by 1).*
2. *Generate the list of variables by enumerating all the triples of a state, another state reachable from the state and a stack symbol.*
3. *Generate productions for the start symbol S as follows:*

$$S \rightarrow V_{0Zi}$$

for every state q_i including the start state q_0 itself.

4. *Generate productions for each transition (a, T; λ) from state q_i to q_j that shrinks the stack as follows:*

$$V_{iTj} \rightarrow a$$

5. *Generate the productions for each transition (a, T; XY) from state q_i to q_j that grows the stack as follows:*

   For all states k, for all states l,

$$V_{iTk} \rightarrow a\, V_{jXl}\, V_{lYk}$$

6. *Simplify the resulting grammar (if necessary).*

Production rules of the type $V_{iTj} \rightarrow a$ say that one way to take the PDA from state q_i where the top of the stack symbol was T to state q_j is to match it to the input symbol a (and pop T off the stack). The second kind of production rules, $V_{iTk} \rightarrow a\, V_{jXl}\, V_{lYk}$, say that one way to go from state q_i to any state q_k is to match to the input symbol a, pop off T, push XY onto the stack and figure out a way to get rid of X and Y. The PDA can pop off X by going from state q_j to some state q_l and then pop off Y by going from that state q_l to state q_k.

The above procedure usually generates large numbers of variables and productions many of which are useless. As seen in Chapter 7, useless productions are those that involve unreachable or non-generating variables. The grammar may also contain unit productions. Removing all such undesirable elements makes the grammar more readable. However, as you can see for example by applying the above method to the PDA in Fig. 8.8, grammars generated by the above method are rarely as compact, elegant or readable as the grammars that we write manually.

You may recall from Chapter 4 that converting a regular expression to an NFA was straightforward but writing regular expressions for finite automata was not easy. On the same lines, here we see that converting a CFG to a PDA is easier than conversion in the other direction from the machine to the grammar.

With the two methods for converting between PDAs and CFGs, we come to know that PDAs and CFGs are equivalent, that is, for every PDA there is an equivalent CFG and for every CFG, there is an equivalent (non-deterministic) PDA. It can be proved that any string generated by a Greibach Normal Form grammar for a language using a leftmost derivation is accepted by the equivalent PDA, thereby showing that the CFL is a subset of the language of the PDA. It can also be showed, in the other direction, that any string accepted by the PDA is in fact derived by the grammar, thereby establishing the full equivalence (see Appendix B).

What does the equivalence between PDAs and CFGs really mean? Why are they equivalent? How is the restriction of having only one variable (and no terminal symbol) on the left-hand side of every production rule in the CFG equivalent to the last-in-first-out (LIFO) behavior of the PDA's stack memory? Just as the stack does not allow the machine to see what is below the top of the stack, the CFG expands the variable on the left-hand side of a production without bothering about what other symbols are to the left or right of the variable in the sentential form. Thus, context-free and stack behaviors correspond to one another.

Before concluding this chapter, let us see why PDAs need to be non-deterministic.

8.7 Non-Determinism in PDAs

We saw in Chapter 3 that non-deterministic and deterministic finite automata are equivalent. However, it turns out that non-deterministic PDAs and deterministic PDAs are not equivalent! Why does it take non-determinism to deal with the power of CFGs? A deterministic PDA is one in which there is only one transition from a given state for every input symbol and every symbol on the top of the stack. The input symbol can be λ as a special case and, if so, there cannot be a transition from the same state and the same stack symbol for any other input symbol. In other words, there is never more than one choice for a transition in a given configuration of a deterministic PDA.

NFAs are equivalent to DFAs because they do not contain any memory element. The non-determinism (or multiple parallel execution paths) of an NFA can be modeled in an equivalent deterministic finite automaton (DFA) simply by increasing the number of states (to the power set of the set of states of the NFA in the worst case). The DFA has one state for every combination of states that the NFA could be in at any point in processing an input (see Chapter 3). Such a construction does not work in the case of pushdown automata because they have a memory unit in the form of a stack in addition to their states. If we try to capture all the different configurations in which a non-deterministic PDA could be at the same time (in its multiple parallel execution paths) in an equivalent deterministic PDA, there will be multiple configurations of the stack to be modeled. The stack in each parallel execution will have different contents from others. These multiple versions of stacks cannot be modeled in the single stack of a deterministic PDA.

Thus, deterministic PDAs are less powerful than non-deterministic PDAs. The languages of deterministic PDAs, known as *deterministic context-free languages* (DCFLs), are a proper subset of CFLs. We will study them in Chapter 11 when we look at a complete hierarchy of various classes of formal languages (see Sec. 11.12).

EXAMPLE 8.6

An example of a deterministic CFL is the language of odd palindromes, wcw^R, wherein the symbol c in the middle which makes its length odd is distinct from all the symbols in w. In comparison with the language of even palindromes (Example 8.3), a PDA for odd palindromes does not need to guess the midpoint of the input string and therefore does not need non-determinism. Figure 8.9 shows a deterministic PDA (DPDA) for the language of odd palindromes over $\{a, b\}$.

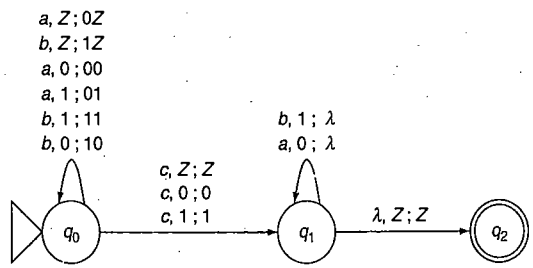


FIGURE 8.9 A deterministic PDA for odd palindromes wcw^R.

8.8 The CFL–CFG–PDA Triad

With the first family of concepts that we studied in this book, we started with machines called finite automata, defined their languages as regular languages, then learnt a notation called regular expressions and finally figured out how to write grammars for regular languages. In the second family, we started with context-free grammars, defined their languages as CFLs, and looked at a class of machines called pushdown automata. In simple terms, these PDAs are a bit peculiar. PDAs, while being much more powerful than finite automata, seem to have a use-and-throw kind of memory. If they use it, they lose it! If they recall (by popping the stack) what they remembered (by pushing onto the stack), they also erase the memory (i.e., the stack becomes empty). The machine cannot use it a second time to match with another part of the input. In the next chapter, we look at the limitation of PDAs and CFLs arising from this behavior to understand why not all formal languages are context-free. In particular, we will see that languages such as multiplication and ww are not context-free.

What is the difference between the two languages: ww^R and ww? It seems rather straightforward to construct a PDA for the first one but not the second. Why is this so? Of course, we can explain it away by saying that a stack only supports LIFO behavior and automatically reverses any string that is stored in it by successively pushing the symbols in the string onto the stack. This is why we are not able to do the matching in the case of the second language ww. Does it help to have a second stack? We will revisit this question in Chapter 11 (see Sec. 11.9 and Example 11.2).

8.9 Key Ideas

1. A memory unit is added to a finite automaton to overcome its limitations of not being able to count or remember a sequence of input symbols.
2. Adding a memory unit that behaves like a stack data structure to a finite automaton gives us a pushdown automaton.
3. The stack memory is supposed to be of unlimited size although the number of states in the pushdown automaton must still be finite. There is no reason to limit the memory size of a PDA and thereby eliminate some input strings from being accepted.
4. The language of a pushdown automaton is a context-free language.

5. Context-free languages require the PDA to be non-deterministic. A deterministic PDA cannot handle languages such as even palindromes that require guessing the midpoint of the input string non-deterministically.
6. Context-free grammars and pushdown automata are equivalent.
7. A grammar in Greibach Normal Form can be used to convert it to an equivalent pushdown automaton. The terminal symbol in the production rule matches the next input symbol and the non-terminal symbol on the left-hand side of the production is popped off the stack to be replaced by the non-terminals on the right-hand side.
8. A pushdown automaton can be converted to an equivalent context-free grammar although the resulting grammar is often unreadable.
9. Deterministic pushdown automata can only handle deterministic context-free languages which are a proper subset of context-free languages.
10. Although there is no algorithm to design a pushdown automaton, the set of mantras discussed in this chapter help us in designing an accurate PDA for a given problem.

8.10 Exercises

A. Construct a pushdown automaton (PDA) for the following languages.

1. Odd palindromes wcw^R over $\{a, b, c\}$ where $w = (a + b)^*$. Show an accepting sequence of configurations for the input *abbcbba*. Show how it rejects *abbcbb*. See Example 8.6.
2. Proper nesting of parentheses and flower brackets. For example, {(()){{()}{}}}. Show how it rejects {(){{()}}}.
3. $a^n b^n a^m b^m$, $n \geq 0$, $m \geq 0$. Show, along with two different accepting sequences of configurations, how non-determinism works to accept the string *aaabbb* in two different ways.
4. $a^n b^n ab$, $n > 0$. Make sure that the PDA is deterministic.
5. $a^n bab^n$, $n > 0$. Make sure that the PDA is deterministic.
6. $a^n b^m c^k$, where $2n = m$ and $k \geq 2$. Make sure that the PDA is deterministic.
7. $a^n b^m$, where $m = n$ mod 3. How much stack memory do you need to handle this language?
8. $a^n b^m$, where m is the nearest number equal to or higher than n that is divisible by 5. How much stack memory do you need to handle this language? How many states do you need in the PDA? Explain.
9. $a^n b^m$, where n is a multiple of 3 and m is $n/3$. Make sure that the PDA is deterministic.
10. $a^n b^m$, where $m = n \times 3$. Make sure that the PDA is deterministic.
11. The language of subtraction, that is, strings of the form $a^n b^m c^k$ where $k = n - m$ if $n \geq m$ or else $k = m - n$. Show an accepting sequence of configurations for the input *aabbbbcc*. Show the rejecting sequence of configurations for the input *aaaabbccc*.
12. The language of addition with positive or negative numbers, that is, strings over the alphabet $\{a, b, c, -\}$ of the form:
    (a) $a^n b^m c^k$ where $k = n + m$ and both numbers are positive; for example, *aabbcccc*.
    (b) $-a^n b^m c^k$, where $k = m - n$ if $m \geq n$ and the first number is negative; for example, *−aabbbbcc*.

(c) $-a^n b^m -c^k$, where $k = n - m$ if $n > m$ and the first number is negative; for example, *−aaaabb−cc*.

(d) $a^n -b^m c^k$, where $k = n - m$ if $n \geq m$ and the second number is negative; for example, *aaaa−bbcc*.

(e) $a^n -b^m -c^k$, where $k = m - n$ if $m > n$ and the second number is negative; for example, *aa−bbbb−cc*.

(f) $-a^n -b^m -c^k$, where $k = n + m$ and both numbers are negative; for example, *−aaa−bbb−cccccc*.

Show an accepting sequence of configurations for each of the example strings shown above.

13. $a^i b^j c^k$, where either $i = j$ (and k is any number) or k is the difference between i and j. Show an accepting sequence of configurations for the input *aabbbc*. Show how the PDA rejects both *aabbbbc* and *aaaabbbcc*.

14. $a^n b^m a^m b^n$. Can this be a deterministic PDA? Explain.

15. ww^R where each $w = (ab)^* + (ba)^*$. Unlike in the generic case where w can be anything, can this be a deterministic PDA? Explain. Can you also ensure that the number of stack cells used is just one-fourth the length of the input string?

B. Convert each of the following context-free grammars to an equivalent pushdown automaton.

16. $S \rightarrow aSA \mid \lambda, A \rightarrow bB, B \rightarrow b$

17. $S \rightarrow 0S1 \mid A, A \rightarrow 1A0 \mid S \mid \lambda$

18. $S \rightarrow aA, A \rightarrow aABC \mid bB \mid a, B \rightarrow b, C \rightarrow c$. Show how *aaabc* is accepted.

19. $S \rightarrow 0AA, A \rightarrow 0S \mid 1S \mid 0$

20. $S \rightarrow aA \mid bB \mid cC, A \rightarrow Sa, B \rightarrow Sb, C \rightarrow \lambda$

21. $S \rightarrow aA, A \rightarrow aA \mid bA \mid a \mid b$

C. Convert each of the following pushdown automata (PDAs) to an equivalent context-free grammar.

22. Consider the PDA shown in Fig. 8.10. What is the language of the PDA (and the resulting grammar)?

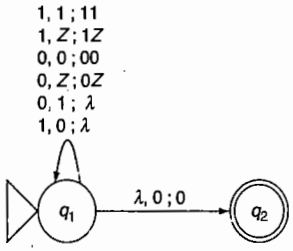


FIGURE 8.10

23. Consider the pushdown automaton shown in Fig. 8.11. What is its language?

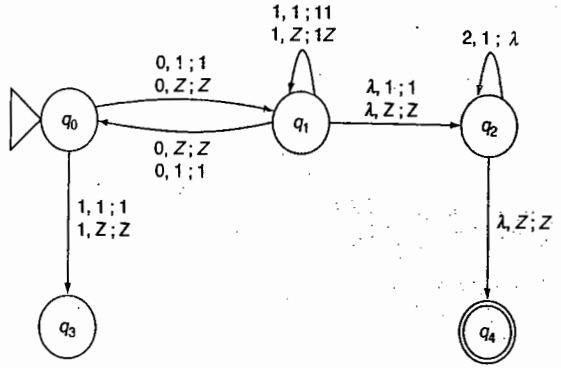


FIGURE 8.11

24. Consider the pushdown automaton shown in Fig. 8.9 (Example 8.6). Do you get back the same grammar as in the example?

25. Consider the pushdown automaton specified by the transitions shown below where q_2 is the accepting state:

$$\delta(q_0, a, Z) = (q_0, AZ)$$
$$\delta(q_0, a, A) = (q_0, AA)$$
$$\delta(q_0, b, Z) = (q_0, BZ)$$
$$\delta(q_0, b, B) = (q_0, BB)$$
$$\delta(q_0, a, B) = (q_1, \lambda)$$
$$\delta(q_0, b, A) = (q_1, \lambda)$$
$$\delta(q_1, a, B) = (q_1, \lambda)$$
$$\delta(q_1, b, A) = (q_1, \lambda)$$
$$\delta(q_1, \lambda, Z) = (q_2, Z)$$

D. Debug and fix the following pushdown automata (PDAs).

26. For $a^n b^{2n}$; $n > 0$ (Fig. 8.12):

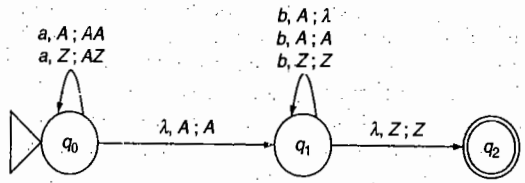


FIGURE 8.12

27. For simplified XML tag strings as used in Exercise 71 of Chapter 7, for example,

<a><a><a><c></c>

where a, b and c are the only tag names (Fig. 8.13):

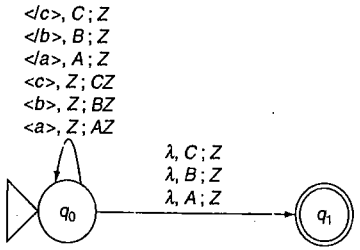


FIGURE 8.13

28. For the language of table tags in HTML {*TABLE*, *TH*, *TR*, *TD*} (as specified in Exercise 69 of Chapter 7), the pushdown automaton specified by the transitions shown below where q_1 is the accepting state:

$$\delta(q_0, \langle TABLE\rangle, Z) = (q_0, TZ)$$
$$\delta(q_0, \langle TH\rangle, \mathrm{T}) = (q_0, HT)$$
$$\delta(q_0, \langle TD\rangle, \mathrm{H}) = (q_0, DH)$$
$$\delta(q_0, \langle /TD\rangle, \mathrm{DH}) = (q_0, H)$$
$$\delta(q_0, \langle TR\rangle, \mathrm{T}) = (q_0, RT)$$
$$\delta(q_0, \langle /TR\rangle, R) = (q_0, \lambda)$$
$$\delta(q_0, \langle /TABLE\rangle, T) = (q_1, \lambda)$$

Chapter Nine

Nature of Context-Free Languages

Learning Objectives

After completing this chapter, you will be able to:

- Learn how to combine context-free languages to obtain more complex languages.
- Learn why context-free languages are not closed under intersection or complementation.
- Learn to answer questions such as emptiness and finiteness of context-free languages.
- Learn that equivalence or ambiguity questions of context-free languages cannot be answered!
- Learn why some languages are not context-free.
- Learn to apply the Pumping Lemma to show that a language is not context-free.

This chapter is similar to Chapter 6 and explores the nature of context-free languages. Unlike regular languages, this chapter shows how there are a few surprises in the behavior of context-free languages, thus setting the stage for the study of computability and decidability in the final chapters of the book. It also extends the Pumping Lemma to context-free languages and illustrates why certain languages are not context-free from the points of view of both grammars and pushdown automata.

9.1 Closure Properties

Context-free languages (CFLs) are not as well behaved as regular languages. To understand their behavior, just like in the case of regular languages, we first explore closure properties of CFLs. These properties are not merely of theoretical interest; they determine the practicality of using such languages whether in the design of programming languages or in data representation and processing.

Having studied context-free grammars (CFGs) and pushdown automata (PDAs) in the previous two chapters, we are now ready to summarize by defining a *context-free language* as a formal language for which we can construct

1. a non-deterministic pushdown automaton (PDA) or
2. a context-free grammar (CFG)

to accept, parse or generate all the strings in the language and no other string.

If we need to show that a particular set of strings is a CFL, all we need to do is to show that either a CFG or a PDA can be constructed for the set. We use this method below to show that the results of some operations on CFLs are also CFLs.

Regular languages remained regular when any set-theoretic operation was performed on them. Not all such *closure properties* are applicable to CFLs. Only the following closure properties apply to CFLs.

9.1.1 Union of CFLs

The set union of two context-free languages is a context-free language.

The result of the union of two CFLs L_1 and L_2 is a CFL because if G_1 and G_2 are CFGs for L_1 and L_2, that is,

$$G_1.\text{language} = L_1$$

and

$$G_2.\text{language} = L_2$$

with start symbols S_1 and S_2, respectively, then we can define a new context-free grammar G with a start symbol S by adding the production rule:

$$S \to S_1 \mid S_2$$

while retaining all other production rules of the two grammars (after re-naming any common variables to avoid conflicts, if any). This is a valid CFG and it generates any string that belongs to either language. Thus, the new grammar has as its language:

$$G.\text{language} = G_1.\text{language} \cup G_2.\text{language} = L_1 \cup L_2$$

For example, we know that the set of all strings over $\{a, b\}$ with equal numbers of a and b (i.e., $n_a = n_b$) is generated by the grammar:

$$S_1 \to aS_1b \mid bS_1a \mid S_1S_1 \mid \lambda$$

Similarly, the set of all strings over the same alphabet with twice as many as as bs is generated by

$$S_2 \to aaS_2b \mid abS_2a \mid baS_2a \mid aS_2ab \mid aS_2ba \mid bS_2aa \mid S_2S_2 \mid \lambda$$

By combining the two, we get a new grammar for strings which have either equal numbers or twice as many as as bs:

$$S \to S_1 \mid S_2$$

9.1.2 Concatenation of CFLs

The concatenation of two context-free languages is a context-free language.

The result of the concatenation of two CFLs L_1 and L_2 is also a CFL because if G_1 and G_2 are CFGs for L_1 and L_2, that is,

$$G_1.\text{language} = L_1$$

and

$$G_2.\text{language} = L_2$$

with start symbols S_1 and S_2, respectively, then we can define a new context-free grammar G with a start symbol S by adding the production rule:

$$S \to S_1S_2$$

while retaining all other production rules of the two grammars (after re-naming any common variables to avoid conflicts, if any). This is a valid CFG and it generates any string that is a concatenation of any string from L_1 with any string from L_2. Thus, the language of the new grammar G.language is the concatenation of G_1.language and G_2.language or $L_1.L_2$.

For example, the language $a^mb^mb^nc^n$, $m \geq 0$, $n \geq 0$, is a concatenation of two languages: one with equal numbers of a and b in that order followed by the other with equal numbers of b and c in that order. We can obtain a grammar for it as follows:

$$S \to S_1S_2$$
$$S_1 \to aS_1b \mid \lambda$$
$$S_2 \to bS_2c \mid \lambda$$

Note that it would be incorrect to further simplify this grammar by trying to combine the productions for the two languages although they are rather similar. For example, the following grammar is **incorrect** for the above language:

$$(S \to SS$$
$$S \to XSY \mid \lambda$$
$$X \to a \mid b$$
$$Y \to b \mid c)$$

9.1.3 * Closure of CFLs

*The * closure of a context-free language is a context-free language.*

The result of the * closure of a CFL is also a CFL because if G_1 is a CFG with G_1.language = L_1 and start symbols S_1, then we can define a new context-free grammar G with a start symbol S by adding the production rule:

$$S \to S_1 \mid SS_1 \mid \lambda$$

while retaining all other production rules of the grammar. This is a valid CFG and it generates all strings that are obtained by repeated concatenation of any string from L_1 zero or more times. Thus, the language of the new grammar, G.language is the * closure of G_1.language.

For example, given the grammar for the *simple nesting* language (see Example 7.2 in Chapter 7) a^nb^n, $n \geq 0$, we can obtain the grammar for a new kind of nesting language (which we may call *flat nesting*) as follows:

$$S \to S_1 \mid SS_1 \mid \lambda$$
$$S_1 \to aS_1b \mid \lambda$$

This language is not the same as the *proper nesting* language (see Example 7.11 in Chapter 7). For example, strings such as $a^3b^3a^4b^4$, or using parentheses ((()))(((()))), are present in both languages but a string such as $a^4b^3a^3b^4$, or using parentheses ((((()))((()))), is present in the proper nesting language but not in the present *flat nesting* language. This may also be seen from the fact that although the above grammar is non-linear, it does not have the production $S \rightarrow SS$.

Good behavior of CFLs ends here as far as closure properties are concerned. The following subsections show the closure properties which were true for regular language but are not obeyed by CFLs.

9.1.4 Complement of CFLs

*The complement of a context-free language may **not** be a context-free language.*

The complement of a language is the set of all strings that do not belong to the language. In other words, it is the set of all strings in Σ^* that are not in the given language.

To show that a statement is true for CFLs, we have to show that it is true for *every* CFL. To show that a statement is not true for all CFLs, that is, to prove a negative result, all we need to do is to show one negative example. If we show that the complement of one CFL is not context-free, then we will have shown that the above statement about the complement of a CFL not being a CFL is true.

An example of such a CFL is the subset of $a^*b^*c^*$ where either the number of as is not equal to the number of bs or the number of bs is not equal to the number of cs (i.e., $n_a \neq n_b$ or $n_b \neq n_c$). This is a CFL since we can write a CFG for it:

$S \rightarrow S_1 \mid S_2$	Either unequal a-b or unequal b-c
$S_1 \rightarrow S_3 \mid S_4 \mid S_1c \mid c$	At least one extra a or one extra b; any number of cs
$S_3 \rightarrow aS_3 \mid aS_3b \mid a$	At least one extra a
$S_4 \rightarrow S_4b \mid aS_4b \mid b$	At least one extra b
$S_2 \rightarrow S_5 \mid S_6 \mid aS_2 \mid a$	At least one extra b or one extra c; any number of as
$S_5 \rightarrow bS_5 \mid bS_5c \mid b$	At least one extra b
$S_6 \rightarrow S_6c \mid bS_6c \mid c$	At least one extra c

The complement of this language, however, is the language $a^nb^nc^n$ which is not a CFL. In fact, we will prove that this language is not context-free in Section 9.6 (see Example 9.1). Thus, we have an example of a CFL whose complement is not a CFL. This is sufficient to show that the complement of any CFL may not be a CFL. The relationships between the various sub-languages involved in this example are shown in Fig. 9.1.

9.1.5 Intersection of CFLs

The intersection of two context-free languages may not be a context-free language.

Again, we need just one counterexample to show that the statement is true. The same example that we used for complementation also works here.

Consider the two CFLs $L_1 = a^nb^nc^m$ with equal numbers of a and b and any number of cs and $L_2 = a^nb^mc^m$ with any number of as and equal numbers of b and c. They are both CFLs since we can easily construct CFGs or PDAs for each of them. Their intersection, the language where all three numbers are equal, is $a^nb^nc^n$ which is not a CFL. Hence there are CFLs whose intersections are not context-free.

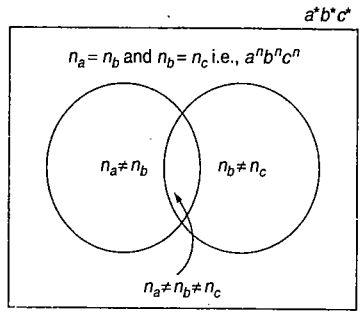


FIGURE 9.1 Complement of a context-free language may not be a context-free language.

An easier way to see this is through the set-theoretic relation between intersection and complementation: the intersection of two sets (or languages) is the complement of the union of the complements of the two sets:

$$L_1 \cap L_2 = (L_1.\text{Complement} \cup L_2.\text{Complement}).\text{Complement}$$

If CFLs were closed under intersection, they would have to be closed under complementation as well. We have already shown above that this is not true. In fact, L_1.Complement $\cup$ L_2.Complement is the language used in the example above for complementation.

However, *the intersection of a context-free language and a regular language is context-free.* This can be understood by designing a simulation wherein a PDA for the CFL and a DFA (deterministic finite automaton) for the regular language process the input in parallel and the input string is accepted only if both the PDA and the DFA accept it. Since the DFA has no stack or any other memory unit, there is no dependency from its processing on how the PDA manages its stack. The same argument does not work for two CFLs; we cannot run two PDAs in parallel to argue that the intersection of the languages of the two PDAs is context-free because of the conflicts between the ways in which the two PDAs manage the contents of their individual stacks.

We may ask why a smaller language, being the intersection of two CFLs, is not context-free? Should it not be easier to describe than larger languages that are themselves context-free? This line of thinking is not right. The largest possible language for any alphabet, namely Σ^*, is always a regular language! It is in describing some of its subsets in compact ways that we encounter problems. Moreover, since most of the languages that we consider are infinite languages, it does not make much sense to talk about them as smaller than others (except in ways introduced in Chapter 11).

9.1.6 Difference of CFLs

*The set difference of two context-free languages may **not** be a context-free language.*

Since the complement of a language is nothing but the set difference between the universal language Σ^* and the given language, and CFLs are not closed under complementation, they are also not closed under set difference. In fact, in many cases, the universal language considered such as Σ^* (or $a^*b^*c^*$ in the above examples) is a regular language. It follows, therefore, that even the difference between a regular language and a CFL need not be context-free.

9.2 Answering Questions about CFLs

Next, we address the following elementary questions about CFLs and look for methods for answering them:

1. *Is a given context-free language empty?* That is, given a representation of the language in the form of a context-free grammar, is there any string that is generated by it or is the language a null set?

   Method: Consider the status of the start symbol S in the grammar. If S is a useless symbol by being a non-generating symbol (see Chapter 7), then the language of the grammar is empty. If S generates at least one terminal string, then the language is not empty. The following CFG represents an empty language because any sentential form derived by the start symbol S in this grammar contains variables that can never be replaced by only terminal symbols:

   $$S \rightarrow aA \mid bB \mid C$$
   $$A \rightarrow aA \mid Ab \mid bB$$
   $$B \rightarrow bB \mid Ba \mid aA$$
   $$C \rightarrow bA \mid aB \mid aSb \mid bSa$$

2. *Is a given context-free language finite or infinite?*

   Method: Consider the given CFG for the language after eliminating any λ and unit productions (as outlined in Chapter 7). If the grammar contains a useful production of the form $A \rightarrow xAy$, where x and y are any sequences of terminal or non-terminal symbols, then the language is infinite (provided, of course, that A is not unreachable or non-generating). If there is no such production with the same variable on either side of the production, there may be a cycle of variables in the set of productions which enables the grammar to derive infinitely many strings. For example, the variable A may derive B, B derives C, C derives D, ... and, T derives A again. If none of these variables are useless (i.e., each is both reachable and generating), then the grammar represents an infinite language. If the graph of relationships of derivation among the variables in the grammar is an acyclic graph (i.e., there is no such circular dependency), then the language of the grammar is finite. For example, the following grammar generates an infinite language because of the cyclic derivations of A and B by S, A and B by A, and B and S by B:

   $$S \rightarrow aA \mid bB \mid c$$
   $$A \rightarrow Ab \mid bB$$
   $$B \rightarrow bB \mid aS$$

3. *Does a given string w belong to a* context-free *language L?* This is called the *membership question.*

   Method: To check the membership of a given string in the language of a given CFG, we can convert the grammar to Chomsky Normal Form and apply the CYK algorithm to determine whether the string is derivable from the grammar (see Chapter 7). Thus, CFLs can be used as programming or other data representation languages since we can determine whether or not a given string is grammatical (i.e., syntactically correct, parseable, interpretable, etc.).

9.3 Surprising Facts about Context-Free Languages

While it was possible to find methods for answering the above three questions, certain other questions about CFGs are impossible to answer! Such questions are said to be *undecidable* (see Chapter 12). For example, there is no algorithm to determine if a given CFG is ambiguous. This is an undecidable problem. Any formal procedure for this necessarily suffers from the *halting problem* for some inputs; it goes into an infinite loop and never decides one way or the other. Hence it is not considered an algorithm. It may decide for some inputs but will not be able to decide for some others. This is our first encounter with an *undecidable* problem, something that cannot be computed at all.

The following questions cannot be answered in the case of context-free languages!

1. *Are two context-free languages the same?*

   Given two CFGs, how do we know whether they represent the same language or not? This is an important practical problem in the design of programming and data representation languages. If two designers independently designed CFLs given the same set of requirements, it may be necessary to see if the two grammars they wrote are indeed the same. Surprisingly, there is no method for doing such a comparison. This, being an undecidable problem, will never have a solution. No algorithm, whether already invented or not, will ever be able to tell whether two arbitrary grammars given to it are equivalent. Incidentally, given two arbitrary programs, even if they are in the same programming language, there is no method to determine if they are equivalent (except in trivial cases where the programs handle only a finite set of inputs)!

2. *Is a given context-free language ambiguous?*

   As noted above, determining whether a given grammar is ambiguous is also an undecidable problem. This is again surprising because we have seen examples of how to convert an ambiguous grammar to an unambiguous one (see Chapter 7). Yet, there is no algorithm for determining whether a given grammar is ambiguous! That is, there is no formal procedure that is guaranteed to halt and say YES or NO for any grammar. Any method we may devise may work for some inputs, but for some other inputs, it will never halt; it necessarily goes into an infinite loop and never says YES or NO. There can never be an algorithm, program or implementation for such undecidable problems.

9.4 Why are Some Languages Not Context-Free?

We saw in Chapter 6 that some (infinite) languages were not regular because their strings did not have a simple repeating pattern. They violated the Pumping Lemma for regular languages. Computing machines needed unlimited amount of memory to process such languages. The machines for regular languages namely, finite automata did not have such memory and they could not process such languages given only a finite number of states. We added an unlimited amount of memory in the form of a stack to finite automata and we saw in the last chapter that the resulting PDAs could handle such CFLs. We will see below that languages such as $a^n b^n c^n$ and ww are not even CFLs, and, that they violate the *Pumping Lemma for context-free languages*. First, let us understand intuitively why they are not context-free.

From the point of view of PDAs, their stack allows them to count or remember only one set of symbols (or one pattern) at a time. If the count or pattern is used for matching with a later part of the input, it gets

popped off the stack and is no longer available for further use. This was the reason why we could not construct a PDA for a language such as $a^n b^n c^n$; it requires matching the same count twice, once with bs and then again with cs. Moreover, a string of symbols stored in a PDA's stack gets automatically reversed. This is why a PDA can handle the language of even palindromes ww^R, but not the language ww.

Another way to see why some languages are not context-free (or why they are *context-sensitive*) is through the kinds of patterns in sentential forms that CFGs can handle. Given their context-free nature, that is, the ability to expand a variable using any production for that variable without regard to other symbols that occur to the left or right of the variable in the sentential form, they are able to handle only simple correlations between two patterns in a sentential form. They can handle only mirror images of patterns such as what we see in palindromes, or, in simpler versions of it such as parenthesis matching or the language of addition. Of course, because of their ability to be non-linear (in terms of their grammars) and non-deterministic (in terms of their computing machines), context-free languages include such languages as proper nesting and even palindromes.

When a language involves more than two correlated patterns as in $a^n b^n c^n$ or two patterns that are not mirror images as in ww, CFGs and PDAs are unable to handle them. We will now state this property of CFLs formally in the form of a Pumping Lemma.

9.5 The Pumping Lemma for Context-Free Languages

The Pumping Lemma for CFLs is very similar to its counterpart for regular languages with the exception that strings in CFLs have not just one but two repeating patterns separated by other parts: a first part u, the first repeating pattern v, the separator x, the second repeating part y and the last part z, that is,

$$w = uvxyz$$

Both repeating patterns can be pumped up or down in equal numbers. Both cannot be null. If one is null, the language is regular (provided it is also not too far from the beginning). The two repeating patterns cannot be too far from each other (i.e., length of vxy is constrained to be less than or equal to some constant m for the given language). The *Pumping Lemma for CFLs* can be stated in plain English as follows:

> *Context-free languages are context-free because every non-trivial string in every infinite context-free language (that is not regular) has two correlated repeating patterns at least one of which is not empty and the two patterns are not too far from each other; these two patterns can be repeated in equal numbers any number of times to generate infinitely many strings in the language.*

The repeating patterns are matched to each other by using the stack to store or count symbols in the PDA for the language. Formally, the *Pumping Lemma for CFLs* says:

> *Every infinite context-free language L has a constant m, specific to that language, such that all strings w, $|w| \geq m$ belonging to L can be split into $w = uvxyz$, where $|vxy| \leq m$ and $|vy| \geq 1$ and for all $i = 0, 1, 2, \ldots$, the strings uv^ixy^iz belong to L.*

The constant m is the minimum size of strings for which the lemma applies. Strings shorter than that need not have any repeating patterns because there can only be a finite number of such short strings and an automaton can handle them with a finite number of states.

Any longer string w with its length $|w| \geq m$, according to the Pumping Lemma can be split into five parts – u, v, x, y and z – and this partitioning of the string must follow two rules:

1. The two repeating patterns in the middle v and y cannot both be empty or of length 0, that is,

$$|vy| \geq 1$$

2. The two repeating patterns v and y cannot be too far from each other, that is, the length of the three middle parts vxy cannot be greater than the constant m that is,

$$|vxy| \leq m$$

The non-repeating patterns u, x and z can be handled by a finite number of states. If repeating patterns exist but are infinitely far away from each other, it does not help our cause: the corresponding PDA would need an infinite number of states to handle the part of the string x between the two patterns.

Once we have the two repeating patterns that are not too far from each other, the Pumping Lemma says that we can *pump* the repeating patterns v and y (which are not null), that is, we can repeat them zero times ("pump them down") or any number of times ("pump them up") as long as we pump both of them the same number of times. We write this by saying, for all $i = 0, 1, 2, \ldots$ the strings uv^ixy^iz, where the repeating patterns v and y are repeated i times, belong to the context-free language L.

Just as in the case of regular languages, we do not use the Pumping Lemma to show that a given language is context-free; we have easier ways of doing that: we can construct a PDA or CFG for the language to prove that it is a CFL. Rather, we use the Pumping Lemma to show that a given language is *not* context-free.

We use a proof by contradiction to show that a given language is not context-free. We begin by assuming that the language is context-free. Therefore, the Pumping Lemma must be applicable to it. We apply it, but we cleverly choose a particular string w belonging to the language to set up a contradiction. The contradiction is shown by pumping v and y (down to $i = 0$ or up to $i = 2, 3, \ldots$) to create an imbalance in the string, that is, a mismatch between the repeating patterns in the string. We arrive at an uxz (for $i = 0$) or an $uvvxyyz$ (for $i = 2$) or an $uvvvxyyyz$ (for $i = 3$) which no longer belongs to the language. This is a contradiction. Either the Pumping Lemma is false or our assumption is false. The rest is mere argumentation for the sake of completeness to show that our assumption must have been wrong; that the given language must not have been context-free.

Before we apply such a proof by contradiction, we need to be sure that the Pumping Lemma is true for all CFLs. Consider a context-free language L and a context-free grammar that generates the language (i.e., the grammar derives every string in the language and no other string). The grammar has a finite number of non-terminals (and the constant m in fact has a linear relation to the number of non-terminals). Since the language is infinite, there must be strings in the language that are infinite in length. Let us consider a sufficiently long string in the language and its derivation, that is, the sequence of sentential forms starting from the start symbol S to the final sentential form which is the string itself.

For convenience, let us consider a leftmost derivation of the long string using a Greibach Normal Form grammar for the language (see Chapter 7). In such a derivation:

1. There is no limit on the length of the input string.
2. There is no limit on the depth of the derivation tree.
3. There is no limit on the length of the path from a leaf node to S in the derivation tree.
4. But there is a finite number of variables in the grammar.
5. Thus there must be at least one repeated variable along the path.

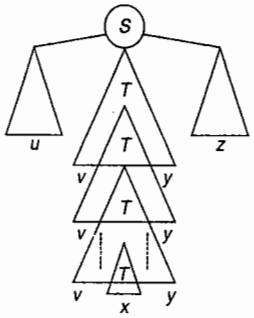


FIGURE 9.2 Collapsible sub-trees in parse trees of context-free languages.

The productions used at the different occurrences of the repeated variable can be interchanged according to the grammar to either collapse or pull out and expand the derivation tree. This is what is called pumping on i in the Pumping Lemma for CFLs. The resulting five parts of the yield of the tree are u, v, x, y and z (see Fig. 9.2):

1. u is the part derived by everything before the repeating variable T.
2. v is the part generated by T before repeating itself.
3. x is the part generated by T finally when it stops reappearing.
4. y is the part that is generated by T after repeating itself.
5. z is the part generated by everything after T.

Thus, if we examine the derivation or parse tree for the sentence, there must be a path in it from the root to a leaf node where at least one non-terminal T is repeated along the path (see Fig. 9.2). We can also apply the pigeonhole principle (see Chapter 6) to make this argument. With a sufficiently deep parse tree, there are more nodes along the path than the number of non-terminals in the grammar. Therefore, at least two nodes along the path must be labeled with the same non-terminal in the grammar.

The repeated node T generates vTy, that is, the grammar has a production of the form $T \rightarrow vTy \mid x$. As such, we can apply this production to expand T any number of times. In other words, the derivation of sentential forms exhibits the following pattern:

$$\begin{aligned} S &\Rightarrow uTz \\ &\Rightarrow uvTyz \\ &\Rightarrow uvxyz \end{aligned}$$

With the same set of productions, the following two derivations must also be possible:

$S \Rightarrow uTz \Rightarrow uxz$ Pumping down to $i = 0$ (T directly expands to x)

$S \Rightarrow uTz \Rightarrow uvTyz \Rightarrow uvvTyyz \Rightarrow uvvvTyyyz \Rightarrow \ldots \Rightarrow uv^iTy^iz \Rightarrow uv^ixy^iz$ Pumping up

Each time we pump up, one more pair of repeating patterns v and y is generated. Thus, the grammar can generate infinitely many strings in the CFL.

As noted in Chapter 6, even in the case of regular languages, there may be more than one repeated pattern, for example, $a(bc)^*(cb)^*$. However, there is one important difference in regular languages:

when there are multiple loops (or repeated patterns), the number of times the first loop is traversed is independent of the number of times any other loop may be traversed in generating a string, and so on for all the other loops. In CFLs though, there are indeed dependencies or correlations, such as, for example, by saying that there are two loops and both loops must be traversed an equal number of times. As already noted, if there are more than two correlated loops, that language is not even context-free; it is context-sensitive.

It may be noted that there is also a special case of the *Pumping Lemma for linear languages* wherein the length of the extreme parts, that is $uvyz$, is limited to m. In other words, the repeating patterns in linear languages cannot be too far from either end of the string.

9.6 Using the Pumping Lemma for Context-Free Languages

We use the lemma to prove that certain languages are not context-free by applying the Pumping Lemma to generate a string that is not in the language, that is, by setting up a contradiction. It may be noted that the application of Pumping Lemma to show that certain languages are not context-free sometimes involves clever mathematical jugglery to show that the resulting count of some symbol in terms of m does not match the pattern required by the language. Let us now apply the Pumping Lemma in a proof by contradiction to show that a given language is not context-free.

EXAMPLE 9.1

We have seen previously that we cannot construct a CFG or PDA for a language such as $a^nb^nc^n$: a certain number of a s followed by an equal number of b s and then an equal number of c s. This formal language can be seen as an extension of the *simple nesting* language of matching nested parentheses in programming languages. We will now use the Pumping Lemma to prove that the extended language of simple nesting with three parts of equal length is not context-free.

Since this is a proof by contradiction, we assume that the opposite of what we want to prove (i.e., the *contra-diction*) is true. In this case, we assume that the language $a^nb^nc^n$ is context-free. If it is context-free, the Pumping Lemma must be applicable to it. Accordingly, if we consider any string w that is sufficiently long, that is, $|w| \geq m$ for some unknown constant m, it must be possible to split the string into five parts $u.v.x.y.z$ while satisfying the remaining requirements of the Pumping Lemma.

Our objective is to arrive at a contradiction, that is, to find some value for i such that when the repeating patterns v and y in the string w are both pumped i times, the resulting string no longer belongs to the given language. The clever part of this particular proof by contradiction is in choosing the right w. In this example we choose

$$w = a^mb^mc^m$$

(i.e., $n = m$), where m is the Pumping Lemma constant for the given language (whose value we do not know). The string we have chosen is thrice as long as the minimum length for which the Pumping Lemma is applicable. In fact, it is crucial that its first m symbols are all just a s and that the first and last parts are separated by m b s.

What can the Pumping Lemma tell us about the repeating patterns v and y in this language apart from the fact that they both cannot be empty?

We know that vxy can be at most m symbols long, that is,

$$|vxy| \leq m$$

There are several cases possible (see Fig. 9.3):

Case 1: vxy is made up only of as.
v and y together must contain at least one a. Pumping them up or down will change the number of as without changing the number of bs or cs. Thus, the resulting string with more as (or less as) than bs and cs does not belong to the language.

Case 2: vxy is made up of some as and some bs.
v and y cannot contain any c. Pumping them up or down will change the number of as and/or bs without changing the number of cs. Thus, the resulting string with more (or less) as and/or bs than cs does not belong to the language.

Case 3: vxy is made up only of bs.
v and y together must contain at least one b. Pumping them up or down will change the number of bs without changing the number of as or cs. Thus, the resulting string with more bs (or less bs) than as and cs does not belong to the language.

Case 4: vxy is made up of some bs and some cs.
v and y cannot contain any a. Pumping them up or down will change the number of bs and/or cs without changing the number of as. Thus, the resulting string with more (or less) bs and/or cs than as does not belong to the language.

Case 5: vxy is made up only of cs.
v and y together must contain at least one c. Pumping them up or down will change the number of cs without changing the number of as or bs. Thus, the resulting string with more cs (or less cs) than as and bs does not belong to the language.

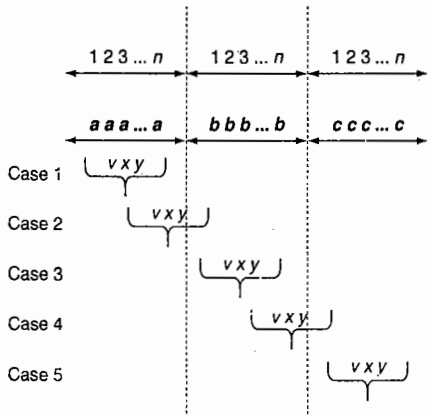


FIGURE 9.3 Span of repeating patterns in Pumping Lemma for context-free languages.

In any case, the string with the repeating patterns vxy, according to the Pumping Lemma, cannot be longer than m. As a result, it is not long enough to span all the three symbols – as, bs and cs – in the chosen string w. Pumping the string generates non-members of the language in every case, thereby contradicting our assumption that the language is context-free.

Table 9.1 summarizes the five cases analyzed in applying the Pumping Lemma to this language (where regular expressions are used to show what patterns can occur in vxy). We will see in Chapter 11 how we can construct a more powerful kind of grammar known as *context-sensitive grammar* for this language.

If we had chosen a string that was inconvenient for us, we would not have been able to set up a contradiction. For example, if we had chosen

$$w = a^k b^k c^k$$

where $k = 1 + (m/3)$, then upon partitioning w into $uvxyz$, the pattern vxy could cross over all three parts. In such a case, v could be a and y could be bc for example. Pumping them up or down would still keep the numbers of as, bs and cs equal. There would be no contradiction. This, however, does not mean that the language is context-free; rather, it means that our choice of w was not good enough to prove that it is not context-free.

TABLE 9.1 Various Cases in Partitioning $uvxyz$ and Pumping the String $a^n b^n c^n$

Case	vxy	v and y	i	n_a	n_b	n_c	Remarks
1	Only as	aa^*	≥ 2	↑	=	=	Too many as
2	as and bs	aa^*bb^*	≥ 2	↑	↑	=	Too few cs
3	Only bs	bb^*	≥ 2	=	↑	=	Too many bs
4	bs and cs	bb^*cc^*	≥ 2	=	↑	↑	Too few as
5	Only cs	cc^*	≥ 2	=	=	↑	Too many cs

Just as in the case of the Pumping Lemma for regular languages (Chapter 6), proving that a language is not context-free using the Pumping Lemma can be viewed as an *adversarial game*. Tips presented there for choosing the right string w are applicable to CFLs as well.

EXAMPLE 9.2

Consider the language ww of all strings over the alphabet $\{a, b\}$ where the second half is the same as the first half (as noted at the end of Chapter 8, we cannot construct a PDA for this language; see also Example 6.4 in Chapter 6). Let us assume that this language is context-free and choose a string $w = a^m b a^m b$ to apply the Pumping Lemma. The various cases possible and the corresponding strings obtained after pumping up to $i = 2$ are shown in Table 9.2. As you can see from the table, in every case, a contradiction is generated, thereby, concluding that the language is not context-free.

TABLE 9.2 Various Cases in Partitioning *uvxyz* and Pumping the String *ww*

Case	*vxy*	*v* and *y*	I	First *w*		Second *w*		Remarks
				n_a	n_b	n_a	n_b	
1	Only *a*s	aa^*	≥2	↑	=	=	=	Too many *a*s in first *w*
2	*a*s and *b*	aa^*b	≥2	↑	↑	=	=	First *w* too long
3	Only *b*	b	≥2	=	↑	=	=	Too many *b*s in first *w*
4	*b* and *a*s	baa^*	≥2	=	↑	↑	=	Too many *b*s in first *w* or too many *a*s in second *w*
5	Only *a*s	aa^*	≥2	=	=	↑	=	Too many *a*s in second *w*
6	*a*s and *b*	aa^*b	≥2	=	=	↑	↑	Second *w* too long
7	Only *b*	b	≥2	=	=	=	↑	Too many *b*s in second *w*

EXAMPLE 9.3

Let us show that the formal language of *multiplication is not a context-free language.* The language of multiplication of positive integers is the set of all strings which have three numbers in them and the third number is the product of the first two numbers. Just as in the case of the language of addition (see Example 6.5 in Chapter 6 and Example 7.10 in Chapter 7), we use the simplest possible number system for representing the language, namely, the *unary number system.* Recall that there is only one symbol {1} in this system. Any number is represented by a sequence of as many 1 s. As before, we use three different symbols {*a*, *b*, *c*}, one for each of the three numbers. With this notation, the language of multiplication is the set of all strings of the form $a^pb^qc^r$, that is, p number of *a*s followed q number of *b*s foilowed by r number of *c*s where $r = p \times q$. If in any string $r \neq p \times q$, then it does not belong to the language of multiplication.

We assume that the language of multiplication is context-free and choose

$$w = a^m b^m c^{m \times m}$$

When we pump this up to $i = 2$, we see that the resulting string no longer belongs to the language because, in cases 1, 2 and 3 (see Table 9.3), the new value of the first exponent p or the second exponent q is too high to match the product, that is, the number of *c*s which do not change. In case 5, the two factors p and q remain unchanged and only the product increases, once again resulting in an incorrect value for the product of the two factors. Case 4 is rather interesting since both the factor q and the product increase.

Can they increase in such a way that the product is indeed correct?

Not quite. Here is why: Let *vy* contain j *b*s and k *c*s. For $i \geq 2$, the string will continue to belong to the language only when

$m \times (m + j) = (m \times m) + k$ $\quad n_b$ increases by j; n_c by k

$m^2 + m \times j = m^2 + k$ $\quad n_a \times n_b$ increases by $m \times j$; n_c by k

$m \times j = k$

But this is impossible since

$$|vxy| \leq m$$

Thus

$$(j + k) \leq m$$

which implies

$$k < m$$

Table 9.3 summarizes the various cases analyzed in showing that the language of multiplication is not context-free.

TABLE 9.3 Various Cases in Partitioning *uvxyz* and Pumping the String $a^mb^mc^{m \times m}$

Case	*vxy*	*v* and *y*	*i*	n_a	n_b	n_c	Remarks
1	Only *a*'s	aa^*	≥2	↑	=	=	Wrong product
2	*a*s and *b*s	aa^*bb^*	≥2	↑	↑	=	Wrong product
3	Only *b*s	bb^*	≥2	=	↑	=	Wrong product
4	*b*s and *c*s	bb^*cc^*	≥2	=	↑	↑	Wrong product
5	Only *c*s	cc^*	≥2	=	=	↑	Wrong product

This completes our study of two families of languages, grammars and machines: regular and context-free. In the remaining three chapters, we study the largest classes of formal languages and grammars along with the most powerful kinds of computing machines. We also look at limitations of even the most powerful class of computing machines in handling certain kinds of problems and languages.

9.7 Key Ideas

1. Combining context-free languages is not as simple as combining regular languages. Context-free languages are well-behaved only under the operations of union, concatenation and * closure.
2. The complement of a context-free language or the intersection or set difference of two context-free languages may or may not be context-free.
3. A context-free language is empty if in any equivalent grammar, the start symbol S is non-generating.
4. A context-free language is finite if its grammar has no recursive production rule or cycle among any of its non-terminals.
5. Most other questions about context-free languages cannot be answered. Whether two context-free grammars have the same language or whether a given context-free language is ambiguous are undecidable questions. There can never be an algorithm to answer such questions about context-free languages!
6. Every regular language is context-free.

7. There are languages that are not context-free.
8. A PDA can count or remember one sequence of symbols and match it with another sequence. However, in the process of matching, the PDA forgets the count by emptying its stack. If the language requires matching with a third sequence, for example, such a language is not context-free.
9. All infinite context-free languages have a pair of correlated repeating patterns that are not too far from each other.
10. The repeating patterns can together be repeated any number of times to generate an infinite set of strings each of which must belong to the context-free language. This is called the Pumping Lemma.
11. The Pumping Lemma is used only to show that a given language is not context-free in a proof by contradiction.
12. The Pumping Lemma is used in an adversarial game where the opponent chooses an unknown value for the constant m and an unknown partition of the string w into $uvxyz$. We can choose the string w and the value of i to our convenience.
13. Application of the Pumping Lemma works like an adversarial game because of alternating universal and existential quantifiers. We are free to choose any value when a variable is universally quantified but must ensure that our arguments work for any unknown value when under an existential quantifier.
14. The key to a successful application of the Pumping Lemma is to choose a string that includes the constant m and takes full control over the crucial sub-string of length m.
15. Context-free grammars can generate strings with a pair of coordinated repeating patterns only if the two patterns mirror each other at opposite ends of strings. They are unable to deal with either more than two patterns, for example, $a^nb^nc^n$, or with kinds of patterns that do not correspond to the behavior of a stack, for example, ww.

9.8 Exercises

1. Prove that the set of all regular languages is a proper subset of the set of all context-free languages.
2. We have seen that elements of the syntax of high-level programming languages such as arithmetic expressions, nested parentheses and nested if-then-else statements are all context-free languages. Assuming that programs written in the language are made up of a sequence of such expressions and statements each of which is context-free, show that the set of all such programs is also context-free.
3. Show that the set of all binary strings which are palindromes and, when interpreted as positive integers, are divisible by 3, is context-free.
4. Give an example of a context-free language whose complement is not context-free (other than the example given in Sec. 9.1.4).
5. Show that the set of all strings over $\{a, b, c\}$ that are either even palindromes or odd palindromes is context-free.
6. Can we show that the set of all strings over $\{a, b, c\}$ that are neither even palindromes nor odd palindromes is context-free? Explain.

7. If a programming language is context-free, every program that can be written in the language can be generated from the grammar of the language. Can we show that the set of all invalid programs, that is, those with syntax errors, is not context-free? Explain.
8. XHTML is a version of HTML that meets the stricter requirements of XML (e.g., case-sensitivity, proper nesting of tags, etc.). Assuming that both HTML and XML are context-free languages, can we say that XHTML is a context-free language?
9. We know that the set of strings $w = (a + b)^*$ is a regular language and therefore also a context-free language. We also know that the concatenation of two context-free languages is a context-free language. Consider the set of concatenations $w.w$. We showed in Example 9.2 that this language is not context-free. Is this a contradiction? Is this the language of the following grammar? Explain.

$$S \rightarrow AA, A \rightarrow aA \mid bA \mid \lambda$$

10. What if in $w.w$, $w = (ab)^* + (ba)^*$? Is this language context-free or not?
11. Is there a deterministic language that is not context-free?
12. Consider the PDA shown in Fig. 9.4 which accepts all strings over $\{a, b, c\}$ that are *not* odd palindromes (with c occurring only as the separator). Since we could construct a PDA for it, this language must be context-free. However, its complement – the set of odd palindromes – is also context-free since we can easily construct a PDA or a CFG for it. We also know that the complement of any context-free language may not be context-free. Is there a contradiction here? Explain.

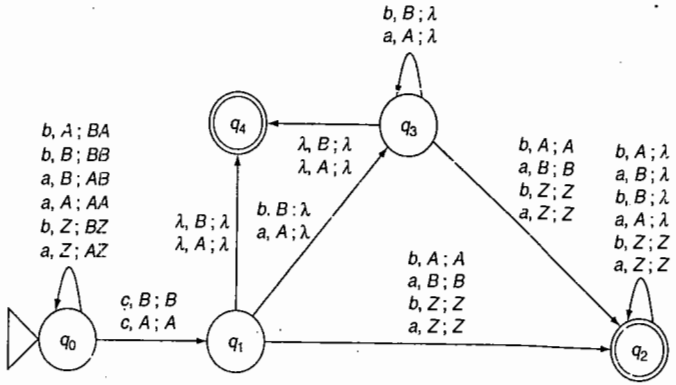


FIGURE 9.4 PDA for all strings that are not odd-palindromes.

13. Are the languages of the following two grammars (with start symbols S_1 and S_2) the same? How can we be sure?

$$S_1 \rightarrow aS_1b \mid \lambda$$

$$S_2 \rightarrow ATB \mid \lambda, T \rightarrow CTD \mid S_2, C \rightarrow A, D \rightarrow B, A \rightarrow a, B \rightarrow b$$

14. Show that $a^nb^mc^m$, where $n \neq m$, is not a context-free language.
15. Show that $a^nb^mc^m$, where $n = 2m$, is not a context-free language. Why can't we construct a PDA that pushes a symbol onto the stack for each a and pops one symbol for each b or c?

16. Show that $a^n b^m c^k$, where $n < m < k$, is not a context-free language.
17. Show that $a^n b^m c^k$, where $k = m^n$, is not a context-free language.
18. Show that division is not a context-free language.
19. Is the set of all strings of the form $a^n b^m c^k$, where $k = m \times n$ and $k < 1000$, $m \geq 0$, $n \geq 0$, a context-free language or not?
20. Show that the set of all strings of the form 1^p, where p is a prime number (i.e., prime numbers represented in the unary number system), is not context-free.
21. There is a stronger version of the Pumping Lemma for context-free languages known as *Ogden's Lemma* which states that we need not consider the entire string w but just the symbols at any m (or more) chosen positions in the string w. The string w is partitioned into $uvxyz$ as in the Pumping Lemma. The rest of the formulation and application of the lemma are also similar to that of the Pumping Lemma except that we now say that the two repeating patterns, that is vy, must contain at least one of the chosen symbols. Ogden's Lemma can be used to show that certain languages are not context-free even though the Pumping Lemma fails to set up a contradiction for such languages. One such language is $0^n 1^n 2^k$, where $n \neq k$. Show using Ogden's Lemma that this language is not context-free.

Chapter Ten

Turing Machines

Learning Objectives

After completing this chapter, you will be able to:

- Learn how a simple computing machine can compute anything that is computable.
- Learn to construct simple Turing machines for languages and computable functions.
- Learn techniques for constructing more complex Turing machines.
- Understand how variations of Turing machines are all equivalent.
- Learn how to compile a Turing machine into a binary string.
- Understand how a universal Turing machine works as a stored-program computer.

The idea of a universal computing machine is introduced here, beginning with a brief historical context of the development of mathematics and philosophy that led to the birth of computer science and the invention of the Turing machine as a computing device. The construction of simple Turing machines is illustrated, their running is demonstrated through a physical model of symbol manipulation and their design outlined in a set of mantras. Fundamental ideas in computer science such as computable functions and algorithms are introduced on the basis of the Turing machine. The Church–Turing thesis is introduced to quickly dismiss all variations and enhancements of Turing machines as essentially equivalent to the standard version. Some of the variations are put together to construct the universal Turing machine thereby introducing the key idea of a stored-program computer.

10.1 *The Idea of* a Universal Computing Machine

In the year 1936, before modern digital computers were invented, Alan Turing, who is considered to be the *father of computer science* proposed a simple abstract machine and claimed that it could compute any mathematical function that was computable. He went even further to show that there are some functions that are not computable by any machine! Turing had two grand objectives for his work: to show why it was not possible to automatically prove or disprove all mathematical statements, and, to show that what goes on in the human mind was a form of computation. In a sense, Turing was at once trying to define what computer science was, what its limitations were and how it could lead to the development of artificial intelligence.

At a time when mathematicians were trying to systematize the way theorems were being proposed and proved so as to create a universal set of all valid mathematical theorems, in 1928, David Hilbert had posed a grand challenge known as *Entscheidungsproblem* or the *Decidability Problem*. This is the problem of deciding algorithmically whether any given statement is true or false. Soon after, in 1931, Kurt Gödel proved his famous *Incompleteness Theorem* which showed that any formal mathematical system necessarily had statements that could neither be proved nor disproved. Gödel's incompleteness showed that there could not be a solution to Hilbert's *Entscheidungsproblem*. But why not? Why can't we create a computing machine that takes any statement and tells us whether it is true or false? Turing's invention of the universal computing machine was in response to this challenge – to show why the Decidability Problem was unsolvable as a result of the *halting problem* (see Sec. 12.3, Chapter 12) of his invention which we now call a Turing machine.

Turing reasoned about how the human mind solves problems in mathematics. His conjecture was that the mind moved from one mental state to another while manipulating symbols on a sheet of paper which served as an external aid to human memory. He did not think it was necessary for the external memory to be a two-dimensional space of squares (see Fig. 10.1); rather, the rows of squares in such a mathematical worksheet could very well be collapsed and aligned into a single row of squares. Turing proposed to give his abstract computing machine a paper tape with such a row of cells, each of which could hold a single symbol.

This machine, or the Turing machine as we now call it, has a linear tape on which all computation is done. The tape is of infinite length: why limit its length in an abstract theoretical machine? It is as though the Turing machine is a computer with infinite memory capacity. The tape has a sequence of cells, each cell acting as a unit of memory. There is a *reading head* on the tape that can read the symbol in the current cell on the tape (i.e., the cell at which the head is currently pointing). It can also write a symbol onto the cell. And it can move one cell to the right or left. These are the only operations that the reading head can perform. When compared to modern digital computers, we can say that the Turing machine has a really simple instruction set with only four operations: read, write, move left and move right.

A set of symbols such as 0, 1, 2, x, y, z, etc. is used to write on the tape. These symbols are used to represent the input string as well as any output string on the tape which also acts as a scratchpad by storing any "local variables," values or intermediate results in its cells. The entire operation of the machine is controlled by a table of instructions that is encoded in the form of a finite automaton called the *control unit* (see Fig. 10.2). The machine processes the input string and makes transitions from one state of the finite automaton to another according to the transition function of the automaton. At the end, when there are no more transitions from the current state, we say that the Turing machine has halted. What is written on the tape at that point is the output of the computation, provided the state in which the machine halts is a final state of the automaton.

The transition function (or the behavior) of a Turing machine, like that of a pushdown automaton (PDA), depends not only on the current state of the machine but also on the current symbol at which the reading head of its tape is pointing. However, there is an important difference between a Turing machine and other computing machines that we have studied before – both finite automata and pushdown automata (PDAs). Unlike in those automata, the input string is not fed externally to the machine. As such, a Turing machine is not constrained to process the input string left-to-right in a single pass. It can move its reading head back and forth any number of times along the tape on which the input string is written. We will see soon how this ability makes it more powerful than earlier automata.

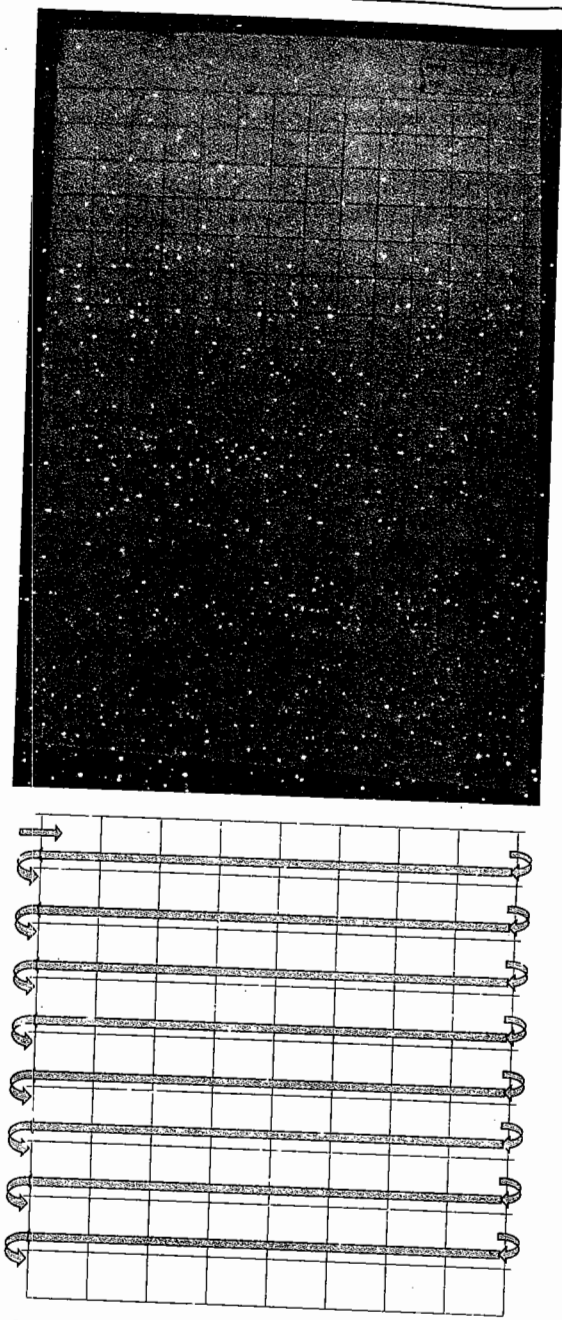


FIGURE 10.1 From a mathematics worksheet to a Turing machine tape.

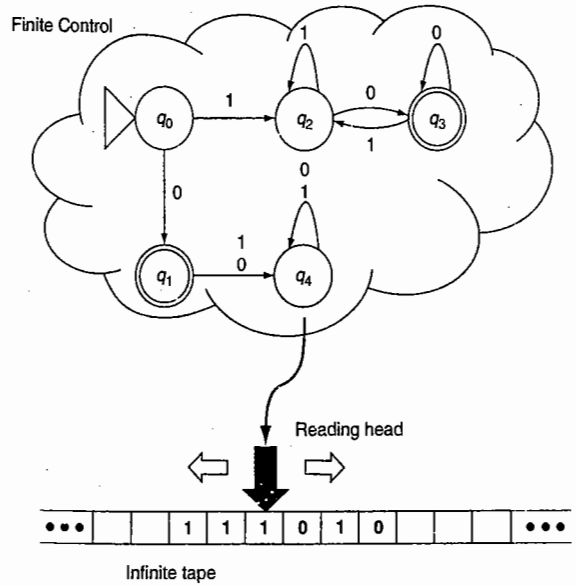


FIGURE 10.2 A Turing machine.

At the start of a computation, the control unit is in the start state of its automaton and the reading head is pointing to a cell on the tape. By convention, the input string is already written on the tape starting from either the current cell or the one immediately to its right. Sometimes, a special symbol such as \$ or < is used to indicate the starting cell. In some cases, there may also be a special marker such as > to indicate the end of the input string. Use of such special markers prevents the machine from going off the deep end looking for a non-existent symbol on its infinitely long tape (e.g., if there is a loop or "jolly ride" in the automaton that says move left).

Turing machines can produce two kinds of outputs:

1. They can actually compute a result from the input string and write the result on the tape. Or
2. If a possible result of a computation is given as a part of the input, they can give us a Boolean answer by halting either in a final or non-final state of the automaton, thereby indicating whether the given input is true or false or whether the given result is correct or not.

In either case, it is important that the machine halts after a finite number of transitions from its start state.

10.2 Constructing Simple Turing Machines

Before formally defining a Turing machine, let us see how it can solve a few computing problems.

EXAMPLE 10.1

Consider the problem of doubling a number (i.e., multiplying by 2). The input number is written on the tape in binary with the reading head at the beginning of the input (as shown in Fig. 10.4 (a)). Can we use a finite automaton to solve this problem? Yes, if you make it a finite transducer so that it gives output (see Sec. 3.6). Let us see how a Turing machine can act as a finite-state transducer to compute the double.

A Turing machine does its computation by going through a sequence of state transitions. Each transition is defined by the current state of the finite control and the current symbol on the cell at which the reading head is pointing. The machine starts in state q_0 as shown in Fig. 10.3. Our algorithm for doubling the input number involves moving to the end of the number and appending a 0 (which doubles the binary number). How do we make the reading head move to the end of the input? We simply skip each digit of the number until we come to a blank cell in the tape. As shown in Fig. 10.3, the control unit remains in state q_0 and repeatedly skips over 0s and 1s, moving the reading head right by one cell every time. These transitions are indicated by the loop from q_0 to q_0 which is labeled "0; 0, R" and "1; 1, R." These triples indicate the current symbol read from the tape; the new symbol written onto the same cell of the tape and the direction in which the reading head moves at the end of the transition – R meaning right and L meaning left. Notice that in these two transitions the symbol on the tape does not change at all. Still, it is customary to indicate the old and new symbols on the transition label.

Figure 10.4 shows the tape contents before, during and after solving the problem.

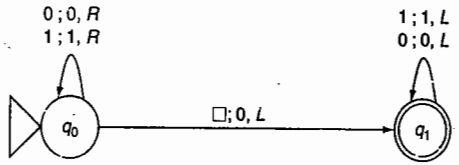


FIGURE 10.3 Turing machine for doubling a binary number.

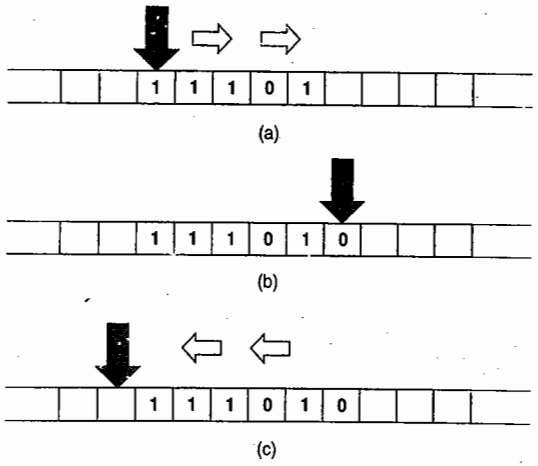


FIGURE 10.4 Tape contents of a Turing machine in doubling a binary number: (a) Before; (b) During and (c) Upon halting the computation.

The moment the reading head points to a blank cell, the machine changes its state to q_1 by going through the transition from q_0 to q_1 labeled "□; 0, L" (with the symbol □ denoting a blank cell on the tape) wherein it also writes the symbol 0 on the blank cell. At this point, the number has been doubled successfully and written on the tape as shown in Fig. 10.4(b). The machine is in the final state q_1. For the sake of completeness, the reading head can be brought back to the beginning of the output by looping back through all the 0s and 1s, without changing the state or the tape symbol but simply moving the head one cell to the left for each symbol. These transitions are shown as a loop from q_1 to q_1 with the labels "0; 0, L" and "1; 1, L." Notice that the machine will halt in state q_1 with the reading head pointing to the cell just to the left of the first symbol in the output as shown in Fig. 10.4(c). It is assumed that this cell is either blank or contains a special marker such as $. The machine will halt here because there is no transition going out of state q_1 on any symbol other than 0 or 1. As you can see, this machine is more like modern computing machines with which we are familiar because it can actually compute something.

EXAMPLE 10.2

Our next example is adding two numbers. As we saw in Example 6.5 in Chapter 6, this problem cannot be solved by finite automata but a PDA or an equivalent context-free grammar can solve it as shown in Example 7.10 in Chapter 7. As before, we will use the unary number representation to write the input and output numbers on the tape. The input to a Turing machine has to be provided by suitably initializing its tape with the input string written on it with appropriate delimiters. For the present problem, the input on the tape will constitute a unary number followed by a blank cell and then the second unary number. Another blank cell will be used to separate the second number from the sum of the two numbers in the output. There is no need to introduce different symbols such as a, b, and c for the three numbers. Thus, the language of addition in the unary number system is $1^n\square 1^m\square 1^{n+m}$ (with the alphabet {1, □}) and a corresponding context-free grammar (from Example 7.10) is as follows:

$$S \rightarrow 1S1 \mid \square A$$
$$A \rightarrow 1A1 \mid \square$$

What is our algorithm for performing the addition?

In the unary representation, addition is merely removing the blank between the two arguments! Given the input 111□1111 (i.e., 3 + 4) the correct output is 1111111 (i.e., 7).

How do we erase the blank in between? We could left-shift the second number by one cell. A Turing machine that accomplishes this is shown in Fig. 10.5. The machine first skips over the first number until it reaches the blank cell between the two numbers. Upon finding the blank, it changes its state from q_0 to q_1 to remember that it has moved past the first number and is now scanning the second number.

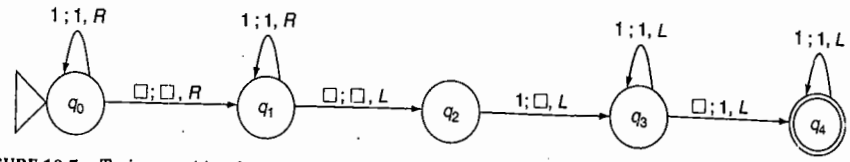


FIGURE 10.5 Turing machine for adding unary numbers.

In state q_1, it skips over the second number to find a blank beyond it and changes to state q_2. In q_2, it will be pointing to the last symbol of the second number [as shown in Fig. 10.6(b)]. It erases this symbol (i.e., replaces the 1 with a blank) and changes to state q_3. It next comes back over the second number and replaces the blank between the two numbers with a 1, thereby generating the sum of the two given numbers as the output on the tape. On doing this, the machine reaches the final state q_4 and, optionally, moves left over the first part of the number till it comes back to the cell just before the output number [see Fig. 10.6(d)].

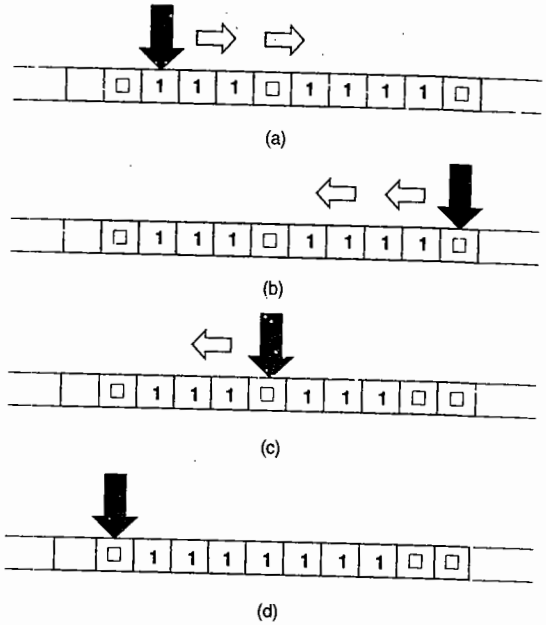


FIGURE 10.6 Tape contents during the execution of the adding Turing machine: (a) In state q_0, (b) in state q_2; (c) in state q_3; (d) in halting configuration.

Meanings of the different states of the machine are shown in Table 10.1 below. In the column for sample tape contents, the position of the reading head is indicated by inserting a downward arrow ↓ to the left of the symbol at which the head is pointing.

TABLE 10.1 Meanings of States in the Turing Machine of Fig. 10.5

State	Meaning	Sample Tape Contents
q_0	Start state; scanning the first number	...□□ ↓111□1111□□...
q_1	Scanning the second number	...□□111□↓1111□□...
q_2	Found the blank after the second number; now pointing to the last 1 of the input	...□□111□111↓1□□...
q_3	Have replaced the 1 with a blank; moving left along the second number	...□□111□↓111□□□...
q_4	Replaced the blank between the two numbers with a 1; final state; moving to the left through the first number	...□ ↓□1111111□□□...

Note that efficiency of computation (i.e., the total number of steps or transitions taken by the machine to process an input string) is not a concern in designing Turing machines. We are typically interested only in showing that the machine *can* perform the computation or that it can accept the language. It is far more important to ensure that the Turing machine is designed to handle all exceptions and boundary conditions. For example, what happens when the input is null? What if the first number to be added is zero (i.e., a null string in the unary representation)? What if the second number is zero? What if both are null? Are we sure that the Turing machine in Fig. 10.5 handles all these cases? Does it ever run off the deep end, traversing blank cells endlessly on its infinitely long tape?

Designing a Turing machine to handle straightforward cases and typical inputs is usually quite easy. Handling all the possible exceptions and special cases often takes a lot more effort. By the way, this is true in general about software development for modern computers as well.

A Turing machine can also be readily simulated in a physical model. Figure 10.7 shows such a simple model constructed using a cloth tape onto which symbols written on pieces of cardboard are stuck using strips of Velcro fasteners. Of course, in this simple model, the automaton and its transitions are executed by a human demonstrator.

FIGURE 10.7 A simple physical model of a Turing machine.

10.3 Turing Machine Definition

A Turing machine M has five elements:

1. M.alphabet: Denoted by Σ, the tape alphabet is the set of symbols used to write input, output and intermediate strings on the tape.
2. M.states: Also denoted by Q, it is the set of all states in the automaton.
3. M.startState: Usually denoted by q_0, it is the start state of the automaton in the control unit.
4. M.finalStates: Denoted by Q_F, it is the subset of M.states that are final states of the automaton.
5. M.transitionFunction: It is a function δ from $Q \times \Sigma$ to $Q \times \Sigma \times$ {Left, Right}, that is, a mapping from the current state and the current symbol under the reading head on the tape to a new state, a new symbol in the cell and a movement of the reading head by one cell to the left or to the right.

In addition, a special symbol □ denotes a blank cell. Obviously this symbol cannot be also present in the tape alphabet, inputs or outputs. The tape memory is in fact a sequential access memory (not quite random access memory, RAM, but RAM is merely more efficient than sequential access memory). In a Turing machine, there are no pre-assigned addresses to memory cells and we cannot compute a memory address and then jump to that address the way a C program accesses array elements. Nevertheless, the Turing machine can eventually bring its reading head to any memory cell. There is no *Jump* or *Go To* operation; only a *move left* or *move right* by one cell at a time.

The tape alphabet can include symbols which are not used in input strings; such symbols are generally used only in keeping intermediate results. The tape head *must* move left or right in every transition. If the machine halts in a final state after processing the input, it is said to accept the input string; otherwise, it is said to reject the input.

We also assume that the machine is a *deterministic Turing machine*. That is, for a given state and a given current symbol on the tape, there is only one possible transition. In other words, the transition function is indeed a mathematical function. It is interesting to note that deterministic finite automata had the same power as non-deterministic finite automata, but we needed non-deterministic PDAs to handle context-free languages. Turing machines, like finite automata, do not really need non-determinism. We will see later in Section 10.8 why this is so.

The snapshot or configuration of a Turing machine at a particular step during a computation can be captured by including the contents of its tape (excluding the blank cells that extend infinitely on either side), the position of the reading head and the current state of the automaton in the control unit. A configuration is usually denoted by a string such as

$$q_0{:}\square\downarrow xxyyxx\square$$

where q_0 is the current state, the tape contains the string $xxyyxx$ and the reading head denoted by the downward arrow ↓ is pointing to the first non-blank symbol x.

A Turing machine is said to accept a string if, given the string as input, the machine processes it and halts in a final state of the machine. The set of all strings over the alphabet that a Turing machine accepts is called the language of the machine (sometimes denoted by M.language).

10.4 Constructing More Complex Turing Machines

Let us consider a few more examples of constructing Turing machines, at the same time trying to formulate a set of *mantras* useful for the design of Turing machines.

EXAMPLE 10.3

Let us consider the example of accepting a language that is not context-free: $a^n b^n c^n$ that we could not solve using finite automata or PDAs. Figure 10.8 shows a Turing machine that accepts the equivalent language $0^n1^n2^n$ for the trivial case of $n = 1$, that is, for the simplest string in the language, namely, 012.

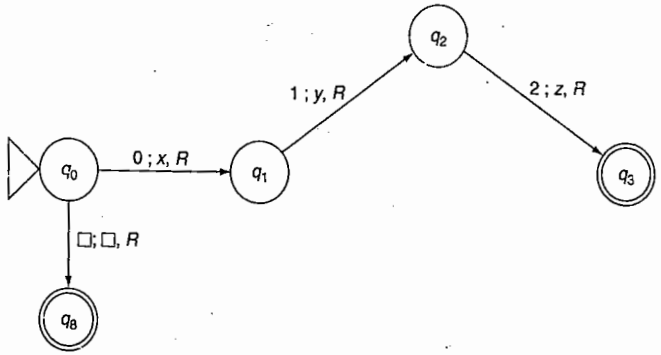


FIGURE 10.8 Trivial Turing machine for $0^n1^n2^n$.

A non-trivial sample string in the language is 000111222 for $n = 3$. What is our algorithm for computing this language? First, we must be able to skip over intermediate symbols. For example, having marked the first 0, we must skip over the remaining 0s before we find a 1, and then skip over the remaining 1s before finding a 2. Similarly, we must be able to come back to the left to find the next matching triple and in coming back we must skip over not only all the remaining 1s and 0s but also any new symbols such as x and y that we have used to mark 0s and 1s that are already matched. Changing the input symbol to indicate that it is already processed is a common technique that is very useful in designing Turing machines.

Our Turing machine must process the input, halt and accept the string only if it is in the language. Finite automata cannot deal with this language since they do not have any memory to count and match the number of 0s, 1s or 2s. PDAs also cannot process this language since they are unable to match three counts in their single stack memory. Since Turing machines are not constrained to process the input left-to-right in a single pass, in our solution the machine can move back and forth along the tape matching corresponding triples of 0s, 1s and 2s. That is, as soon a 0 is found, the machine can move ahead and find the corresponding 1 and then move further to the right to find the corresponding 2. This process can be repeated n times in a loop to completely consume the input string. Each symbol that is processed must also be marked, as already noted, by changing the symbol so that the same symbols are not processed over and over again.

This is essentially a recursive solution: given $0^n1^n2^n$, it erases (or replaces) a 0, a 1 and a 2 thereby generating $0^{n-1}1^{n-1}2^{n-1}$. Finally, it arrives at $0^01^02^0$, or the null string which it must recognize efficiently. It can also be seen as an iterative solution which goes through a loop n times matching one set of symbols in each loop. Such a loop of states can be seen in the Turing machine in Fig. 10.9 which replaces 0 by x, 1 by y and 2 by z as it matches them. Each loop through the states q_1–q_2–q_3–q_4–q_1 finds and matches one set of the three symbols. The terminating condition for when it should reach and halt in its final state is interesting. As it is coming back left to find the next 0 (q_3 to q_4), it finds that there is a y immediately next to an x (in q_4 to q_5), indicating that the 0s are exhausted. It must ensure that 1s and 2s are also exhausted (in q_5 and q_6), for, if not, there would be an extra 1 or an extra 2 thereby telling us that the input string is not in the language. The meanings of the various states in this Turing machine are shown in Table 10.2 below.

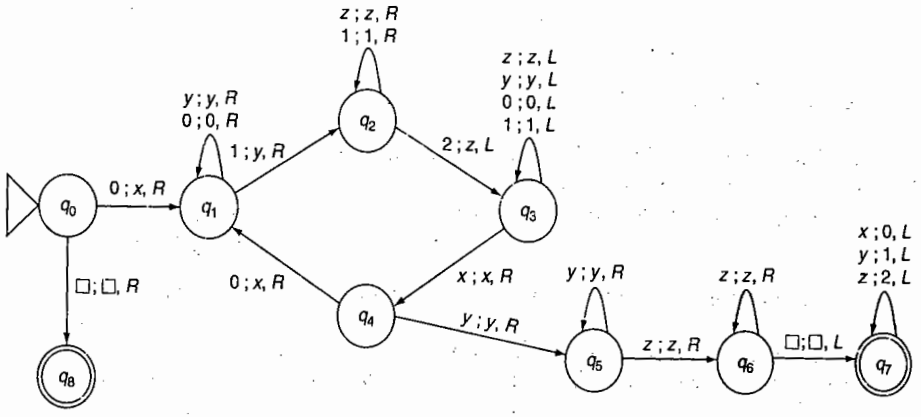


FIGURE 10.9 Turing machine for $0^n1^n2^n$.

How does this machine reject input strings that are not of the expected format? Let us see what happens when there are more 0s than 1s or 2s. The machine will halt in state q_1, a non-final state and thereby reject the input. If there are more 1s than 0s or 2s, the machine will halt in state q_5, unable to find a z immediately after the last y. We must convince ourselves that the machine behaves appropriately for all of the following possible inputs. Does it? It is useful for the reader to trace the execution of this machine for these input strings to fully understand how it works:

1.	0012	Extra 0s
2.	0011122	Extra 1s
3.	00011122222	Extra 2s
4.	0111222	Too few 0s
5.	00011222	Too few 1s
6.	00112	Too few 2s
7.	111222	0s missing altogether

8. 000222	1s missing altogether
9. 000111	2s missing altogether
10. 002211	Wrong order
11. 110022	Wrong order
12. 112200	Wrong order
13. 221100	Wrong order
14. 220011	Wrong order
15. λ	Empty input string

TABLE 10.2 Meanings of States in the Turing Machine of Fig. 10.9

State	Meaning	Sample Tape Contents
q_0	Start state; if the input is blank, go right to the final state q_8	...□□↓000111222□□...
q_1	Have just found a 0 and replaced it with an x; skipping over any more 0s or y s to find a matching 1	...□□x00↓111222□□...
q_2	Have just found a 1 and replaced it with a y; skipping over any more 1s or z s to find a matching 2	...□□x00y1↓1222□□...
q_3	Have just found a 2 and replaced it with a z; moving back to the left skipping over z s, 1s, y s and 0s	...□□↓x00y11z22□□□...
q_4	Found an x, have come back to the beginning of the unprocessed input; if there is a 0, replace it with an x and go back to q_1 to continue matching the next set of symbols	...□□↓x00y11z22□□...
q_5	There is a y next to an x, that is, no more 0s; move right skipping over all the y s till a z is found (i.e., no extra 1s in the input)	...□□xxx↓$yyyzzz$□□...
q_6	There is a z next to a y, that is, no more 1s; move right skipping over all the z s till a blank is found (i.e., no extra 2s in the input)	...□□$xxxyyy$↓zzz□□...
q_7	Just found a blank and the end of the input; final state; move left replacing the original symbols (2 for z, 1 for y and 0 for x)	...□□$xxxyyyzz$↓z□□...
q_8	Special final state for handling null input (i.e., $n = 0$)	... ↓□□

As you can see, it often takes much more effort to show that all the types of non-members of the language are rejected than to show that valid members are accepted.

This is not the only possible Turing machine for the language. Let us consider the alternative shown in Fig. 10.10 which does not replace 0, 1 and 2 by x, y and z. Instead, it erases the input symbols, assuming that it is not important to keep the input string on the tape at the end of the computation. One difficulty that arises in such a machine is in deciding when to stop moving left looking for a symbol (so that the machine does not go infinitely far to the left). As shown in Fig. 10.10, we can introduce a special marker \$ at the beginning of the input to address this problem. Table 10.3 lists the meanings of all the states in this machine.

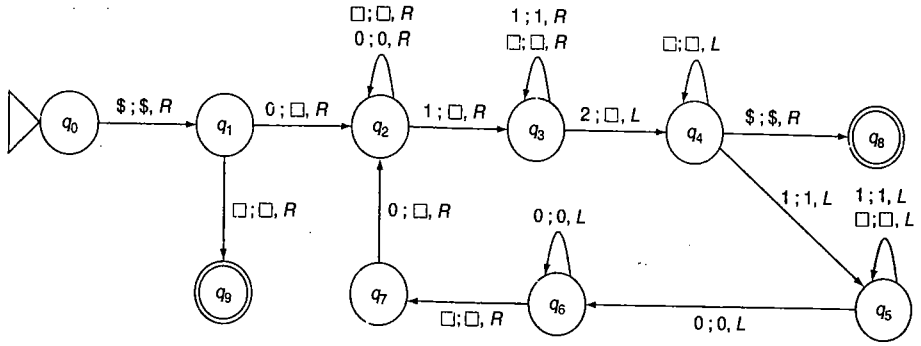


FIGURE 10.10 Another Turing machine for $0^n1^n2^n$ – has a bug!

After erasing one set of matching 0, 1, and 2 triples (q_2–q_3–q_4), the machine comes back to the left finding more 1s (in q_4–q_5) that are yet to be matched and more 0s as well (in q_5–q_6). It finds the leftmost 0 which is not matched (in q_6–q_7) and proceeds back onto the loop by erasing the symbol (in q_7–q_2).

TABLE 10.3 Meanings of States in the Turing Machine of Fig. 10.10.

State	Meaning	Sample Tape Contents
q_0	Start state; the input must start with a \$	...□ ↓\$000111222□...
q_1	Erase a 0 and move to q_2; if the input is blank, go right to the final state q_9	...□\$□ ↓00111222□...
q_2	Have just found a 0 and erased it; skipping over any more 0s or blanks to find a matching 1	...□\$□00□↓11222□...
q_3	Have just found a 1 and erased it; skipping over any more 1s or blanks to find a matching 2	...□\$□00□1↓1□22□...
q_4	Have just found a 2 and erased it; moving back to the left skipping over blanks looking for a 1; if \$ is encountered instead, there are no more symbols and the final state q_8 can be reached	...□\$□00□1↓1□22□...
q_5	Skip over 1s and blanks, moving left and looking for a 0	...□\$□0↓0□11□22□...
q_6	Found a 0; skip over any more 0s moving left to find the leftmost unprocessed 0	...□\$ ↓□00□11□22□...
q_7	Found a blank; move right to the first 0; erase it and loop back to q_2 to find matching 1 and 2	... □\$□□↓0□11□22□...
q_8	Final state reached	...□\$↓□□□□□□□□□□...
q_9	Special final state for handling null input (i.e., $n = 0$)	...□\$ ↓□...

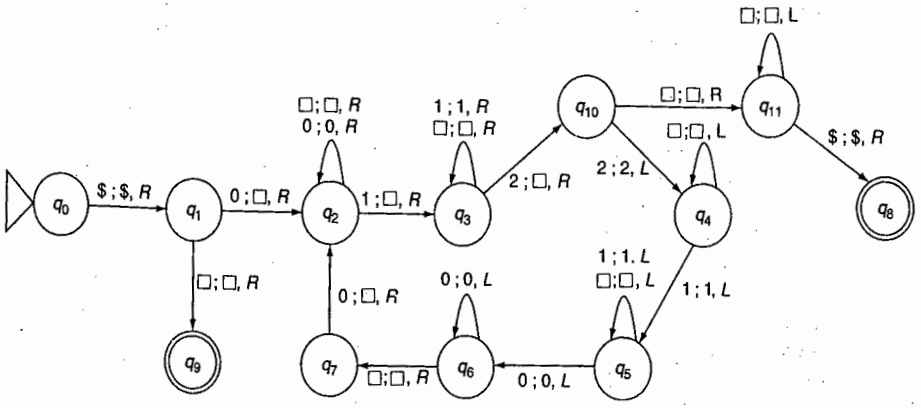


FIGURE 10.11 Another Turing machine for $0^n1^n2^n$ – Bug fixed.

Is the machine shown in Fig 10.10 perfect? If you analyze and debug this machine, you will discover a problem: what happens when the input is \$0011222? Figure 10.11 shows a repaired Turing machine for the same language which fixes this bug. The fix introduces two new states q_{10} and q_{11} to verify that there are no extra 2s at the end of the input string. The machine cannot look for extra 2s after coming back left to reach the \$ symbol. This is because, if there aren't any extra 2s, it will go off infinitely to the right and never halt! To overcome this problem, the machine in Fig. 10.11 looks for extra 2s when it is in the right context, that is, in state q_{10}. Having just found a 2 and erased it, it inspects the next symbol to the right. If it is a 2, it comes back to the left (and to state q_4) looking for matching 1s and then 0s. If it is a blank, the 2s are all exhausted and the machine goes to state q_{11} and verifies that there are no more 1s and 0s all the way back to \$. Only then the string is accepted. If any other symbol (1 or 0) exists before reaching \$, the input is rejected by halting in the non-final state q_{11}.

Is the machine in Fig. 10.11 a complete and correct solution for the language $0^n1^n2^n$? Figuring this out is left as an exercise for you.

EXAMPLE 10.4

Finally, let us construct Turing machines for two seemingly similar languages: the language of even palindromes ww^R and the language ww, which we know is not context-free (from Example 9.2 in Chapter 9). Figure 10.12 shows a Turing machine for even palindromes over the input alphabet $\{x, y\}$. It replaces x by 1 and y by 2 as it goes on matching symbols inwards from either end of the input string. That is, if it finds an x at the beginning, it goes right to the end of the unprocessed string to find and mark the corresponding x in the palindrome. At the end, all the symbols have been replaced by 1 or 2 and the machine halts in its only final state q_6. This machine does not bother to replace all the original symbols back but that can be accomplished fairly easily as in earlier examples.

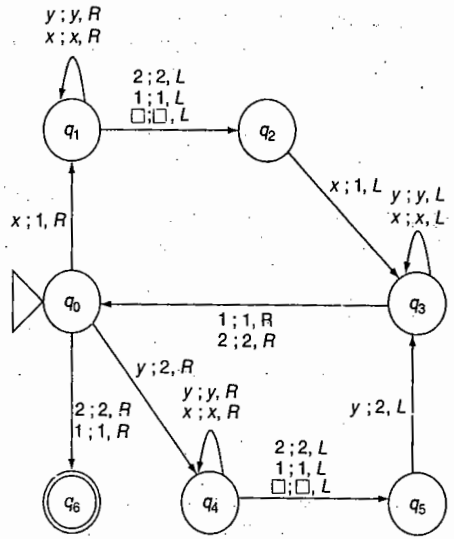


FIGURE 10.12 A Turing machine for even palindromes.

A complete sequence of 29 configurations that this Turing machine goes through in processing the sample input string *xxyyxx* is shown below:

q_0:□↓xxyyxx□ ⇒ q_1:□1↓xyyxx□ ⇒ q_1:□1x↓yyxx□ ⇒
q_1:□1xy↓yxx□ ⇒ q_1:□1xyy↓xx□ ⇒ q_1:□1xyyx↓x□ ⇒
q_1:□1xyyxx↓□ ⇒ q_2:□1xyyx↓x□ ⇒ q_3:□1xyy↓x1□ ⇒
q_3:□1xy↓yx1□ ⇒ q_3:□1x↓yyx1□ ⇒ q_3:□1↓xyyx1□ ⇒
q_3:□↓1xyyx1□ ⇒ q_0:□1↓xyyx1□ ⇒ q_1:□11↓yyx1□ ⇒
q_1:□11y↓yx1□ ⇒ q_1:□11yy↓x1□ ⇒ q_1:□11yyx↓1□ ⇒
q_2:□11yy↓x1□ ⇒ q_3:□11y↓y11□ ⇒ q_3:□11↓yy11□ ⇒
q_3:□1↓1yy11□ ⇒ q_0:□11↓yy11□ ⇒ q_4:□112↓y11□ ⇒
q_4:□112y↓11□ ⇒ q_5:□112↓y11□ ⇒ q_3:□11↓2211□ ⇒
q_0:□112↓211□ ⇒ q_6:□1122↓11□

Notice that unlike the PDA for this language of even palindromes (see Example 8.3 in Chapter 8), our Turing machine is deterministic. We do not require non-determinism to guess the midpoint of ww^R as in the case of the PDA. This is due to the additional power that Turing machines have as they are not constrained to process the input string left-to-right in a single pass. Our Turing machine did not bother to find the midpoint of the input string; instead, it kept matching leftmost and rightmost symbols in the input by moving its reading head as needed. Eventually, just before it halts, it is anyway at the midpoint of the string.

EXAMPLE 10.5

The non-context-free language ww is accepted by the Turing machine shown in Fig. 10.13. In this case, we do need to know where the second w starts, that is, we do need to find the midpoint. However, it is too difficult to count the length, divide it by 2 and use it to match symbols in the first half with corresponding ones in the second half. We want to somehow find the midpoint and separate the two halves of the string without having to count lengths. This, we are able to do by building a machine that shifts the second half of the string by one cell to the right, creating a blank space between the first and the second halves.

The top half of the Turing machine in Fig. 10.13 (i.e., q_0–q_1–q_2–q_3–q_4–q_5–q_6) finds the midpoint of the string using such a method. In addition, it also marks the midpoint by right-shifting the entire second half of the string by one cell. Its algorithm is:

1. Mark a symbol at the left of the first string w (q_0–q_1).
2. Go till the end of the string (q_1–q_2).
3. Right-shift the rightmost symbol of the second string w (q_2–q_3/q_5–q_4).
4. Come back to the left and repeat (q_6–q_0).

Notice that the symbols are replaced (0 for x; 1 for y) only in the first half of the input. Thus, the first half is easily distinguishable from the second half. In the end, if the string is of even length, a blank will be encountered in the middle (in state q_0) and the machine will move on to the second part of the algorithm shown in the bottom half of Fig. 10.13 (i.e., q_0–q_7–q_8–q_9–q_{10}–q_{11}–q_{12}–q_{13}):

1. Come back from the midpoint to the beginning of the first string (q_7).
2. Insert a special symbol \$ to mark the start of the first string (q_7–q_8).
3. Erase a symbol (0 or 1) at the left of the first string (q_8–q_9 or q_8–q_{11}).
4. Skip over to the right until a matching x or y is found and erase it (q_9–q_{10} or q_{11}–q_{10}).
5. Come back to the first part skipping over blanks (q_{10}–q_{13}).
6. If there are more symbols in the first half, come back to q_7 and continue looping.
7. Else go to the final state q_{12}.

Are there any useless transitions in this Turing machine? Is the link from q_0 to q_{12} or the one from q_{10} to q_{12} ever used? What happens when q_7 to q_8 is traversed in successive loops? A good way to answer such questions is to trace the configurations of the Turing machine as it processes an input string. A trace for the input xyxy is provided below:

q_0:□↓$xyxy$□□ ⇒ q_1:□0↓yxy□□ ⇒ q_1:□0y↓xy□□ ⇒
q_1:□0yx↓y□□ ⇒ q_1:□0yxy↓□□ ⇒ q_2:□0yx↓y□□ ⇒
q_5:□0yx□↓□□ ⇒ q_4:□0yx↓□y□ ⇒ q_6:□0y↓x□y□ ⇒
q_6:□0↓yx□y□ ⇒ q_6:□↓0yx□y□ ⇒ q_0:□0↓yx□y□ ⇒
q_1:□01↓x□y□ ⇒ q_1:□01x↓□y□ ⇒ q_2:□01↓x□y□ ⇒
q_3:□01□↓□y□ ⇒ q_4:□01↓□xy□ ⇒ q_6:□□0↓1□xy□ ⇒

q_0:□01↓□xy□ ⇒ q_7:□0↓1□xy□ ⇒ q_7:□↓01□xy□ ⇒
q_7: ↓□01□xy□ ⇒ q_8:\$↓01□xy□ ⇒ q_9:\$□↓1□xy□ ⇒
q_9: \$□1↓□xy□ ⇒ q_9:\$□1□↓xy□ ⇒ q_{10}:\$□1↓□□y□ ⇒
q_{13}: \$□↓1□□y□ ⇒ q_7:\$↓□1□□y□ ⇒ q_8:\$\$↓1□□y□ ⇒
q_{11}: \$\$□↓□□y□ ⇒ q_{11}:\$\$□□↓□y□ ⇒ q_{11}:\$\$ □□□↓y□ ⇒
q_{10}: \$\$□□↓□□□ ⇒ q_{13}:\$\$□↓□□□□ ⇒ q_{13}:\$\$↓□□□□□ ⇒
q_{13}: \$↓\$□□□□□ ⇒ q_{12}:\$\$↓□□□□□

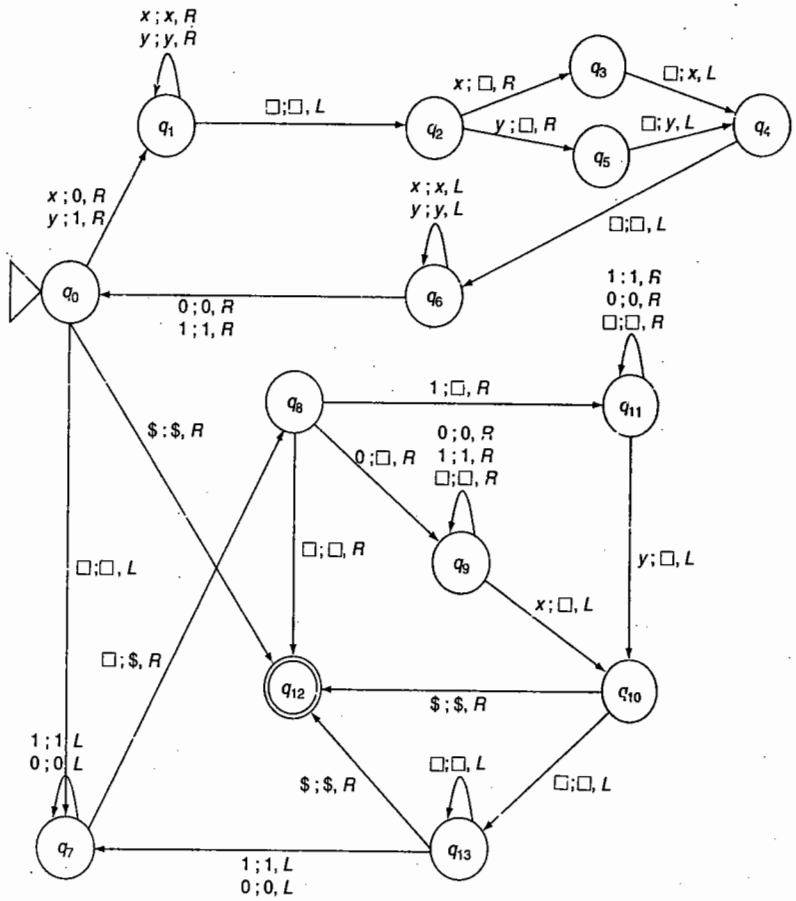


FIGURE 10.13 A Turing machine for the language ww.

10.5 *Mantras* for Constructing Turing Machines

Having constructed several Turing machines, let us write down a set of *mantras* for designing them:

1. Find a good algorithm for the given problem before adding circles, arrows and loops to the state transition diagram of your Turing machine.
2. Consider the simplest string in the language and construct a Turing machine to accept that string.
3. Consider various exceptions and boundary conditions and verify that each one is satisfied, the null input λ in particular.
4. Consider the complement of the language, that is, strings that are not in the language and ensure that they are all rejected.
5. Analyze the intermediate tape contents and outputs to test the accuracy of the machine.
6. Ensure that the machine does not run off to the left or right *ad infinitum* looking for a non-existent symbol. Consider inserting special markers such as \$, < or > to prevent such a run off. Incidentally, this is similar to ensuring the terminating condition of a loop in a program.
7. Ensure that the machine halts for every possible input. For some problems, this will turn out to be impossible as we will see in Chapter 12.
8. State precisely the meaning of each state in the machine. If it is difficult to explain the meaning of a state concisely, its status in the design of the Turing machine is questionable.
9. Trace the execution of the Turing machine for a variety of inputs by going through successive configurations in detail.

You may feel that you have to consider too many possibilities in designing a perfect Turing machine. However, you must realize that programming in a high-level programming language and designing a Turing machine are essentially one and the same. In programming too, you need to consider numerous possibilities; you need to worry about various exceptions and boundary conditions. Hence in programming too, in real life, we often follow a similar methodology: write the code (or build the Turing machine) for the most normal case first and then worry about outliers and exceptions.

At times, we are not sure if the Turing machine we have designed is correct or not. There is again a parallel between this and what happens in real-life software development. It is unrealistic to think that software requirements can be specified completely and precisely and that doing a detailed design will result in perfect code.That is why we have the important stages of software testing, verification and validation. Even Turing machines need to tested and verified either manually or using a simulation package.

10.6 *The Ideas of* Computation, Computable Functions and Algorithms

We are now ready to formally define fundamental concepts such as *computation* and *algorithm*. The very idea of *computation* is defined as a series of moves or transitions that a Turing machine makes when given an input until it *halts* in some configuration (whether in an accepting state or otherwise).

A *computable function* is a function that a Turing machine can compute. Most functions with which we are familiar are in fact computable. In other words, it is possible to construct a Turing machine for most functions that are of interest to us.

Turing showed the reason why some numbers (or functions) are not computable. For such numbers, if you try to give a table of instructions to a Turing machine, that machine will never halt; today we call it an infinite loop. Before high-level programming languages existed, Turing showed that, no matter which table of instructions you give, you will always end up with a machine that goes into an infinite loop for some inputs. Later in Chapter 12 (Sec. 12.3) we will see why this is so.

We can use this idea to define what an algorithm is. An *algorithm* is defined simply as a Turing machine that does not go into an infinite loop for any inputs. In other words, an *algorithm* is a Turing machine that computes a function and halts in some configuration for all inputs in the domain. This is really nothing revolutionary: we have already defined computation as anything that a Turing machine does and a computable function as one that the same machine can compute, so what the machine does can also be termed an algorithm.

An algorithm must terminate after a finite number of steps in either an accepting or a rejecting state. An algorithm is a systematic procedure, a set of instructions that will always halt and give an answer, an output. Fortunately, there are algorithms for most practical problems. A majority of real-world problems, for the most part, do not suffer from Turing's halting problem (see Sec. 12.3). They are all computable problems. Real-world software engineering and the computing industry need not be bogged down by the limitations of computability shown by Turing.

10.7 The Church–Turing Thesis

We just defined important terms such as computation and algorithm using Turing machines. What if there are computing problems that Turing machines cannot handle? Fortunately, Alan Turing (and also Alonzo Church, independently) showed that anything that can be computed can be computed by a Turing machine. This statement has no proof and is not a theorem. However, no one so far has been able to find a computing problem that Turing machines cannot compute but some other machine, mechanism or formalism can. Hence this claim is known famously as *The Church–Turing Thesis.*

We can also turn this around and say only functions (or numbers or problems) that are computable by Turing machines are computable. It is a strange fact that this highly abstract and simple machine with a seemingly trivial instruction set (in comparison to a modern CPU) is in fact universal and most powerful. At the same time, from a practical point of view, it is perhaps also the most inefficient computing platform that one can build! Yet, its simplicity is the basis for proving some of the most important theoretical results in computer science.

10.8 Variations of Turing Machines

Many attempts have been made to enhance or modify the design of a standard Turing machine to make it more powerful. All the resulting variations have turned out to be strictly equivalent to the standard Turing machine in terms of the set of functions that it can compute, thereby supporting the Church–Turing thesis.

We can show that a particular variant is equivalent to the standard machine by simulating the computation done by one machine in the other. Instead of providing lengthy proofs of the equivalence of each variation and the standard Turing machine, let us explore a few modifications and enhancements, and briefly outline the reason for their equivalence (see Appendix B for formal statements and sketches of their proof):

1. **Machines with the stay option:** In the standard Turing machine, the reading head must move left or right by one cell in every transition. What if we allow the head to remain in the same cell? Such a stay-option Turing machine is still equivalent to a standard machine because staying in the same cell is the same as moving once to the right (or left) and moving back left (or right) in an additional dummy transition. Similarly, if you would like to allow the reading head to jump to a location by moving more than one cell in a single transition, it would still not alter the computing power of the machine (although it might make the machine more efficient by reducing the total number of moves it must make to solve a problem).
2. **Multi-track machines:** What if we make the tape multi-track, that is, each cell is divided into say 4 tracks so that 4 symbols can be stored in a single cell? The reading head would also be able to read all four symbols at once from a cell. This is still equivalent to the standard machine because each sequence of four consecutive cells on the standard tape become equivalent to a single cell on the 4-track tape. Once again, efficiency might be gained but computing power does not change.
3. **Semi-infinite tape machines:** What if the tape is infinitely long only on one side instead of at both ends? This too can be shown to be equivalent to the standard Turing machine. Using the idea of multi-track machines, if we split the semi-infinite tape into a two-track tape, the second track can be used to simulate the contents of the missing half of a full infinite tape.
4. **Multi-tape machines:** How about giving the machine more than one tape, each with its own reading head? It can be shown once again that the workings of a multi-tape machine can be simulated on the single tape of a standard machine albeit with an increase in the number of transitions required to accomplish the same computation. The idea of a multi-tape machine is central to the design of a Universal Turing Machine, as shown in Sec. 10.9.
5. **Multi-dimensional tape machines:** Even if the tape is two-, three-, or multi-dimensional instead of being a linear sequence of cells, the machine can compute the same set of functions that the standard machine can.
6. **Non-deterministic machines:** Non-deterministic Turing machines have more than one possible transition for the same current state and the same symbol on the tape. They too are equivalent to their deterministic counterparts because the guesswork that they do can be seen as a form of backtracking search which can be implemented in a deterministic machine by carrying out appropriate bookkeeping operations (to remember all the paths still to be explored). A Turing machine does not need non-determinism. Showing that even non-determinism does not increase the computing power of standard Turing machines further confirms the Church–Turing thesis that anything that can be computed by any kind of computing machine can also be computed by the standard Turing machine.

The standard Turing machine is most powerful in terms of the set of functions that it can compute (or formal languages that it can accept). However, there is still something ridiculously impractical about it, namely, that we have to design and build different Turing machines for each computable function! The adding computer cannot multiply and the multiplying computer cannot recognize even palindromes, and so on. Turing overcame this apparent limitation and showed that a particular kind of multi-tape Turing machine can in fact be a universal computing machine – a single computing machine that can compute any computable function.

10.9 The Universal Turing Machine

A *Universal Turing Machine* is a "programmable" machine that can compute anything that is computable as opposed to separate Turing machines computing individual functions that they were hardcoded to compute. A universal machine M_U should be able to simulate the computation carried out by any hardcoded Turing machine M. Turing proposed such a machine by providing the standard Turing machine with more than one tape (and with one independent reading head for each tape). The universal Turing machine M_U has three tapes (see Fig. 10.14):

1. The *input–output tape* is used to simulate the tape of the particular machine M (or computable function) which it is simulating. The input as well as any output is written on this tape.
2. The *compiled program tape* stores "the program," that is, the table of instructions or an encoding of the transition functions of the Turing machine M which it is simulating. This tape is a read-only tape.

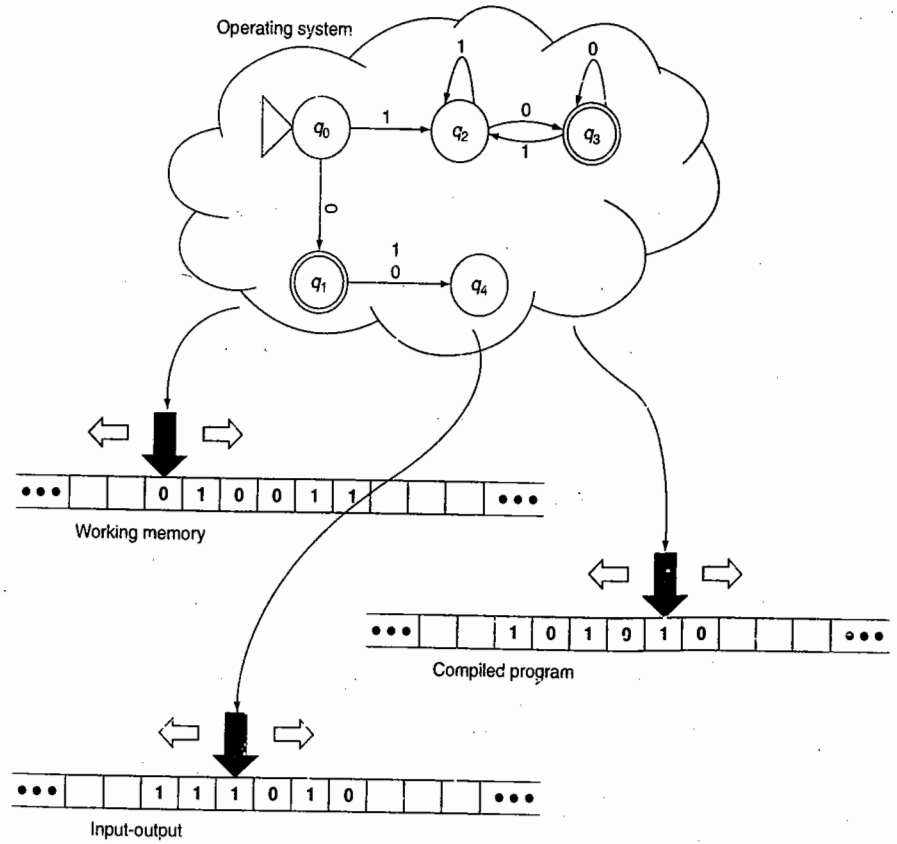


FIGURE 10.14 Universal Turing machine M_U.

3. The *working memory tape*, on which the current state of the machine M being simulated is maintained, along with any working memory (i.e., scratchpad) it needs.

The three tapes and their three independent reading heads are controlled by a finite control unit with its own states and transition functions. This unit can very well be considered the *operating system* of the universal Turing machine. It works as follows:

1. Look up the current state information maintained in working memory and the current symbol on the input–output tape.
2. With these two, scan the compiled program tape to find a transition for the given state and current symbol.
3. Execute the transition, that is, write the new symbol on to the input–output tape and move its reading head to the right or left as specified in the transition; note the new state mentioned in the transition in the working memory.
4. Repeat these steps until there are no more transitions in the compiled program.
5. When it halts, if the current state is a final state in the machine M being simulated, then the input is accepted; otherwise the input is rejected.

In order to build such a universal computing machine M_U, we need a way of encoding the transitions of the machine M as a string that can be loaded onto the compiled program tape of M_U. In other words, we need to define a *universal programming language* and a *compiler* that encodes the entire Turing machine M as a string that can be executed by M_U. Such an encoding language can be designed using binary numbers to represent the entire table of instructions of the Turing machine M as follows:

1. Standardize all elements of the Turing machine M except its transition function – let us assume that the alphabet is always $\{a_1, a_2, a_3, \ldots\}$; that its states are always numbered q_1, q_2, q_3, etc. where q_1 is always the start state and q_2 is always the single final state. It can be shown that there is no loss of generality in doing this. The benefit of doing this is we can drop the qs and as and merely use the state number and the symbol number in encoding the machine M.
2. Each transition from the state q_i with current symbol a_j going to state q_k and new symbol on the tape a_l with the reading head moving left (L) or right (R), that is, $(q_i, a_j) \rightarrow (q_k, a_l, \text{R/L})$ is encoded as a sequence of numbers $(i, j, k, l, 1/11)$, where 1 stands for R and 11 for L. This sequence can be encoded as unary numbers separated by a single 0. For example, the transition $(q_3, a_1) \rightarrow (q_4, a_2, \text{R})$ would be encoded as 111010111101101.
3. Each such encoding of a transition in M can be padded by two 0s on either side, for example, as 00111010111110110100 and the encodings of all the transitions can be concatenated to create a single binary string that represents the entire Turing machine M. The resulting string is a compiled program or executable binary program of the machine M.
4. This string is ready to be loaded onto the compiled program tape of the universal machine M_U. The control unit (or operating system) of M_U acts as an interpreter (or *virtual machine*) for this encoding so that the computation carried out by any Turing machine M can now be simulated on the universal machine M_U.

We thus have a universal Turing machine M_U that can load and execute the program of any other Turing machine M using the encoding mechanism (or programming language) outlined above. The universal

machine M_U can simulate the behavior of any computable function. Hence M_U can compute anything that can be computed. A standard Turing machine M which computes a function or accepts a language can be simulated in an M_U using its three tapes with independent reading heads which are all controlled by its single control unit.

10.10 *The Idea of* a Stored-Program Computer

Turing suggested that such a universal Turing machine can simulate the behavior of any computing device. This idea is what has been implemented in modern digital computers. Software programs are written in high-level programming languages such as C or Java™. They are compiled by a compiler for the programming language. The compiler encodes the computing program into a binary executable program (or to an intermediate language to be interpreted at run time as in Java™). This is then loaded onto the memory of the digital computer and its operating system (sometimes with the help of a virtual machine such as the one for Java™) executes the program in its general purpose CPU.

Turing's idea of storing an encoded table of instructions (i.e., a program) of a Turing machine on one of the tapes of a universal Turing machine made it possible for us to build modern computers that can compute (practically) anything that is computable. This idea is one of the greatest contributions of Alan Turing because of which we have today programmable, general-purpose computers that can execute any program given to them. Also, Turing machines for simple computable functions can be combined with one another to compute more complex functions. This is easy to see in the encoded program in a universal Turing machine since we can easily concatenate the encoded programs of multiple Turing machines (after renaming or rather renumbering states and symbols as needed) to create programs for more complex Turing machines. This is similar to what we do today in the name of modularity in programming, APIs and software system integration.

It may also be noted that Turing's idea of a tape with a sequence of memory cells and a reading head has been implemented quite literally in modern computers in the form of RAM, magnetic tapes, hard disks and compact disks. It is quite remarkable that in one research paper in 1936 Alan Turing gave to the world such a comprehensive picture of modern day computing.

10.11 Key Ideas

1. The Turing machine is a simple, abstract computing machine that can compute anything that is computable. In particular, Turing machines are the first kind of computing machines that we have seen which can accept languages that are not context-free.
2. The Turing machine is a finite automaton with a tape memory of unlimited size. Memory cells are accessed by a reading head that moves sequentially along the tape. The tape serves as a working memory for the automaton.
3. The input to a Turing machine as well as its output is written on the same tape. Unlike other automata, a Turing machine can process its input string in any order, not just left-to-right in a single pass.
4. A *computation* is defined as a series of moves or transitions that a Turing machine makes when given an input until it *halts* in some configuration (whether in an accepting state or otherwise).

5. A *computable function* is a function that a Turing machine can compute. Most functions with which we are familiar are computable. In other words, it is possible to construct a Turing machine for most functions that are of interest to us. However, Turing machines are not unique for a given function. There can be more than one machine (or table of instruction) for comuting the same function.
6. An *algorithm* is a Turing machine that computes a function and halts in some configuration for all inputs in the domain of the function.
7. Fortunately, there are algorithms for most practical problems. For some problems though, Alan Turing showed that all computing machines suffer from the *halting problem*, a limitation of computing.
8. Alan Turing and Alonzo Church independently showed that anything that can be computed can be computed by a Turing machine. No one so far has been able to find a computing model that can compute things that Turing machines cannot. Hence this claim is known as *The Church–Turing Thesis.*
9. All variations of the standard Turing machine are equivalent to it. Stay-option, multi-track, semi-infinite, multi-tape, multi-dimensional and non-deterministic Turing machines are all equivalent to the standard machine. They cannot compute anything that is not computable by the standard Turing machine.
10. We can use a multi-tape machine to design a universal Turing machine that can simulate the computation of any other Turing machine.
11. A Turing machine whose computation is to be simulated by the universal machine is first encoded as a binary string. This is equivalent to compiling a program into executable code.
12. Turing's universal machine is the basis for all modern stored-program computers that load an executable program into memory and then run the program. This way, the universal Turing machine can compute anything that any other Turing machine can.
13. Although there is no algorithm to design a Turing machine, the set of *mantras* discussed in this chapter help us in designing an accurate Turing machine for a given problem.

10.12 Exercises

A. Construct a Turing machine to perform each of the functions given below. Also show an accepting sequence of configurations for each of the following example strings.

1. Accept all odd palindromes over $\{a, b\}$ (where the symbol at the middle is also a or b). Show an accepting sequence of configurations for the input *baaaaab*.
2. Accept the language of the RegEx: $(ab + ba)(a + b)^*(ba + ab)$. Show how it rejects *ababaa*.
3. Accept all positive binary numbers divisible by 4.
4. Separate a given string into two equal halves by finding the midpoint of the given string and inserting a blank at that point. For example, given *abbaab*, the output shall be *abb aab*.
5. Break a given string into three equal parts. Show an accepting sequence of configurations for the input *abbbaabba*, the expected output being *abb baa bba*.
6. Match properly nested parentheses. Show an accepting sequence of configurations for (((()(()))())).

7. Match properly nested XML tags (as described in Exercise 71 in Chapter 7 and Exercise 27 in Chapter 8), assuming that the only possible tag names are $\{a, b, c, d\}$.
8. Sort the symbols in the input string: for example, given *babcacba*, the output shall be *aaabbbcc*. The alphabet is $\{a, b, c\}$.
9. Enhance Example 10.2 by including addition with positive or negative numbers, that is, strings over the alphabet $\{a, b, c, -\}$ of the form:
   (a) $a^n b^m c^k$ where $k = n + m$ and both numbers are positive; for example, *aabbcccc*.
   (b) $-a^n b^m c^k$ where $k = m - n$ if $m \geq n$ and the first number is negative; for example, *–aabbbbcc*.
   (c) $-a^n b^m –c^k$ where $k = n - m$ if $n > m$ and the first number is negative; for example, *–aaaabb–cc*.
   (d) $a^n –b^m c^k$ where $k = n - m$ if $n \geq m$ and the second number is negative; for example, *aaaa–bbcc*.
   (e) $a^n –b^m –c^k$ where $k = m - n$ if $m > n$ and the second number is negative; for example, *aa–bbbb–cc*.
   (f) $–a^n –b^m –c^k$ where $k = n + m$ and both numbers are negative; for example, *–aaa–bbb–cccccc*.
10. Find the maximum of two positive binary integers. For example, given 1001 11000 as input, the output is the second number, which is indicated by writing a 1 onto the tape: 1001 11000 1 (similarly a 0 if the first number is the maximum of the two or if the two numbers are the same).
11. Reverse a given string. Show a sequence of configurations for the input strings *ababbb*.
12. Copy a given string, that is, the tape should contain a second copy of the input string separated by a single blank cell from the given string.
13. Accept strings of the form $a^n b^n c^n d^n$. Show an accepting sequence of configurations for the input *aaabbbcccddd* and show how *aaabbbccccdd* is rejected.
14. Check if the first part of the string is present as a sub-string of the second part of the string, the two parts being separated by a single blank cell. The output shall be a 0 or a 1 indicating the absence or presence of the sub-string in the given string, respectively. Show the computation for the input 011 001011011.
15. Find the index of a symbol in a given string, that is, given a symbol followed by a blank and a string, for example, *a bcbaaba*, it finds the first position where the given symbol occurs in the string, for example, 4 in the example, and outputs the number in unary. The final contents of the tape in the example shall be *a bcbaaba* 1111.
16. Divide a given number by two in the unary number system. The quotient and the remainder (separated by a blank) should be written on the tape when the machine halts. For example, given 1111 on the tape, the output should be 11 (i.e., 4/2 = 2 and there is no remainder). As another example, given 1111111, the output should be 111 1 (i.e., 7/2 = 3 and the remainder is 1).
17. Perform insertion sort, that is, given a string over {0, 1, 2, 3}, write the symbols in increasing order on the tape by repeatedly inserting the next symbol in its proper place in the sorted output. For example, given 3021312, the output shall be 0112233 (with the intermediate contents of the tape being 3, 03, 023, 0123, 01233, 011233, 0112233).
18. Extract all symbols in odd-numbered positions. If the input is a string $w = w_1 w_2 w_3 \ldots w_n$ where each w_i is a or b and n is an even number, the machine halts with the contents of the tape being $w_1 w_3 w_5 \ldots w_{n-1}$.

19. Implement a simple tokenizer: given a string with single blank spaces separating other symbols, count the number of tokens in the string and output this number in unary. For example, given

ab bc abc bca abc a b c

the output shall be

ab bc abc bca abc a b c 11111111

20. Interleave two strings. Given two strings of equal length (separated by a single blank cell), such as

$$a_1a_2a_3\ldots a_n\, b_1b_2b_3\ldots b_n$$

the machine halts after writing on the tape an interleaved string composed of the first elements of the two strings, followed by the second elements of the two strings, and so on, ending with the last elements of the two strings, in that order:

$$a_1b_1a_2b_2a_3b_3\ldots a_nb_n$$

B. Make changes to the given Turing machine.

21. Modify the Turing machine shown in Fig. 10.13 to accept *wcw* where $w = (a + b)^*$ and c is a special character separating the two parts.

22. Show what happens when the Turing machine in Fig. 10.9 is given the input 000011112211.

C. Debug and fix the following Turing machines.

23. The Turing machine to compute the weight of a binary string, that is, the number of 1s in the string (to be written as a unary number on the tape) shown in Fig. 10.15.

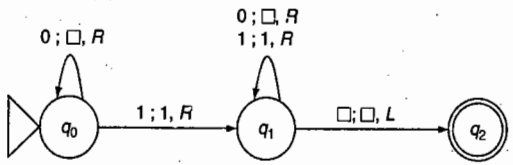


FIGURE 10.15

24. The Turing machine to accept a given binary string if it has an unequal number of 0s and 1s shown in Fig. 10.16.

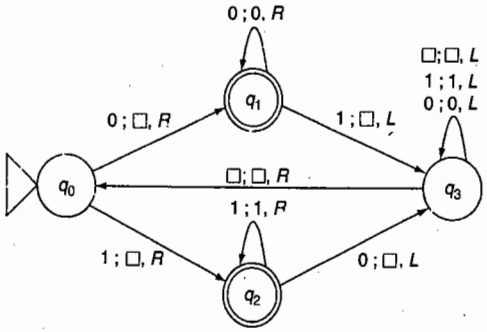


FIGURE 10.16

25. The Turing machine to add 2 to a given binary number shown in Fig. 10.17.

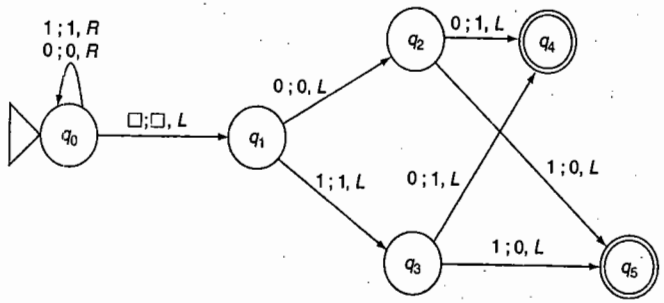


FIGURE 10.17

26. The (incomplete) Turing machine (Fig. 10.18) to search for its first argument in its third argument and replace every occurrence of it with its second argument, for example, given

b p bababbabbabba

the ouptput is

papappappappa

the alphabet being $\{a, b, p\}$.

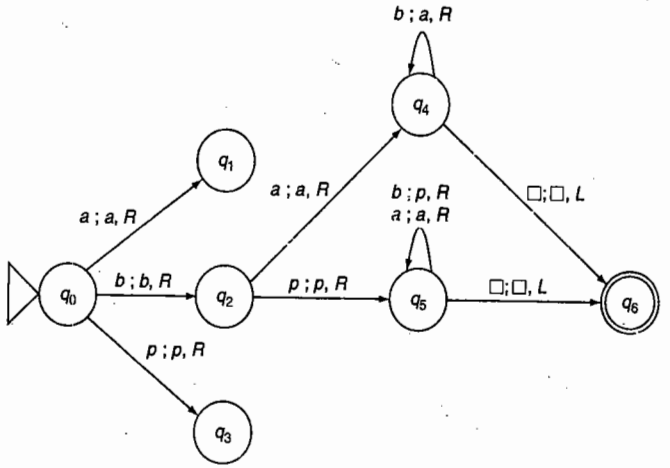


FIGURE 10.18

D. Show that each of the following variations of a Turing machine is equivalent to the standard Turing machine.

27. *Multi-track machine*: The tape has multiple parallel tracks. *Hint*: Interleave the contents of the multiple tracks on a single tape (similar to Exercise 20 above).

28. *Multi-tape machine*: There is more than one tape (as in the universal Turing machine). *Hint*: Interleave the contents of the multiple tapes on a single track (similar to Exercise 20 above).

29. *Multi-dimensional tape machine*: The tape has two, three, or higher dimensions. *Hint*: Linearize or serialize the contents of the multi-dimensional tape on a single, one-dimensional tape (the same way address calculations are done in a multi-dimensional array in a modern programming language).
30. *One-sided tape machine*: The tape is semi-infinite, with a definite left-most cell. *Hint*: Split it into a two-track tape with the second track serving to make up for the missing infinitely long tape to the left.
31. *Turing machine with stay-option*: Where it is not mandatory that a transition makes the head move left or right.
32. *Non-deterministic Turing machine*: Where there can be more than one transition for the same current state and the same symbol on the tape. *Hint*: Use the idea of a backtracking algorithm.

E. Universal Turing machine.

33. Encode the Turing machine you constructed in Exercise 1 as a binary string.
34. Show how a universal Turing machine can simulate the computation of the above encoded Turing machine for the input string *abbba*.
35. Encode the unary adding Turing machine from Example 10.2 (Fig. 10.5) as a binary string. Also construct a Turing machine for subtracting a given (larger) unary number from another (smaller) unary number. Encode this too as a binary string. Show that the same universal Turing machine can simulate the computations of both the adding and the subtracting Turing machines.

Chapter Eleven

The Chomsky Hierarchy

Learning Objectives

After completing this chapter, you will be able to:

- Learn the difference between countably infinite and uncountably infinite sets.
- Learn the diagonalization technique to show that a language is uncountable.
- Learn how to properly enumerate a language.
- Learn why some languages are not enumerable.
- Understand the difference between acceptance and membership in a language.
- Understand why some languages are enumerable but not recursive.
- Learn to construct a context-sensitive grammar for a language.
- Learn to construct a pushdown automaton with two stacks.
- Learn how unrestricted grammars work.
- Learn how linear-bounded automata work.
- Learn how all formal languages can be arranged in the Chomsky Hierarchy.

This chapter is devoted to completing the big picture of formal languages, grammars and computing machines, known as the Chomsky Hierarchy, which is being painted throughout the book. In doing so, two gaps must be filled: context-sensitive languages (CSLs) and their context-sensitive grammars (CSGs) and the minor class of deterministic context-free languages (DCFLs) and deterministic pushdown automata (DPDAs). Before that, the chapter establishes the classes of recursively enumerable and recursive languages. In order to do this, the difficult concepts of enumeration and diagonalization are explained. The chapter also shows the distinction between acceptance and membership functions for languages, thus setting the stage for the study of the halting problem and undecidability in the following chapter.

11.1 Languages, Grammars and Machines

We have studied two main classes of formal languages in earlier chapters: regular and context-free languages (CFLs). We have also seen that computing machines that correspond to CFLs, namely pushdown automata (PDAs), are more powerful than finite automata which correspond to regular languages. In the last chapter, we saw that Turing machines were even more powerful than PDAs because

they could solve problems that we could not earlier solve using PDAs. At this point, therefore, we raise the following questions about larger classes of formal languages:

1. What class of languages corresponds to Turing machines?
2. What kind of grammars are equivalent to Turing machines?
3. Are there languages beyond the languages of Turing machines?
4. If Turing machines are the most powerful computing devices, are there classes of languages in between CFLs and those of Turing machines?
5. Are there types of computer memory more powerful than a PDA's stack but less powerful than a Turing machine's tape that correspond to these intermediate languages?

11.2 Recursively Enumerable Languages

What is the language of a Turing machine? It is the set of strings over the alphabet of the Turing machine which is accepted by the machine. Consider now the set of all possible Turing machines (an infinite set) and the set of all languages that they accept. The Church–Turing thesis (see Sec. 10.7) says that anything that is computable can be computed by a Turing machine. Therefore, this set of languages is an interesting one since it corresponds to the set of all computable functions (see Sec. 12.6 for a precise definition of computable functions). This is also equivalent to the set of all valid computer programs that one could ever write. We call this class of languages *recursively enumerable languages*. This is the largest class of formal languages that are "computable."

11.3 Counting Alphabets, Languages and Computing Machines

Why are the languages of Turing machines called *recursively enumerable?* To understand this, we need to look at ways of enumerating or counting elements of different kinds of sets. The smallest set with which we are usually concerned is the alphabet: the finite set of symbols used in a computing machine or in a formal language. A language, being a set of strings over the alphabet, is often infinite since there is no limit on the length of its strings. In fact, we know that a language with a limit on the maximum length of its strings is finite and that a finite language is regular and thus rather simple. The simplest language over an alphabet Σ is its closure Σ^*, the set of all strings over the alphabet including strings of infinite length wherein symbols from the alphabet can be repeated any number of times in a string. As we know, this is a regular language. Any other language over the same alphabet is a proper subset of this simple language. How many such distinct languages can be there over a given alphabet? Since each language is a subset of the infinite language Σ^*, the set of all languages over the alphabet is the power set of the language Σ^* (i.e., the set of all subsets of Σ^*, sometimes written 2^{Σ^*})! These relationships and sizes are illustrated in Fig. 11.1.

Are there more elements in this power set than the number of strings in the infinite language Σ^*? This is an important question in understanding the hierarchy of formal languages. To answer this, although both are infinite sets, we need to distinguish between two kinds of enumeration of infinite sets. Does it make sense to say that one infinite set has "more elements" than another infinite set?

But first, let us ask how many Turing machines are there. We know from our study of the universal Turing machine in the last chapter that a Turing machine can be encoded as a binary string. The binary

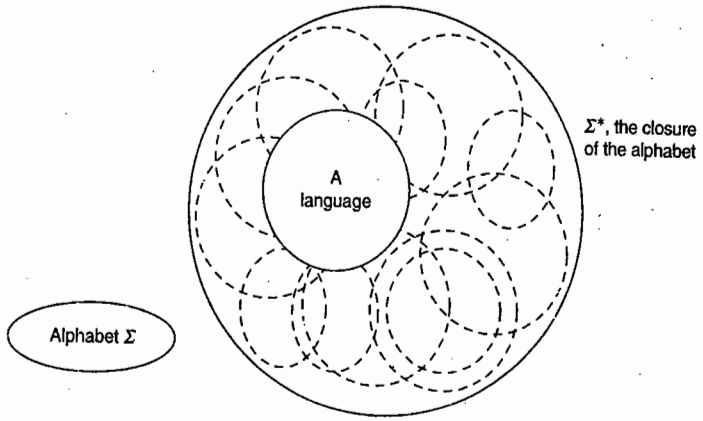


FIGURE 11.1 A language is a subset of the largest language Σ^*, an infinite set. The set of all languages is the set of all "circles" (each of which can be an infinite set) which we can draw inside the outer circle.

alphabet is {0, 1} and the set of all binary strings $(0 + 1)^*$ is an infinite set. Since every Turing machine can be encoded as a binary string but not all binary strings are valid encodings of Turing machines, we can conclude that the set of Turing machines is a proper subset of the set of all binary strings! Thus, the set of all Turing machines is a formal language, a proper subset of the language $(0 + 1)^*$.

What about the set of all formal languages over {0, 1}? Let us learn how to enumerate such sets.

11.4 *The Idea of* Enumeration

Consider the set of all natural numbers $\mathbb{N}$ (i.e., the set of non-negative integers). Can we write a program which will enumerate all these numbers? That is, can we build a Turing machine for enumerating natural numbers, for example by writing each successive number onto its tape? Obviously this program will never terminate because the set is infinite. Barring a "hardware" failure, this machine will not halt; it stays in an infinite loop. However, our consideration is, given a particular finite number n, no matter how large it is, it will be written in finite time. Such a set is said to be *enumerable* (or *denumerable*):

An *enumerable* (or *denumerable*) set is one in which any given element of the set can be generated by a Turing machine in finite time. It is also a set whose elements can be placed in a one-to-one correspondence with the set of natural numbers $\mathbb{N}$, so that we can index each element of the set as the ith element if it corresponds to the natural number i (or assign it the ordinal number i). Such a set is also said to be *countably infinite.*

For example, the set of all non-negative numbers divisible by 3 is enumerable. If we take any element of this set and divide it by 3, we get its index, thus creating a one-to-one mapping from this set to the set $\mathbb{N}$ (as shown in Fig. 11.2). This mapping can also be used to argue that these two infinite sets have the same number of elements, that is, they have the same cardinality, or, there are as many non-negative numbers divisible by 3 as there are all non-negative whole numbers! Note that this equality holds only

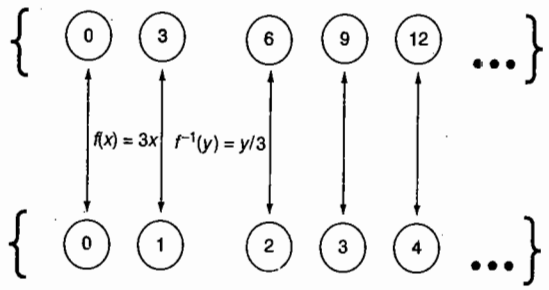


FIGURE 11.2 Mapping from the set of numbers divisible by 3 to the set of natural numbers to show that the first set is enumerable. The two sets also have the same number of elements.

because both sets are infinite; if there was a maximum value for the numbers, then certainly there would be more non-negative numbers than those divisible by 3.

Are there sets that are not enumerable (or *non-denumerable*) for which no such enumerator Turing machine can be constructed? Are there sets that cannot be placed in one-to-one correspondence with the set of natural numbers $\mathbb{N}$? The answer is yes!

Let us consider, for example, the set of all real numbers $\mathbb{R}$ between 0.0 and 1.0. This is also an infinite set:

$$0.0, 0.0000\ldots000001, \ldots$$

Immediately, we have trouble listing the second element of the set (if $\mathbb{R}$ is sorted in increasing order) unless there is a limit on the precision of the numbers. If precision is limited, there is a modification to the above listing procedure which will make real numbers enumerable: write the numbers ordered by length first and then by value (i.e., all the numbers that have one decimal place only, then those with two decimal places, and so on):

$$0.0, 0.1, 0.2, 0.3, \ldots 0.9, 1.0, \ldots\ 9.9, 0.01, 0.02, \ldots$$

If precision is unlimited, a given real number with unlimited precision will never be written on the tape in finite time in any attempt to enumerate the real numbers. For example, if we start enumerating all real numbers in increasing order, the machine will never reach the number 0.01 because there are infinitely many real numbers between 0.00 and 0.01. Moreover, the number 0.01 is the same as 0.009999... and therefore the above modification of ordering by length will also not work. Similarly, in a binary representation of real numbers, the number 0.10000... is the same as the number 0.01111... (since $1/2 = 1/4 + 1/8 + 1/16 + \cdots$).

If an infinite set is like the set of real numbers $\mathbb{R}$, we say that it is *not enumerable* or *non-denumerable*. Such sets are also called *uncountably infinite*. Intuitively, such sets have "too many elements." In a countably infinite set, there are a finite number of elements between any two elements (in the order of enumeration). In an uncountably infinite set, on the other hand, there are an infinite number of elements between any two elements of the set.

There is a formal proof by contradiction which shows why the set $\mathbb{R}$ is not enumerable. The proof uses a technique known as Cantor's *diagonalization*. We use this technique in the next section to show first that the set $\mathbb{R}$ is not enumerable and later to show that there are some languages for which

there is no Turing machine which accepts the language. The diagonalization technique is also used to prove well-known results in mathematics such as Russell's paradox in set theory and Gödel's Incompleteness Theorem.

11.5 *The Idea of* Diagonalization

Let us explore the idea of Cantor's diagonalization. It goes as follows. First we assume that the given set $\mathbb{R}$ is enumerable. Our objective is to set up a contradiction to prove that our assumption is wrong. Since $\mathbb{R}$ is enumerable, let us enumerate all its elements in a tabular form (see Fig. 11.3) where each row is a real number and each column is a decimal place. This table has an infinite number of rows and an infinite number of columns. Nevertheless, we can now talk about the 1st real number, 2nd real number and so on. This would be the output of enumeration if real numbers were enumerable.

Consider the left-to-right diagonal which contains the 1st decimal place of the first real number, 2nd decimal place of the second real number and so on. If we read this diagonal as a real number, we get a

Decimal places

Each row is a real number

i	1	2	3	4	5	6	7	8	9	10	11	12	...	...	...
1st	**1**	0	0	0	0	0	0	1	0	0	0	0	0	0	0
2nd	0	**0**	0	0	1	0	0	0	0	0	1	0	0	0	0
3rd	0	1	**0**	0	0	0	0	0	0	0	0	0	1	0	0
4th	0	0	0	**0**	0	0	1	0	0	0	0	0	0	0	0
5th	0	0	0	0	**1**	0	0	0	0	1	0	0	0	0	0
6th	0	1	0	0	1	**0**	0	0	0	0	0	1	0	0	0
7th	0	0	0	1	0	0	**1**	0	1	0	0	0	0	0	0
8th	0	0	0	0	0	0	0	**0**	0	0	0	0	0	0	1
9th	0	0	0	1	0	0	0	0	**0**	0	0	0	0	0	0
10th	1	0	0	0	0	0	0	0	0	**0**	0	0	0	0	0
11th	1	0	0	0	0	0	0	1	0	0	**0**	0	0	1	0
12th	0	0	1	0	0	0	0	0	0	0	0	**1**	0	0	0
...	0	0	0	0	0	0	1	0	0	0	0	0	**1**	0	1
...	0	0	1	0	0	0	0	0	0	0	0	0	0	**0**	0
...	0	0	0	1	0	0	0	1	0	0	1	0	0	0	**0**

FIGURE 11.3 Diagonalization of real numbers where real numbers are presumed to be enumerated in some order. The diagonal $r_d = 100010100001100\ldots$.

number r_d = 100010100001100.... We now complement it to invert the elements of the diagonal. If the numbers are binary, we take their binary complements; if they are decimal, we choose some other digit between 1 and 8. The resulting inverted number r_d^{-1} = 011101011110011... is also, obviously, a real number.

The question we now address is whether the inverted diagonal number r_d^{-1} already exists in the enumeration, that is, whether it is the same as one of the rows in the table we have constructed. We see that it is not the same as the first row because it differs from it in the first decimal place. It is not the same as the second real number because it differs from it in the second decimal place, and so on. It differs from every enumerated real number in the corresponding diagonal element (i.e., in the *i*th decimal place). But we started with the assumption that every real number in $\mathbb{R}$ has been enumerated in the table. Yet, we now have on our hands a new real number which has not been enumerated. This is a contradiction. The diagonalization argument has shown us that our assumption must have been wrong. Therefore, the set of real numbers $\mathbb{R}$ is not enumerable.

We may wonder what happens if we apply the idea of diagonalization to integers. Since positive integers are in fact enumerable, we must not be able to set up a contradiction. In other words, the inverted diagonal must in fact be the same as one of the enumerated numbers. Since we take complements of diagonal elements to ensure that the new number differs along all diagonal entries, this is not possible. Thus, it must be the case that we cannot construct the inverted diagonal at all. In fact, we encounter difficulties when we try to represent whole numbers in an infinite-by-infinite table since in any such attempt, some of the rows will not even be whole numbers.

11.6 Diagonalization: Not All Languages are Recursively Enumerable

How is this relevant to the study of Turing machines and formal languages? First, we observe that Σ^* is countably infinite. Suppose we write the enumeration program (i.e., a Turing machine or an algorithm) for Σ^* such that it prints strings in this language alphabetically (or, more generally, in lexicographic order). The problem with such an enumeration procedure, with $\Sigma = \{a, b, c\}$ for instance, is that it never prints the string "*bb*":

$$\lambda, a, aa, aaa, aaaa,$$

This is not a proper way to enumerate the set Σ^*. Instead, if we modify the enumeration procedure so that strings are ordered (in *proper order*) first by increasing length and then alphabetically, any given string such as "*bb*" will be printed in finite time. For example, if $\Sigma = \{a, b, c\}$, a proper enumeration of Σ^* is

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

If we can enumerate any such set, we can immediately talk about the *i*th element of the set and thereby map the elements to the set of natural numbers. Given any string of finite length w in Σ^*, the proper enumeration procedure will print w in finite time.

Any formal language over the alphabet is a subset of Σ^*, albeit often an infinite subset. Surprisingly, some of the subsets are not enumerable! We can show this using the same idea of diagonalization

that we used above to show that the set of real numbers is not enumerable. But let us first understand intuitively why not every language is enumerable.

How many formal languages are there over an alphabet Σ? This is the number of subsets of Σ^*, or, in other words, the size of the power set 2^{Σ^*}. Σ^* is countably infinite but the set of all languages is uncountably infinite. Just as there are many more real numbers than there are integers, so also there are many more languages than there are strings in Σ^*. It does not make any sense to talk about the *i*th language because languages are not enumerable.

From our study of universal Turing machines (Sec. 10.9), we know that we can encode all the transitions of a Turing machine as a binary string. We can enumerate all possible binary strings and filter out those that are not valid encodings of a Turing machine according to the scheme used for encoding. Since we already know that every Turing machine can be encoded as a unique binary string, the total number of possible Turing machines is less than the size of Σ^* for the binary alphabet. As such, there are only a countably infinite number of Turing machines in the world but there are an uncountably infinite number of formal languages. It is not possible to establish a one-to-one correspondence between all languages and all Turing machines. There are many formal languages for which there is no corresponding Turing machine. In other words, the set of *recursively enumerable* languages is a proper subset of the set of all formal languages (over a given alphabet). Therefore, *there must exist languages that are not recursively enumerable.* Fortunately, most interesting and practical languages are recursively enumerable; they are computable and decidable (see Sec. 12.6).

Turing machines are enumerable! Interestingly, since the set of all Turing machines is countably infinite, it makes sense, in whatever weird sense, to talk about the *i*th Turing machine, the *i*th algorithm, or in other words, to enumerate all computer programs and to refer to the *i*th program that one could ever write!

The statement that *not all languages are recursively enumerable* can be proven by applying the idea of diagonalization. Consider the closure of the alphabet, that is, Σ^*. Since this set is enumerable, let us number its strings $s_1, s_2, s_3, \ldots s_i, \ldots$. Let us begin the proof-by-contradiction by assuming that the set of all formal languages over Σ is enumerable. That is, the languages belonging to 2^{Σ^*} can also be numbered $L_1, L_2, L_3, \ldots L_j, \ldots$. As in the case of diagonalizing real numbers, let us construct a two-dimensional matrix with an infinite number of rows, the rows being an enumeration of the languages from the power set (which is assumed to be enumerable). One element per row represents a language L_j and each column is a string s_i, as shown in Fig. 11.4. The rows of this matrix can also be seen as an enumeration of all Turing machines with row L_j corresponding to the language of the *j*th Turing machine M_j.

The binary element of the table in row j and column i indicates whether string s_i belongs to language L_j. Now consider the left-to-right diagonal. If we read it as representing another language L_{Diag}, it is the language of all strings s_i that belong to the *i*th language L_i. This language $L_{\text{Diag}} = \{s_1, s_5, s_7, s_{12}, s_{13}, \ldots\}$ has something in common with every language: it either has the same string as the other language (if the diagonal element is a 1) or it does not have the same string that the other language does not have (if the diagonal element is a 0). In fact, L_{Diag} must be the same as one of the rows, that is, one of the enumerated languages (L_{10} in the figure happens to be the same as the diagonal language L_{Diag}). We can also think of L_{Diag} as the set of all strings s_i which belong to the language L_i of the *i*th Turing machine since each row L_i can be seen as the language of the *i*th Turing machine.

Strings in Σ^*

Each row is L_j, the language of Turing machine M_j	s_1	s_2	s_3	s_4	s_5	s_6 ($i=6$)	s_7	s_8	s_9	s_{10}	s_{11}	s_{12}	...	...	...
1st	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0
2nd	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0
3rd	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0
4th	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
5th	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0
6th	0	1	0	0	1	0	0	0	0	0	0	1	0	0	0
7th	0	0	0	1	0	0	1	0	1	0	0	0	0	0	0
$j=$ 8th	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1
9th	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
10th	1	0	0	0	1	0	1	0	0	0	0	1	1	0	0
11th	1	0	0	0	0	0	0	1	0	0	0	0	0	1	0
12th	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0
...	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1
...	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
...	0	0	0	1	0	0	0	1	0	0	1	0	0	0	0

FIGURE 11.4 Diagonalization of formal languages. The diagonal language $L_{\text{Diag}} = \{s_1, s_5, s_7, s_{12}, s_{13}, \ldots\}$.

This language L_{Diag} is computable and recursively enumerable because we can construct a universal Turing machine that simulates the behavior of the ith Turing machine, given s_i. In other words, there exists a Turing machine for L_{Diag}.

We now construct a new language, a new row; we do this so that the new row cannot be identical to any existing row. An easy way to do this is to complement the elements along the diagonal (hence the name diagonalization). This new element is not the same as any of the enumerated elements (i.e., rows) because it differs from the ith row in its ith column. This new language is the complement of the language L_{Diag}, that is, a new language

$$L_{\text{Non-RE}} = \{s_2, s_3, s_4, s_6, s_8, s_9, s_{10}, s_{11}, s_{14}, s_{15}, \ldots\}$$

This language has something different from every other language!

1. If a language L_i (of the ith Turing machine) contains the string s_i, the language $L_{\text{Non-RE}}$ does not contain it.
2. If L_i does not contain s_i, then surely $L_{\text{Non-RE}}$ contains it.

In other words, it differs from every other language along their diagonal elements. It must be a new language, not the same as any of the enumerated languages. This is a contradiction, since we began by enumerating all languages over the alphabet. Therefore, our assumption that the languages are enumerable must be wrong.

Most languages are not recursively enumerable! Most of them are not computable! However, this is not so bad, in reality, because all these innumerable *un-computable* languages are not interesting. In fact, by definition, they can have no direct description, for, if we could describe them directly and concisely, the description can be converted to an algorithm, an enumeration procedure, a grammar or a Turing machine. We can get a glimpse of them only indirectly, via an obscure mathematical construction, namely, diagonalization!

The language $L_{\text{Non-RE}}$ is an example of a language that is not *recursively enumerable*. That is, it has no Turing machine that accepts it. We cannot write an enumeration procedure for its strings. It is not a computable language!

We can now define a language L as recursively enumerable if there exists a Turing machine M such that

$$L = M.\text{language}$$

that is, given any $w \in L$, M accepts w and halts. This implies, in a finite number of transitions, M processes all of w and halts in a final state of M.

Why are they called *recursively* enumerable languages? This term comes from the mathematical theory of recursive functions: an alternative but equivalent formalism for modeling computation known as *primitive recursive functions* and *μ-recursive functions* (see Sec. 12.8).

11.7 *The Ideas of* Acceptance and Membership

Let us now consider subsets of recursively enumerable languages. We begin by understanding the difference between *acceptance* and *membership* in a set – an important distinction that leads to the very interesting issue of the *halting problem* of a Turing machine. Given a string w, what does a Turing machine do? If w belongs to M.language, the machine will *accept* w (as per the definition of the language of a Turing machine). For example, given a grammatical sentence in English or a computer program with no syntax error, the corresponding processor (e.g., parser, compiler or interpreter) for those languages will accept it. Accepting a string means reaching *and halting* in a final state, in a finite amount of time and after a finite number of transitions from the start state. Halting means there must be no further transitions out of that state.

What happens if the input string w does not belong to M.language? For example, what if the English sentence is ungrammatical or the computer program has errors? Given strings that do not belong to the language, what does the machine M do?

If it halts in a non-final state, the string is rejected. Another possibility is that the Turing machine will not halt at all, that is, it goes into an infinite loop and never produces an output. It will never give an answer. We will never know if it eventually reaches a final state or a non-final state. This is called the *halting problem in Turing machines*. We will study this interesting limitation of computing machines in more detail in the next chapter (see Sec. 12.3).

The first kind of machine, which always halts, implements a *membership function*. It is called membership because if w belongs to M.language, M will halt in a final state; else it halts in a non-final

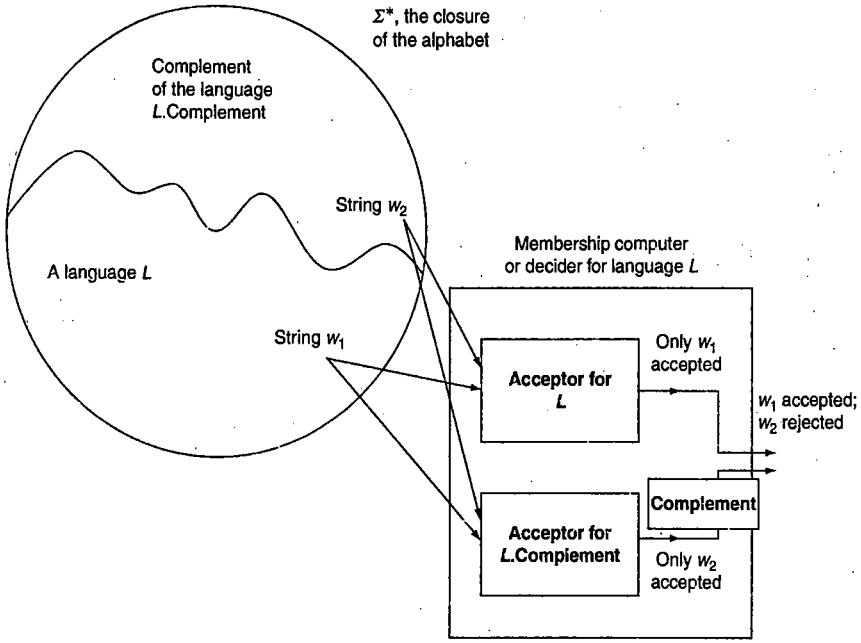


FIGURE 11.5 A language and its complement, subsets of the infinite regular language Σ^*, and their acceptor and decider machines.

state. Thus, the Turing machine is computing membership of strings in the language, given any string over the alphabet no matter whether it belongs to the language of the machine or not. Such a machine is called a *decider machine* (see Fig. 11.5).

For some recursively enumerable languages we cannot build such a Turing machine. We can only build an acceptor for such languages. An acceptor is the second kind of machine: it accepts the input string if it is in the language, that is, halts in a final state for all good inputs. Its behavior is unpredictable for bad inputs; it may or may not halt. There is no guarantee that it will halt for all inputs.

For such recursively enumerable languages, only acceptors exist; no membership function can be defined for them. It is not possible to compute membership for such languages. In other words, given a member of the language, the machine can recognize it but it is unable to handle inputs that are non-members.

By its very definition, every recursively enumerable language has an enumeration procedure or a Turing machine that can enumerate its members. Given any string of finite length in the language, such a machine can always accept the string in finite time. Is it possible to compute membership for every recursively enumerable language? The answer is no. Why is it so?

Consider the example of our diagonal language L_{Diag} and its complement $L_{Non\text{-}RE}$. L_{Diag} is recursively enumerable and has an acceptor Turing machine (because it corresponds to one of the enumerated Turing machines). On the other hand, $L_{Non\text{-}RE}$ is not recursively enumerable and does not have a Turing

machine that can accept its strings. Moreover, these two languages are complements of each other. We have in L_{Diag} a recursively enumerable language whose complement is not recursively enumerable. This is the same as saying that L_{Diag} has an *acceptor* but not a *decider* (i.e., a Turing machine which computes its membership function). Why? Because, if it had a membership function, we could construct a *decider* Turing machine to compute its membership function. Such a machine, if it existed, would accept member strings (i.e., those that are in L_{Diag}) and reject non-member strings (i.e., those that are in $L_{Non\text{-}RE}$) but would always halt for any input. What if we modified this machine by interchanging its final and non-final states (the same way that we modified finite automata in Sec. 6.1.4)? We would then have a membership computer for the language $L_{Non\text{-}RE}$. This is impossible since we have shown that no Turing machine corresponds to this non-enumerable language.

Thus, if a language has a membership function, its complement also has a membership function. We call such languages *recursive* languages (or *Turing decidable languages*). There are other *recursively enumerable* languages (e.g., L_{Diag}) which are not *recursive*. Another example of such a recursively enumerable language which is not recursive is the set of all valid inputs to every possible computer program. In theory, we could enumerate not only every possible program but also each of its valid inputs, but we cannot, given an arbitrary input to an arbitrary program, say whether the input is valid for that program or not. Thus, *recursive* languages constitute a proper subset of *recursively enumerable* languages (which are also known as *semi-decidable languages*).

Here is a summary of what we have established so far:

1. Not all formal languages are recursively enumerable; those that are, are countably infinite and correspond to an enumeration of all possible Turing machines.
2. A proper subset of recursively enumerable languages, known as recursive languages, has membership functions and decider machines.
3. The complement of every recursive language is also recursive but the complement of a recursively enumerable language that is not recursive is not recursively enumerable.
4. We know all of the above from the diagonalization argument.
5. Every context-free language (and therefore every regular language) is a recursive language but not every recursive language is context-free.

How do we know the last statement above that context-free languages are recursive? They are, in fact, recursive because we know how to construct a decider machine for every context-free language: *machines called pushdown automata which accept every string in the context-free language and halt and reject every string that is not in the language*. We can always consider a pushdown automaton (PDA) as a Turing machine which restricts its tape usage to stack-like behaviors. Thus, context-free languages must be recursive. At the same time, we know of languages such as $a^n b^n c^n$ which are not context-free (see Example 9.1 in Chapter 9). We will see below that such languages are also recursive. They show us that context-free languages are a proper subset of recursive languages.

Imagine what would happen if a programming language was like L_{Diag}. If you write and submit a syntactically correct program, then the compiler would accept it. If you made any error at all in your program, the compiler may go into an infinite loop and never produce any output! We surely do not

want such a language! How do we design a programming language so that it does not suffer from such problems? We better study the subset of languages for which a membership function is available, namely, recursive languages.

11.8 Recursive Languages

We define *recursively enumerable* languages as those that have Turing machines *accepting* them and *recursive* languages as those that have Turing machines *deciding* them. In other words, non-members of recursively enumerable (RE) languages may or may not be handled well but must be handled well in the case of recursive languages. We can relate this to the complement of a language: strings in the complement of an RE language may send the machine on an infinite loop, but member strings of the complement of a recursive language will be processed in finite time to make a decision that they are not in the original language.

You may want to remember that if a language and its complement are both RE, then, in fact, they are both recursive. The set of languages where both a language L and L.Complement are RE are called recursive languages. For both it is possible to construct a Turing machine acceptor. By combining the two acceptors, we get a decider or membership function computer for either language. The decider accepts good strings in L and halts (in a finite amount of time) and rejects all bad inputs (i.e., strings which are in L.Complement) as shown in Fig. 11.5.

We also note that the set of Turing machines that can compute membership functions of (recursive) languages is a proper subset of the set of all Turing machines (which are acceptors for RE languages). We will study the difference between these two kinds of Turing machines in more detail when we look at the halting problem in the next chapter (see Sec. 12.3).

A language such as L_{Diag} above that is RE but its complement $L_{\text{Non-RE}}$ is not, is a very peculiar language:

1. **It has an enumeration procedure:** Any finite string in this language can be enumerated (i.e., printed, displayed, etc.) by a formal enumeration procedure in finite time (although the set itself is infinite and can never be exhaustively enumerated in finite time).
2. **However, it has no membership function:** It is not recursive, it is not decidable and its complement $L_{\text{Non-RE}}$ is not enumerable. Therefore, given an arbitrary input string there is no algorithm (i.e., Turing machine) that can tell you if the string belongs to the language L_{Diag}.

A summary of the relationships between classes of formal languages can be shown diagrammatically as in Fig. 11.6. Such a hierarchy of languages is known as the *Chomsky Hierarchy* in honor of Noam Chomsky who developed much of the theory of formal languages and grammars. In fact, his work led to the idea of a *computational grammar* that could be used to develop compilers and therefore high-level programming languages. His work also completely transformed the field of linguistics and led to the formation of an entirely new field of research known as *computational linguistics.*

In Fig. 11.6, the innermost circle represents the set of all regular languages which correspond to the set of machines known as *deterministic finite automata* (DFAs) and also to *non-deterministic finite automata* (NFAs). They can be described concisely using either regular expressions (RegEx) or regular

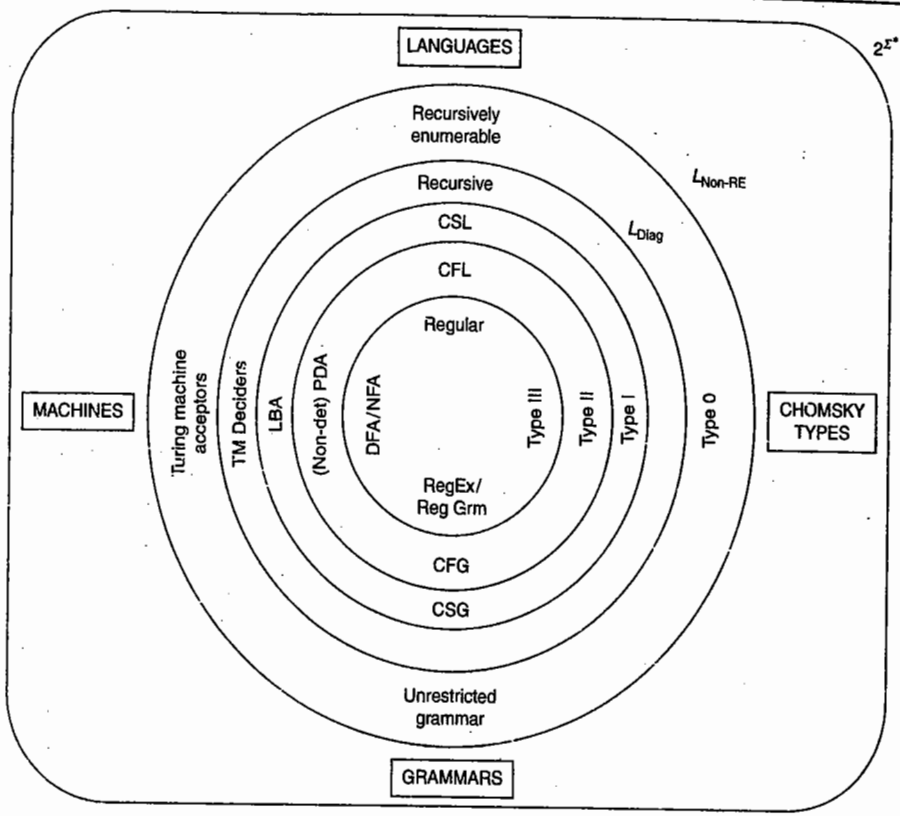


FIGURE 11.6 Chomsky hierarchy of formal languages.

grammars (i.e., either left-linear or right-linear grammars). This class of languages was called **Type III** by Chomsky.

The next outer circle represents the set of all context-free languages (CFLs) which are described by context-free grammars (CFGs) and are decided by machines called *(non-deterministic) pushdown automata* (NPDAs or just PDAs). This class is designated **Type II** in the Chomsky hierarchy.

Outside of this circle is the set of all context-sensitive languages (CSLs) which are described by context-sensitive grammars (CSGs), accepted by machines called *linear-bounded automata* (LBAs) and designated **Type I** (see Sec. 11.11). Moving further outwards, we come across the set of all recursive languages which are decided by Turing machine deciders. There is no named class of grammars or type number for this class.

The outermost circle represents the class of RE languages (e.g., L_{Diag}) which are described by unrestricted grammars and accepted by Turing machine acceptors. This largest class was called **Type 0** in the Chomsky hierarchy. Remember that they too constitute a proper subset of the set of all formal languages 2^{Σ^*} over a given alphabet Σ.

11.9 Context-Sensitive Languages and Grammars

We have seen that there are two classes of formal languages larger than context-free languages: recursive and recursively enumerable languages.

1. Are there recursive languages which are not context-free?
2. Is there any interesting class of languages in between context-free and recursive languages?
3. What if a language is not context-free?

We have already seen examples of such languages when we studied the Pumping Lemma for CFLs (see Sec. 9.6). What type of languages are they?

There is a class of language between CFLs and recursive languages known as *context-sensitive languages* (CSLs). We have seen using the Pumping Lemma that languages such as $a^nb^nc^n$ are not context-free because there are three sets of symbols to be counted and matched. In a context-free language (and in a PDA), we are able to match only two such counts since CFGs cannot control what happens to sentential forms in two places (other than the two extremes) at the same time (and PDAs lose the information on their stacks after matching the first two counts). However, we saw (in Chapter 10) that Turing machines had no trouble dealing with such languages since their reading heads can move around freely to any part of the tape.

1. Are there machines more powerful than PDAs but less powerful than Turing machines?
2. Do such machines correspond to an intermediate class of languages?

Consider a PDA having two stacks instead of one. As shown in Fig. 11.7, this machine can accept the language of $a^nb^nc^n$ as follows: push the as onto the first stack; for bs, pop as to match the number of

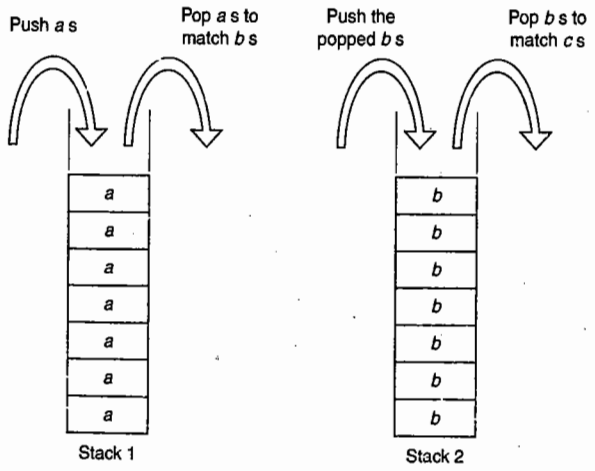


FIGURE 11.7 A PDA with two stacks deciding the language $a^nb^nc^n$.

as and bs; at the same time, also push b symbols onto the second stack; finally, for cs, pop bs from the second stack to match the number of bs and cs. Thus, we have a decider machine for a language that is not context-free. Therefore, it must be a recursive language.

Such languages are called context-sensitive languages and they constitute a superset of CFLs but still a proper subset of recursive languages (see Fig. 11.6). Before we see what could lie between context-sensitive and recursive languages, let us first understand why they are called context-sensitive, and, in fact, why context-free languages are called so.

11.10 *The Idea of* Context

As we saw in Chapter 7, CFGs can have just one variable on the left-hand side of every production rule. As such, their behavior in generating strings in the language is independent of the context, that is, the way we expand a variable in the sentential form is independent of what is to the left of it or what is to the right of it in the sentential form. Hence, they are called context-free grammars and context-free languages.

A grammar for a CSL must take context into account. The expansion of a variable in a production depends on the symbols on either side of the variable in the sentential form. This is accomplished by having more than one symbol on the left-hand side of the production. Such a grammar is called a *context-sensitive grammar* (CSG). In a CSG, any number of variables or terminal symbols can appear on either side of a produce rule, as long as the right-hand side of a production is not shorter than the left-hand side of the same production. In other words, sentential forms in a CSG cannot shrink by the application of a production to derive a string. As a consequence, sentential forms can never be longer than the final string at any point during a derivation (except where a variable disappears by expanding to λ).

EXAMPLE 11.1

Let us see how to construct a CSG for the language of $a^nb^nc^n$. We already know that we cannot construct a context-free grammar for this language.

1. Can we write a grammar by relaxing the restriction that the left-hand side of a production must be a single variable?
2. Can we somehow construct grammars in which variables can run around in the string (i.e., sentential form) and insert required symbols at the right places, the same way that the reading head of a Turing machine moves back and forth along its tape?

For example, whenever an a is to be inserted (using the capability of a context-free production), a variable should go to the end of bs in the sentential form, insert a matching b and c and come back to the starting point. This is not possible in CFGs but it is possible in the larger class of grammars known as context-sensitive grammars. Let us see how.

Suppose we start by saying

$$S \rightarrow abc$$

(the simplest string in the language). To insert additional sets of as, bs and cs, we introduce a variable A:

$$S \rightarrow aAbc$$
$$A \rightarrow \hat{\lambda}$$

Here A acts as the runner variable which runs to the point between bs and cs to introduce a matching b and a matching c.

How does it run from the point where a is to be inserted to the point between b s and c s?

This is where context-sensitive productions come in.

$$Ab \rightarrow bA$$

Repeated applications of this production will shift the variable A progressively through the bs till it crosses the last b. At this point, we are ready to insert b and c after verifying that we have indeed reached the end of bs (or the beginning of cs). However, we also need a mechanism to return to the point where the next a must be inserted. We introduce the new variable B which acts as the backwards runner:

$$Ac \rightarrow Bbcc$$

Notice that this production can fire only in the right context, that is, when the variable A has reached the end of bs (or just before the first c). This ensures that A is in the right place to insert the b and c symbols.

More context-sensitive rules make B run to the beginning of bs, insert the matching a and re-introduce the A variable:

$$bB \rightarrow Bb$$
$$aB \rightarrow aaA$$

The last production above inserts an a only in the right context, that is, when the variable B has come back all the way past the first b to the point just after the last a in the sentential form. This production also re-introduces the forward runner A so that more sets of matching as, bs and cs can be added. Let us see how this can derive the string $a^3b^3c^3$ from S:

$S \Rightarrow aAbc$	
$\Rightarrow abAc$	A moves one step to the right and reaches c
$\Rightarrow abBbcc$	Ac is replaced by $Bbcc$
$\Rightarrow aBbbcc$	B moves one step back to the left to reach the last a
$\Rightarrow aaAbbcc$	aB is replaced by aaA to complete the generation of $a^2b^2c^2$
$\Rightarrow aabAbcc$	A moves to the right again
$\Rightarrow aabbAcc$	A moves one more step to reach c
$\Rightarrow aabbBbccc$	Ac is replaced by $Bbcc$
$\Rightarrow aabBbbccc$	B starts to move to the left
$\Rightarrow aaBbbbccc$	B reaches the end of as

$\Rightarrow aaaAbbbccc$	Matching a is inserted
$\Rightarrow aaabbbccc$	The variable A is nulled since we do not want to generate further sets of as, bs and cs

It is apparent from the above example that the behavior of this CSG is very different from that of context-free grammars. The ability to move variables around came from the left-hand sides of productions being context-sensitive. The behavior of the variables A or B depends on the context. For example, A moves to the right if it is followed by a b, but not if it is followed by a c. Similarly, B moves to the left if it is preceded by a b but kills itself and inserts aA if it is preceded by an a in the sentential form.

EXAMPLE 11.2

Let us consider a second CSL, namely, all strings of the form ww where w is any string over the alphabet, that is, $w = \Sigma^*$. We have earlier seen (Example 9.2 in Chapter 9) with the help of the Pumping Lemma that this language is not context-free. Intuitively, this was because storing the first half of the string in stack memory reversed the string and we were unable to match the reversed string with the second half of the input. This language, as already noted, is very important from a practical point of view since this is what makes most programming languages context-sensitive: variable or function declarations constitute the first "w" in a program and their usage or implementation constitutes the second "w" which must match the first part.

First, let us see if it is possible to construct a PDA with two stacks which can accept this language, as shown in Fig. 11.8. The first stack is used to push the first half of the input string. The second stack is used to reverse the stored string by popping its symbols off the first stack and pushing them onto the

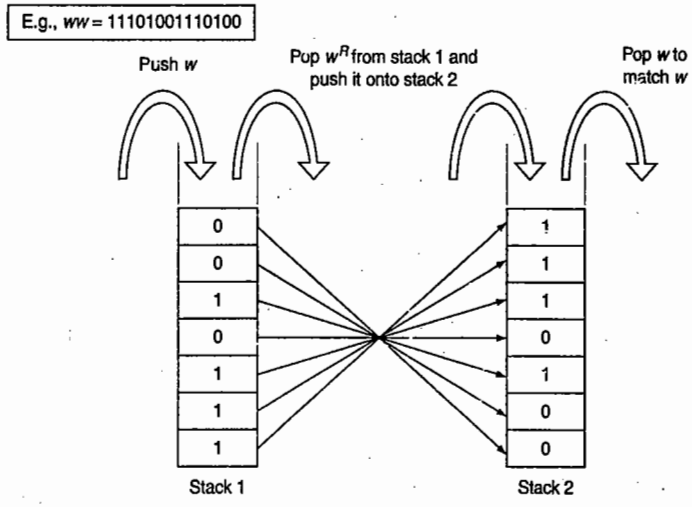


FIGURE 11.8 A PDA with two stacks deciding the language ww.

second stack. The second half of the string can now be matched against this reversed string by popping symbols off the second stack, the same way we match the two halves of an even palindrome (see Example 8.3 in Chapter 8).

Now let us see if it is possible to construct a CSG for such a language. For simplicity, let $\Sigma = \{0, 1\}$. The following grammar generates all strings of the form ww over this binary alphabet. Figure 11.9 illustrates the working of four kinds of variables in this grammar apart from V, the main variable that generates additional pairs of matching symbols to extend the sentential form:

1. P and Q are placeholders that mark the points where symbols are being inserted into the first part of the sentential form.
2. R and T are the "right runners" that go till the end of the sentential form.
3. E is the "end marker" to stop the "right runners" so that they can insert the corresponding symbol in the second part of the sentential form.
4. L and M are the "left runners" that come back to the placeholder. Right and left runners are also known as "messenger variables."

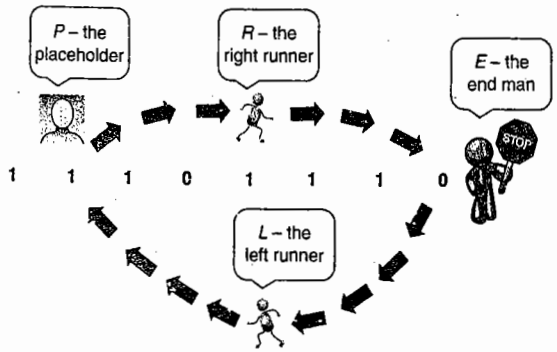


FIGURE 11.9 The context of generating a sentential form as symbols are inserted at two different points using messenger variables.

Context-sensitive grammar for ww:

$S \to 0V0E \mid 1V1E \mid \lambda$	Both parts w begin with a 0 or with a 1; E marks the end of the entire string; V is capable of growing w further as shown below
$V \to 0PR \mid 1QT \mid \lambda$	P holds the place where the first string ends; R runs to the right to insert a matching 0; Q and T behave similarly when 1 is added to both ws
$R1 \to 1R$ $R0 \to 0R$	R runs to the right
$RE \to L0E$	R finds E, inserts a 0 and creates an L to run back to the left
$0L \to L0$ $1L \to L1$	L runs to the left
$PL \to VL$	L comes back to the placeholder P and replaces P with a V to generate further pairs of symbols
$L \to \lambda$	L destroys itself
$E \to \lambda$	Finally, E destroys itself to complete the string

The rules for Q and T are similar to those for P and R:

$T1 \to 1T$ $T0 \to 0T$	T runs to the right
$TE \to M1E$	T finds E, inserts a 1 and creates an M to run back to the left
$0M \to M0$ $1M \to M1$	M runs to the left
$QM \to VM$	M comes back to the placeholder Q and replaces Q with a V to generate further pairs of symbols
$M \to \lambda$	M destroys itself

Note that L, M and E can destroy themselves too soon but this would not generate any final string. P and Q need L and M to replace themselves with a V and only V can go to λ to complete the string. Similarly, if E is destroyed too soon, R or T will never be destroyed thereby ensuring that the grammar does not generate any string that is not in the language of ww.

Finally, let us see how this grammar can generate a sample string of the form ww, namely, 11101110:

$$
\begin{aligned}
S &\Rightarrow 1V1E \\
&\Rightarrow 11QT1E \quad \Rightarrow 11Q17E \\
&\Rightarrow 11Q1M1E \quad \Rightarrow 11QM11E \quad \Rightarrow 11VM11E \\
&\Rightarrow 11V11E \\
&\Rightarrow 111QT11E \quad \Rightarrow 111Q1T1E \quad \Rightarrow 111Q11TE \\
&\Rightarrow 111Q11M1E \Rightarrow 111Q1M11E \Rightarrow 111QM111E \Rightarrow 111VM111E \\
&\Rightarrow 111V111E \\
&\Rightarrow 1110PR111E \Rightarrow 1110P1R11E \Rightarrow 1110P11R1E \Rightarrow 1110P111RE \\
&\Rightarrow 1110P111L0E \Rightarrow 1110P11L10E \Rightarrow 1110P1L110E \Rightarrow 1110PL1110E \\
&\Rightarrow 1110VL1110E \Rightarrow 1110V1110E \Rightarrow 11101110E \quad \Rightarrow 11101110
\end{aligned}
$$

Similarly, we can show that multiplication is a CSL. We have already shown that multiplication is not a context-free language (see Example 9.3 in Chapter 9). To show that it is indeed a CSL, we can either construct a CSG for multiplication or simply argue that a limited type of Turing machine known as linear-bounded automaton (see Fig. 11.10) can be made to carry out multiplication.

11.11 Other Grammars and Automata

All the types of grammars that we have seen so far have had a simple restriction on their production rules: their right-hand sides cannot be shorter than their left-hand sides (with the exception of a single variable going to λ, which, although representing a null string is considered a symbol in the grammar). If we remove this "non-shrinking" restriction, where the right-hand side can be shorter, we get what is called an *unrestricted grammar*. In such grammars, even when the left-hand side of a production contains more than one variable or terminal symbol, the right-hand side can be λ.

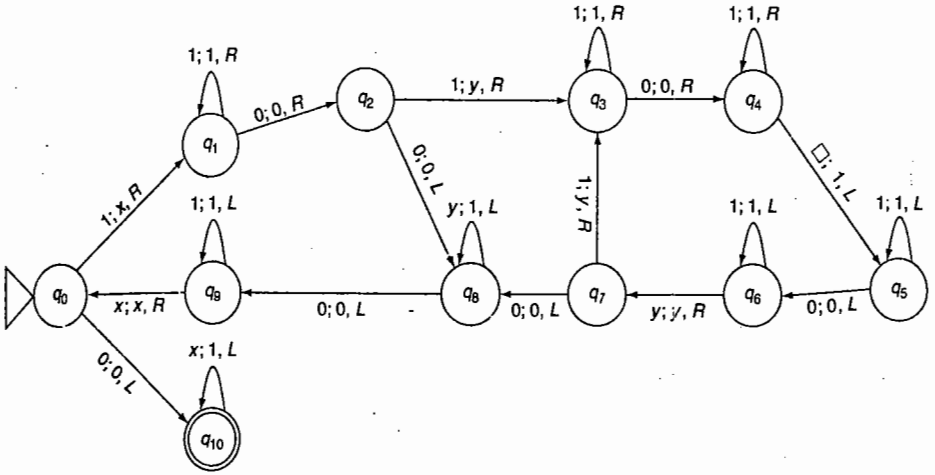


FIGURE 11.10 A linear-bounded automaton (Turing machine) for unary multiplication.

The only restriction in them is that the left-hand side cannot be λ (i.e., something cannot be derived from nothing!).

Unrestricted grammars correspond to the class of RE languages. They are the largest class of grammars and include context-sensitive (and hence context-free, and hence regular) grammars. An example of an unrestricted grammar (or non-context-sensitive grammar) with "shrinking" rules is

$$S \to A1B$$

$A1 \to 0$ A shrinking context-sensitive rule

$0B \to S1$ A context-sensitive rule

$0B \to \lambda$ Another shrinking context-sensitive rule

However, this grammar can be simplified to a left-linear grammar

$$S \to S1 \mid \lambda$$

with the language of the grammar being just 1^*, a regular language. Thus, the language can be a simple one even if the grammar has shrinking rules. At the same time, we cannot easily find an example of a grammar for a non-context-sensitive language (i.e., a recursive or an RE language that is not context-free) because any such language does not have a compact description (e.g., a grammar) and is not very interesting; we can only find such languages through a diagonalization argument similar to how we found the language L_{Diag}.

1. What about machines for CSLs?
2. Are there automata that are more powerful than PDAs but less powerful than Turing machine deciders?

3. In fact, what about PDAs having two stacks that we used above to illustrate the behavior of CSLs?

It turns out that a class of machines known as *linear-bounded automata (LBAs)* correspond to the class of CSLs. These machines are similar to a standard Turing machine but have a finite number of cells on the tape that define the boundaries of its memory. The machine must carry out all of its processing by reading or writing within the bounds on the tape. The length of the tape allowed for use must be a linear function of the length of the input string (which also includes any output string, since, in converting a problem to a formal language, the output is included in the input, e.g., instead of asking the automaton to compute 2 + 3 and give the output, it will be asked whether 2 + 3 = 5 is true). These machines are interesting to the extent that they demonstrate how a limited amount of memory is sufficient to handle all CSLs which include the vast majority of practical problems in computing. We do not need infinite memory to deal with all such problems.

Figure 11.10 shows an linear-bounded automaton (LBA) for the language of unary multiplication. What is shown is in fact a Turing machine which actually computes and writes the product onto the tape (e.g., given 1110110 as the input, i.e., 3×2, the output is 1110110111111, i.e., $3 \times 2 = 6$), but it can be easily converted to an LBA by assuming that the product of the two numbers is also present in the input string which is to be merely accepted when the product is correct or rejected when incorrect.

In a sense, linear-bounded automata are the class of machines closest to modern digital computers. Unlike Turing machines with their infinitely long tapes, these machines have limited memory (and hence can be implemented in practice) but are not limited in ways of using the memory. Unlike PDAs, their memory access behavior is equivalent to random access although it appears to support only sequential access. Moreover, LBAs can also have multi-track tapes with a composite read–write head that can access multiple corresponding cells at a time.

PDAs with two stacks are in fact more powerful than linear-bounded automata. It can be shown that a two-stack machine is as powerful as a Turing machine. It may also be noted that there is no class of grammars that corresponds to the set of recursive languages.

11.12 Linear and Deterministic Context-Free Languages

Earlier in our study of regular languages (Sec. 5.3), we saw that regular grammars are either left-linear or right-linear. A linear grammar is one that has at most one variable on the right-hand side of every production. However, not all linear grammars are regular. For example, the CFL $a^n b^n$ has a linear grammar:

$$S \to aSb \mid \lambda$$

As such, regular languages are a proper subset of linear languages which in turn are a proper subset of context-free languages. Note that we normally refer only to CFLs when we talk about linear grammars and languages. Even CSGs can be "linear." For example, the CSG for $a^n b^n c^n$ in Example 11.1 has only one variable on the right-hand sides of all its productions.

There is another subset of CFLs known as *deterministic context-free languages* (DCFLs). These are the languages which deterministic pushdown automata (DPDAs) can accept without any need for

non-determinism. DCFLs are important from the practical point of view of designing programming languages and their compilers. Since it is very inefficient to implement non-determinism in a compiler, programming languages should be designed to be mostly deterministic context-free (although they invariably contain a few context-sensitive features).

An example of a context-free language that is not deterministic is the language of even palindromes ww^R. We saw earlier (Example 8.3 in Chapter 8) that the PDA had to non-deterministically guess the midpoint of the string in order to handle this language. A DPDA is unable to handle this language.

Non-deterministic finite automata are equivalent to deterministic finite automata. There is nothing new gained by introducing non-determinism to finite automata since these simple computing machines do not have any memory other than their states. As seen in Sec. 3.3, removing non-determinism merely increases the number of states in the machine.

Non-deterministic PDAs, on the other hand, are more powerful than DPDAs since these machines have a separate memory unit in the form of a stack. It is not possible for a DPDA to simulate in its single stack all the configurations of stack contents that a non-deterministic PDA can try out in parallel. For example, in the case of the even palindrome language, non-determinism is required to try every possible position in the string as a potential midpoint before finding (or guessing) the right one. A DPDA is unable to do this while processing the input string from left-to-right in a single pass.

In the case of Turing machines however, we saw in Chapter 10 that non-deterministic Turing machines are equivalent to standard Turing machines. This is because it is possible to simulate non-deterministic behavior of memory (i.e., multiple parallel versions) in a standard Turing machine given its freedom to move back and forth freely along its infinitely long tape.

Languages that correspond to DPDAs, namely *deterministic context-free languages* (DCFLs) are a proper subset of CFLs but overlap with linear languages in the following ways (see Fig. 11.11):

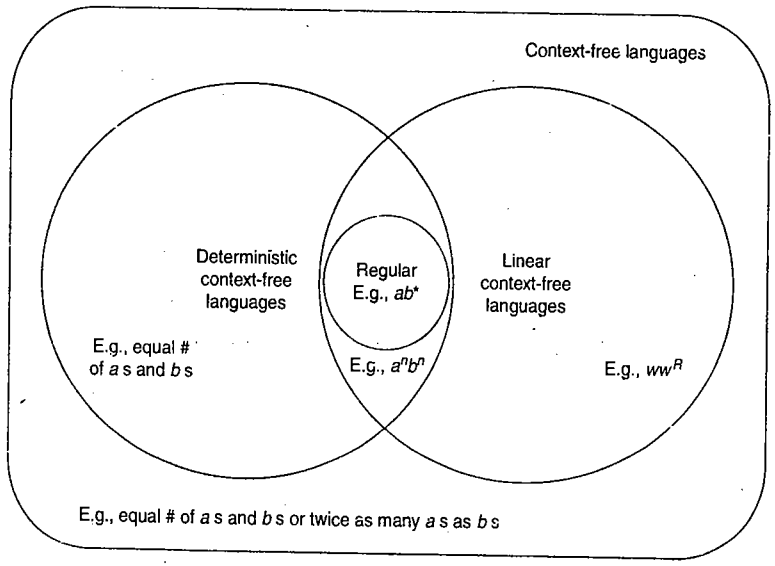


FIGURE 11.11 Deterministic and linear context-free languages.

1. Every regular language is both linear and deterministic context-free.
2. There are linear languages that are not deterministic. For example, ww^R is linear but non-deterministic context-free.
3. There are deterministic context-free languages that are not linear. For example, the set of all binary strings with equal numbers of 0 and 1 is deterministic but not linear. The grammar for this language requires the non-linear rule $S \rightarrow SS$ but a PDA is able to handle the language deterministically (as seen in Example 8.5 in Chapter 8).
4. There are languages that are both linear and deterministic but not regular. For example, simple nesting ($a^n b^n$) or the language of odd palindromes with a distinct marker at the midpoint (wcw^R) can be handled deterministically using the following linear grammars respectively:

$$a^n b^n \ : \ S \rightarrow aSb \mid \lambda$$

$$wcw^R : \ S \rightarrow aSa \mid bSb \mid c$$

5. Finally, there are CFLs that are neither linear nor deterministic. For example, the language of all strings that have either an equal number of a s and b s or twice as many a s as b s is both non-linear and non-deterministic; yet it is a context-free language (see Exercise 15 in Chapter 7).

11.13 A Complete Chomsky Hierarchy

Let us now summarize the mappings between classes of formal languages, machines or automata and grammars (Table 11.1).

TABLE 11.1 Languages. Grammars and Automata

	Formal Language	Grammar	Automaton or Machine	Chomsky Type	Example
1	Regular language	Right-linear or left-linear grammar or regular expression	Finite automata (Deterministic or non-deterministic)	Type III	Numbers divisible by 4, i.e., (0+1)*00 or $S \rightarrow A00$, $A \rightarrow A0 \mid A1 \mid \lambda$
2	Linear language	Linear grammar	–	–	Simple nesting, i.e., $a^n b^n$
3	Deterministic context-free language	Deterministic context-free grammar	Deterministic pushdown automata	–	Equal numbers of 0s and 1s (in any order)
4	Context-free language	Context-free grammar	(Non-deterministic) pushdown automata	Type II	Addition, i.e., $1^m 0 1^n 0 1^{m+n}$ or $S \rightarrow 1S1 \mid 0A$, $A \rightarrow 1A1 \mid 0$
5	Context-sensitive language	Context-sensitive grammar	Linear-bounded automata	Type I	Multiplication, i.e., $1^m 0 1^n 0 1^{m \times n}$

(Contd.)

Table 11.1 (*Continued*)

	Formal Language	Grammar	Automaton or Machine	Chomsky Type	Example
6	Recursive language	–	Turing machine decider	–	–
7	Recursively enumerable language	Unrestricted grammar	Turing machine acceptor	Type 0	L_{Diag}
8	Non-recursively enumerable language	–	–	–	$L_{Non\text{-}RE}$

Figure 11.12 shows a complete Chomsky Hierarchy with machines and grammars mapped to classes of languages. The diagram introduces the intermediate class of *deterministic context-free languages*.

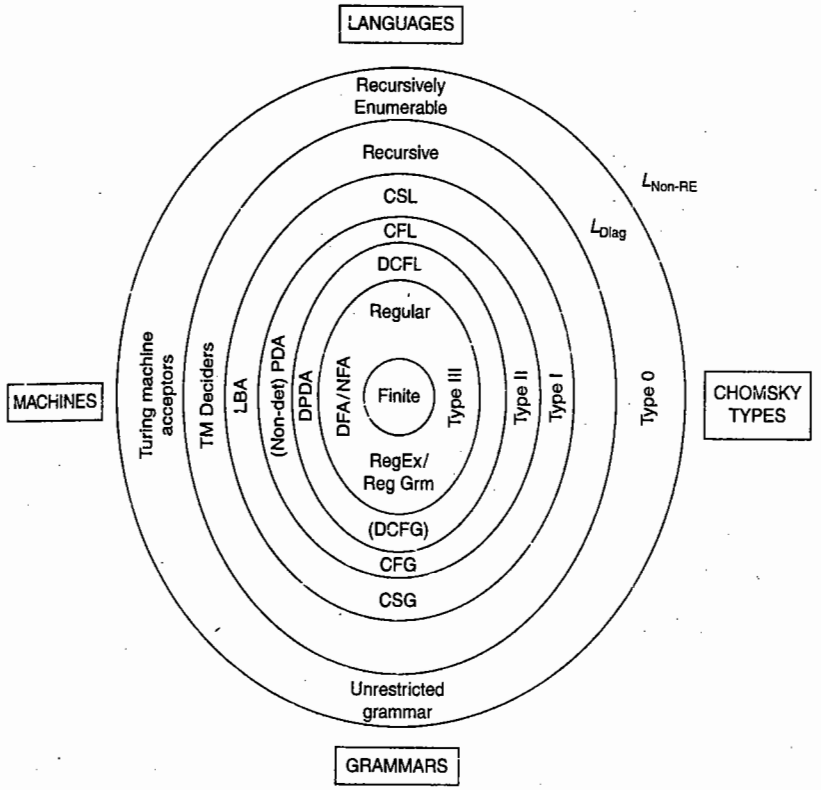


FIGURE 11.12 A complete Chomsky Hierarchy.

Set–subset relationships among the various classes of formal languages are summarized in the list of Key Ideas below. The theory presented in this chapter has important practical applications in two related areas. In the design of high-level programming languages and their parsers, compilers and interpreters, the class of formal languages to which the programming language belongs plays a major role. Most programming languages lie somewhere between context-free and context-sensitive languages. That is, they are context-free for the most part, but with a few context-sensitive features. As noted in Chapter 7, they are also designed to be as deterministic and unambiguous as possible to reduce the complexity of parsers.

The other area of application is in the processing or natural languages such as English. Natural languages are known to be context-sensitive as well. English, for example, has "long-distance dependencies" (e.g., in its use of tag questions) similar to the dependencies between two parts of a sentence that we saw in a context-sensitive formal language such as *ww*. A variety of grammars and parsing techniques such as "tree adjoining grammars" have been developed for "mildly context-sensitive languages" to enable parsing and processing of natural language texts.

Higher classes of languages, namely recursive and recursively enumerable languages, serve to mainly illustrate the limitations of computing – a topic that we explore in the next chapter.

11.14 Key Ideas

1. Turing machines correspond to the class of formal languages known as recursively enumerable languages and the class of grammars known as unrestricted grammars.
2. The largest formal language over a given alphabet is the * closure of the alphabet Σ^*, a regular language. All other languages, however complex, are subsets of this simple language.
3. Any set or language whose elements can be mapped one-to-one to the set of natural numbers is said to be countably infinite.
4. Strings in any infinite language are countably infinite.
5. Not all infinite sets are countably infinite. A set such as the set of all real numbers is uncountably infinite.
6. Uncountability of sets is demonstrated using the technique of diagonalization. This is a proof by contradiction in which a new element of the set is constructed from the diagonal of a matrix. The newly constructed element is different from all the enumerated elements of the set, thereby showing that the set is not enumerable.
7. Any set that is recursively enumerable has a proper enumeration procedure whereby any given element of the set can be enumerated or listed in finite time.
8. Turing machines are countably infinite since they can all be encoded as binary strings. Recursively enumerable languages are also countably infinite.
9. The set of all languages over an alphabet is uncountably infinite. There are more languages than there are Turing machines to accept them. Not all languages are recursively enumerable. Not all languages are computable.
10. Recursively enumerable languages whose complements are also recursively enumerable are called recursive languages. The complement of a recursive language is also recursive.

11. Recursively enumerable languages have Turing machine acceptors that accept any string in the language. Recursive languages have Turing machine deciders that compute the membership function for the languages. They accept all strings in the language and reject all strings that are not in the language.
12. Finite languages are a proper subset of regular languages since every finite language is regular but not every regular language is finite.
13. Regular languages are a proper subset of context-free languages. Every regular language is context-free but not vice versa. A finite automaton for a regular language is nothing but a pushdown automaton that never uses its stack. On the other hand, not every context-free language is regular; this has been shown using the Pumping Lemma for regular languages.
14. Context-free languages (CFLs) are a proper subset of context-sensitive languages (CSLs). Every CFL is a CSL because its context-free grammar is nothing but a context-sensitive grammar in which every production happens to have only a single variable on its left-hand side. Not every CSL is context-free as can be seen by applying the Pumping Lemma for context-free languages.
15. Context-sensitive languages (CSLs) are a proper subset of recursive languages. Every context-sensitive language is recursive since a linear-bounded automaton for a CSL is actually a decider Turing machine for the language. However, not every recursive language is a context-sensitive language as can be shown by applying the diagonalization technique.
16. Context-sensitive languages are the languages of context-sensitive grammars which follow the non-shrinking rule: every production rule must have a right-hand side that is at least as long as the left-hand side so that sentential forms do not shrink during a derivation.
17. Recursive languages are a proper subset of recursively enumerable languages. Every recursive language is recursively enumerable since a decider Turing machine for a recursive language is also an acceptor for the language. On the other hand, there exist languages such as L_{Diag} that are recursively enumerable but not recursive (since its complement $L_{\text{Non-RE}}$ is not recursively enumerable).
18. Recursively enumerable languages are a proper subset of 2^{Σ^*}, the set of all formal languages over the alphabet Σ because of the existence of languages such as $L_{\text{Non-RE}}$ that are not recursively enumerable.
19. In addition, both linear languages and deterministic context-free languages are proper subsets of context-free languages but not subsets of each other. However, regular languages are a proper subset of both linear and deterministic context-free languages.
20. Pushdown automata with two stacks are as powerful as Turing machines. Turing machines with limited lengths of tape known as linear-bounded automata are equivalent to context-sensitive languages.
21. The hierarchy of formal languages along with corresponding classes of grammars and computing machines is known as the Chomsky Hierarchy.

11.15 Exercises

1. Show that the set of all even palindromes over $\{a, b\}$ is countably infinite.
2. Design an enumeration procedure for all possible names of people (without imposing any limit on the length of any part of a name: first, middle or last names).

3. Give an example of a formal language (other than the ones shown in Fig. 11.11) that is:
   (a) Linear but not regular.
   (b) Linear and deterministic context-free but not regular.
   (c) Deterministic context-free but not linear.
   (d) Linear but not deterministic context-free.
   (e) Context-free but neither deterministic nor linear.
4. Old McDonald had a farm... *E-I-E-I-O*... and on that farm he had some chickens... and some sheep... and some horses... and some dogs. On his farm, there must always be more chickens than sheep; more sheep than horses; and more horses than dogs. If we represent the animals on this farm using the alphabets $\{c, s, h, d\}$ standing for chicken, sheep, horse and dog, respectively, then a string such as *ccccccssssshhhdd* represents 6 chickens, 4 sheep, 3 horses and 2 dogs (always in that order). Consider the formal language of all such strings with the above constraints.
   (a) What class of formal languages does this language belong to if there can be at most 1000 animals on the farm? Prove your answer.
   (b) What class does it belong to if there is no limit on the number of animals? Prove your answer. (For example, if you claim that the language is context-free, you must prove that it is context-free AND you must prove that it is not regular.)
5. Construct a context-sensitive grammar for the language of unary multiplication.
6. Show that (unary) division is also context-sensitive.
7. Given two strings of equal length (separated by a single c) followed by (a single c and then) an interleaved string composed of the first elements of the two strings, followed by the second elements of the two strings, and so on, ending with the last elements of the two strings, in that order:

   $$x_1x_2x_3\ldots x_nc y_1y_2y_3\ldots y_nc x_1y_1x_2y_2x_3y_3\ldots x_ny_n$$

   Construct a context-sensitive grammar that accepts such strings for the alphabet $\{a, b, c\}$. For example, *aabacbbabcababbaab* is in the language but not *aaacbbbcabaabb*.
8. Construct a PDA with two stacks for the language $a^nb^nc^nd^ne^nf^n$.
9. Prove or disprove: Every subset of a regular language is a regular language.
10. In the Chomsky Hierarchy of formal languages, why are context-sensitive languages a proper subset of recursive languages?
11. Why are recursive languages a proper subset of recursively enumerable languages?
12. Are there formal languages for which there is no Turing machine that accepts the language?
13. How do we construct a grammar for a language for which there is no Turing machine acceptor?
14. Is there an unrestricted grammar whose language is not context-sensitive but yet can be described concisely (without using diagonalization)? Explain.

15. What is the smallest class of formal languages in which the complements of some of the languages do not belong to the same class?

16. Most modern programming languages have certain syntactic features that are not context-free. Yet, their syntax is specified using BNF (Backus–Naur Form) rules that are context-free. What happens to non-context-free features? How are they handled?

17. Figure 11.10 shows a Turing machine which computes and writes the product of two unary numbers onto the tape (e.g., given 1110110 as the input, i.e., 3×2, the output is 1110110111111, i.e., $3 \times 2 = 6$). Convert this to an LBA for the language of unary multiplication whose input also includes the product of the two numbers. The LBA should accept the input string if the product is correct and reject otherwise.

Chapter Twelve

Computability and Undecidability

Learning Objectives

After completing this chapter, you will be able to:

- Learn to solve simple cases of the Post Correspondence Problem (PCP).
- Learn why PCP is unsolvable.
- Learn the difference between decidability and computability.
- Learn why some functions are not computable.
- Understand why Turing machines suffer from the halting problem.
- Appreciate kinds of problems that are undecidable.
- Get an overview of the field of computational complexity.
- Understand the difference between recognition and recall in computing solutions.
- Wonder about several open questions in the science of computing.

The final chapter is about the key ideas of computability and decidability. It shows the limitations of computing in the form of unsolvable problems such as the seemingly simple and practical Post Correspondence Problem. However, to show why there cannot be an algorithm for this problem, we must first study the Turing machine halting problem and the equivalence of the two problems can only be appreciated through the introduction of derivations in unrestricted grammars. The section that follows tells the reader about several practical problems that are undecidable.

The final part of the chapter presents a brief coverage of the topic of complexity of solvable problems and introduces the reader to one of the greatest unresolved questions in all of computer science today: Is $P = NP$? Finally, it summarizes the entire subject matter and ends with an open question on whether Turing's model of computing is indeed the right one by raising questions about the visibly differing ease with which recognition and recall problems are solved by modern computers based on Turing's model and the human mind.

12.1 The Post Correspondence Problem

We begin by considering a simple and practical problem in string matching known as the *Post Correspondence Problem (PCP)*. This is not the trivial problem of comparing just two strings. Rather, we are given two sets (sequences, in fact) of strings W and V (over the same alphabet) each having n strings:

$$W = \{w_1, w_2, w_3, \ldots, w_n\} \text{ and } V = \{v_1, v_2, v_3, \ldots, v_n\}$$

The problem is to find some concatenation of one or more strings from the set W, in any order, where the resulting concatenated string is identical to the corresponding concatenation, in the same order, of strings from the set V.

Suppose we concatenate all the strings in W in the order in which the elements of W are given: $w_1.w_2.w_3....$ Similarly, $v_1.v_2.v_3....$ Will the two concatenations give us identical strings? That is

$$w_1.w_2.w_3... = v_1.v_2.v_3...?$$

Not always. Is there some other order of concatenation that will give us identical results for both the sets?

The simpler version of the PCP is to find some permutation of the indexes of elements of the set $\{1, 2, 3, 4, ...\}$, say, $i_1, i_2, i_3, i_4, ...$ so that when concatenated in that order, we get identical concatenations, that is,

$$w_{i1}.w_{i2}.w_{i3}... = v_{i1}.v_{i2}.v_{i3}...$$

The V strings must be concatenated in the same order as the W strings. For example, i_1 may be 1, i_2 may be 7, etc.

Can we find such a permutation given any two sets of strings W and V each containing n strings?

This problem can be solved easily: Since n is finite, one can simply enumerate all permutations and try each resulting concatenation. The general case of the PCP is more complex. What if we allow repetitions of strings? That is, each w_i or v_i can be used any number of times in the concatenations. As such, there is no limit on the length of the concatenations, although we only consider finite numbers of concatenations as solutions to the PCP. There are instances of the PCP that have no solution if repetitions are not allowed but do present solutions when repetitions are included (see Example 12.2 for instance).

It turns out that only some instances of the PCP have solutions. For some other pairs of sets W and V, there is no solution to the PCP, that is, there is no way to produce identical concatenations from the two sets. One such trivial case occurs when every string in set W is longer than the corresponding string in the set V, thereby making any concatenation of W strings longer than (and therefore not identical to) a corresponding concatenation of V strings.

What is a good algorithm to solve the Post Correspondence Problem?

Let us try to discover an algorithm by solving a few instances of the PCP.

EXAMPLE 12.1

Consider

$$W = \{bb, baa, bbb\}$$
$$V = \{bbb, aab, bb\}$$

In this case, if we concatenate the strings in the same order in which they are given to us, we get identical concatenated strings: $i_1 = 1$, $i_2 = 2$ and $i_3 = 3$, and

$$w_{i1}.w_{i2}.w_{i3} = bb.baa.bbb =$$
$$v_{i1}.v_{i2}.v_{i3} = bbb.aab.bb = bbbaabbb$$

This instance of the PCP has a simple solution. A greedy strategy worked well in this case.

EXAMPLE 12.2

Now consider another PCP instance:

$$W = \{a, ab, bba\}$$
$$V = \{baa, aa, bb\}$$

1. What is a good algorithm for solving this problem?
2. Can we apply an exhaustive search by considering all possible permutations?

We can easily enumerate all the permutations and conclude that none of them constitutes a solution. There may still be a solution that involves use of one or more strings more than once. For example, we can begin only with w_3 and v_3 since w_1 and v_1 or w_2 and v_2 do not even match each other. If we begin with w_3 and v_3, the only next choice is w_2 and v_2:

$$w_3.w_2 = bba.ab$$
$$v_3.v_2 = bb.aa$$

The only choice at this point, namely w_1 and v_1, does not work since $bba.ab.a \neq bb.aa.baa$. Unless we are allowed to repeat w_3 and v_3, there is no way to proceed further in our search for a solution.

One observation we can make at this point is that as we go about building the two concatenations, they must be identical in their prefixes, up to the end of the shorter concatenation (e.g., $v_3.v_2$ above), for otherwise, they are already not identical to each other and do not constitute a solution to the PCP.

Continuing with the possibility of repeating w_3 and v_3, we get a solution to this PCP:

$$w_3.w_2.w_3.w_1 = bba.ab.bba.a$$
$$v_3.v_2.v_3.v_1 = bb.aa.bb.baa$$

The solution is $i_1 = 3$, $i_2 = 2$, $i_3 = 3$ and $i_4 = 1$. Notice that we can find additional solutions by repeating the entire concatenations any number of times. For example,

$$w_3.w_2.w_3.w_1.w_3.w_2.w_3.w_1 = bbaabbbaabbaabbbaa$$
$$= v_3.v_2.v_3.v_1.v_3.v_2.v_3.v_1$$

is also a solution to the same problem.

EXAMPLE 12.3

Let us look at one final example:

$$W = \{a, aa, b\}$$
$$V = \{aa, ab, bb\}$$

In this case, if we start with w_1 and v_1 we end up with the V-concatenation having too many bs at the end (or too many as at the beginning if we had chosen to repeat w_1 and v_1 at the start):

$$w_1.w_2.w_3.w_3 = a.aa.b.b$$
$$v_1.v_2.v_3.v_3 = aa.ab.bb.bb$$

The other path would have been to start with w_3 and v_3, to end up with too many bs again in the V-concatenation:

$$w_3.w_3.w_2 = b.b.b\ldots$$
$$v_3.v_3.v_3 = bb.bb.bb\ldots$$

All the paths end up in an infinite loop. There appears to be no solution at all to this instance of the PCP.

In other instances, as we grow the concatenations, we often come "back to square one" or go into an infinite loop that is not as easy to recognize. That is, a loop may occur after many more concatenations and it is impossible for any algorithm to predict which choice would result in getting stuck in a loop.

There is no algorithm that can solve all instances of the PCP! Even if we try to make the algorithm smart, even if it detects loops and makes better choices, there will always be instances of the problem where the loops will be longer or it will get misled and ends up stuck in a loop.

It is not just that no one has as yet discovered a good algorithm for solving PCP. There will never be an algorithm that can solve all instances of this problem in string matching because this is an example of a computing problem that is unsolvable! As such, we say that PCP is not *computable* or not *decidable*.

Any "algorithm" we devise for PCP may solve some instances of the problem but not others; it is bound to go into an infinite loop when given other instances of the problem. Why is this so? To understand this surprising result, we need to study the *halting problem of Turing machines*.

12.2 Equivalence of Derivation and PCP

PCP is an unsolvable problem because, any algorithm that we try to devise for PCP may be able to solve some or many instances of it; but it necessarily suffers from the halting problem of Turing machines wherein it will never halt for some instances of the problem.

Why is PCP unsolvable? What is the relation between PCP and Turing machines? We will now apply our knowledge of Turing machines and, in particular, universal Turing machines to study the halting problem and the limitations of computing (and of algorithmic problem solving) that it demonstrates.

In doing so, we use our knowledge of recursively enumerable and recursive languages from Chapter 11. You will remember from that chapter that we used a technique known as diagonalization to prove that there exist formal languages that are not recursively enumerable and therefore that not all recursively enumerable languages are recursive. We will now show that if the Post Correspondence Problem is solvable, that is, if there exists an algorithm that can solve any instance of the PCP without suffering from the halting problem of Turing machines, then all recursively enumerable languages would become recursive. This contradiction is our proof that PCP is unsolvable and that Turing machines do suffer from the halting problem.

As you know, we have been using regular expressions and regular grammars to model (or formally represent) regular languages and context-free grammars to model context-free languages. How do we model the largest class of formal languages known as recursively enumerable languages? For this, we introduced (in Chapter 11) the largest class of grammars known as *unrestricted grammars*. In what follows, we show that a derivation of a string in an unrestricted grammar is equivalent to a concatenation that solves a corresponding instance of the PCP. One can also show that the two concatenated strings in a PCP represent the "computation history" of a Turing machine, that is, the sequence of state transitions that the machine goes through as it carries out its computation.

For convenience, we introduce a version of the PCP known as the *Modified Post Correspondence Problem* (MPCP). In this problem, there is a simple additional constraint that $i_1 = 1$ always, that is, a solution must begin by using the first strings of both the sets (or, rather, sequences) A and B. It can be shown that the two problems are equivalent in the sense that if MPCP is solvable (i.e., if there is an algorithm for MPCP) then the more general PCP will also become solvable, that is, the algorithm, if it existed, could be adapted to solve PCP as well. This is easy to see: the algorithm for PCP, if it existed, simply needs to consider n different versions for each of the n strings permuted as the first string and then the algorithm for MPCP, if it existed, can be invoked to solve at least one of them.

Why is MPCP unsolvable?

Consider unrestricted grammars (see Chapter 11) which are equivalent to Turing machines and recursively enumerable languages. And consider a derivation in such a grammar wherein the start symbol S derives a string w in the language by going through a finite number of intermediate sentential forms. Each of those steps applies a production from the grammar. There is no restriction on how many times the same production can be used in a derivation, and thus strings of unlimited length can be generated using the grammar. Each of those applications of a production corresponds to a (set of) transition(s) in an equivalent Turing machine. And the number of transitions in a Turing machine can be many more than the number of states in its control automaton, the same way that the number of steps in a derivation can be far greater than the number of productions in the grammar.

We will now show that a derivation in an unrestricted grammar corresponds to finding a solution to an instance of MPCP. This requires a strange construction that turns the derivation into an MPCP. The construction sets up so that the W-concatenation introduces $S \Rightarrow$ and the V-concatenation is forced to find a production starting with S to match the W-concatenation. Similarly, for each step in the derivation wherein a production is applied to the sentential form, a pair of strings is selected to extend the concatenation in the partial solution to the MPCP.

EXAMPLE 12.4

Figure 12.1 shows a derivation of the string *aababbabb* which belongs to the language over $\{a, b\}$ of strings with equal numbers of a and b (i.e., $n_a = n_b$). The grammar used for the language is as follows:

$$S \rightarrow aSB \mid bSA \mid SS \mid \lambda$$
$$A \rightarrow a$$
$$B \rightarrow b$$

In fact, this is a context-free grammar but it can be easily converted to an unrestricted grammar without changing the language as follows:

$$S \rightarrow aSBB \mid bSAA \mid SS \mid \lambda$$
$$AA \rightarrow a$$
$$BB \rightarrow b$$

We will continue to use the context-free grammar and not the above unrestricted grammar for the rest of the example. The derivation using the CFG is as follows:

$$S \Rightarrow aSB \Rightarrow aaSBB \Rightarrow aabSABB \Rightarrow aabaSBABB \Rightarrow aaba\lambda BABB \Rightarrow \ldots \Rightarrow aababbabb$$

This entire derivation corresponds to both the W-concatenation (shown on the left) and the V-concatenation (shown on the right of Fig. 12.1).

Let every symbol in the grammar – variables as well as terminals – be strings (of length one) in both the sets in the corresponding MPCP. For every production in the grammar, the left-hand side and the right-hand side of the production will occur as corresponding pairs of strings in the MPCP sets V and W, respectively. As shown in Fig. 12.1, the W-concatenation is one step ahead of the V-concatenation. It is as though the V-concatenation is chasing the W-concatenation to catch up with it. For example, in the first step, the W-concatenation already has the start symbol from the grammar S; the V-concatenation has to find a production that expands S into a string of variables and terminals which together will be concatenated to W.

Let us now look at the complete W and V sets as well as the concatenations obtained in the present example. After the mandatory first pair of strings for MPCP, the next two strings are for the terminals, the next three for the non-terminals and the following six pairs are for the productions in the grammar.

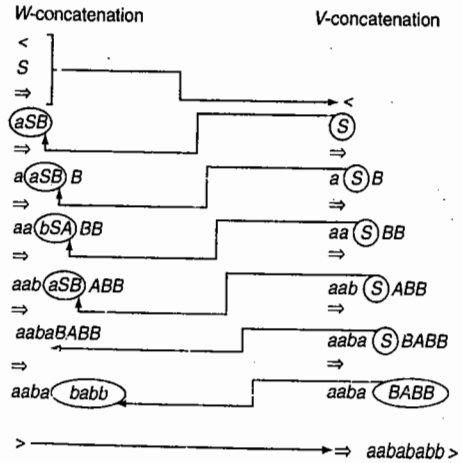


FIGURE 12.1 Concatenations in MPCP correspond to applying productions in a derivation.

The last two are essential to make the correspondence between derivation and MPCP work, including the last string v_{14} which has the entire input string being derived.

$$w_1, w_2, w_3, w_4, w_5, w_6, w_7, w_8, w_9, w_{10}, w_{11}, w_{12}, w_{13}, w_{14}$$
$$W = \{<S \Rightarrow, a, b, A, B, S, aSB, bSA, SS, \lambda, a, b, \Rightarrow, >\}$$
$$V = \{<, a, b, A, B, S, S, S, S, S, A, B, \Rightarrow, \Rightarrow aabaabbabb>\}$$
$$v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}, v_{11}, v_{12}, v_{13}, v_{14}$$

It may also be noted that start and end markers "<" and ">" are used for the concatenations. The final concatenations and their correspondence are shown in Table 12.1.

TABLE 12.1 MPCP Concatenations Corresponding to a Grammar Derivation

#	i	W-Concatenation	w-String	v-String	V-Concatenation
1	1	w_1	$<S \Rightarrow$	$<$	v_1
2	7	$w_1.w_7$	$<S \Rightarrow aSB$	$<S$	$v_1.v_7$
3	13	$w_1.w_7.w_{13}$	$<S \Rightarrow aSB \Rightarrow$	$<S \Rightarrow$	$v_1.v_7.v_{13}$
4	2	$w_1.w_7.w_{13}.w_2$	$<S \Rightarrow aSB \Rightarrow a$	$<S \Rightarrow a$	$v_1.v_7.v_{13}.v_2$
5	7	$w_1.w_7.w_{13}.w_2.w_7$	$<S \Rightarrow aSB \Rightarrow aaSB$	$<S \Rightarrow aS$	$v_1.v_7.v_{13}.v_2.v_7$
6	5	$w_1.w_7.w_{13}.w_2.w_7.w_5$	$<S \Rightarrow aSB \Rightarrow aaSBB$	$<S \Rightarrow aSB$	$v_1.v_7.v_{13}.v_2.$ $v_7.v_5$
7	13	$w_1.w_7.w_{13}.w_2.w_7.$ $w_5.w_{13}$	$<S \Rightarrow aSB \Rightarrow aaSBB \Rightarrow$	$<S \Rightarrow aSB \Rightarrow$	$v_1.v_7.v_{13}.v_2.$ $v_7.v_5.v_{13}$
8	2	$w_1.w_7.w_{13}.w_2.w_7.$ $w_5.w_{13}.w_2$	$<S \Rightarrow aSB \Rightarrow aaSBB \Rightarrow a$	$<S \Rightarrow aSB \Rightarrow a$	$v_1.v_7.v_{13}.v_2.$ $v_7.v_5.v_{13}.v_2$
9	2	$w_1.w_7.w_{13}.w_2.w_7.$ $w_5.w_{13}.w_2.w_2$	$<S \Rightarrow aSB \Rightarrow aaSBB \Rightarrow aa$	$<S \Rightarrow aSB \Rightarrow aa$	$v_1.v_7.v_{13}.v_2.$ $v_7.v_5.v_{13}.v_2.v_2$
10	8	$w_1.w_7.w_{13}.w_2.w_7.$ $w_5.w_{13}.w_2.w_2.w_8$	$<S \Rightarrow aSB \Rightarrow aaSBB$ $\Rightarrow aabSA$	$<S \Rightarrow aSB \Rightarrow aaS$	$v_1.v_7.v_{13}.v_2.$ $v_7.v_5.v_{13}.v_2.$ $v_2.v_8$
11	5	$w_1.w_7.w_{13}.w_2.w_7.$ $w_5.w_{13}.w_2.w_2.$ $w_8.w_5$	$<S \Rightarrow aSB \Rightarrow aaSBB$ $\Rightarrow aabSAB$	$<S \Rightarrow aSB \Rightarrow aaSB$	$v_1.v_7.v_{13}.v_2.$ $v_7.v_5.v_{13}.v_2.$ $v_2.v_8.v_5$
12	5	$w_1.w_7.w_{13}.w_2.w_7.$ $w_5.w_{13}.w_2.w_2.$ $w_8.w_5.w_5$	$<S \Rightarrow aSB \Rightarrow aaSBB$ $\Rightarrow aabSABB$	$<S \Rightarrow aSB \Rightarrow aa$ SBB	$v_1.v_7.v_{13}.v_2.$ $v_7.v_5.v_{13}.v_2.$ $v_2.v_8.v_5.v_5$
13	13	$\ldots w_5.w_5.w_{13}$	$<S \Rightarrow aSB \Rightarrow aaSBB \Rightarrow aab$ $SABB \Rightarrow$	$<S \Rightarrow aSB \Rightarrow aa$ $SBB \Rightarrow$	$\ldots v_5.v_5.v_{13}$
14	2	$\ldots w_5.w_5.w_{13}.w_2$	$<S \Rightarrow aSB \Rightarrow aaSBB \Rightarrow aab$ $SABB \Rightarrow a$	$<S \Rightarrow aSB \Rightarrow aa$ $SBB \Rightarrow a$	$\ldots v_5.v_5.v_{13}.v_2$
15	2	$\ldots w_5.w_5.w_{13}.w_2.w_2$	$<S \Rightarrow aSB \Rightarrow aaSBB \Rightarrow aab$ $SABB \Rightarrow aa$	$<S \Rightarrow aSB \Rightarrow aa$ $SBB \Rightarrow aa$	$\ldots v_5.v_5.v_{13}.$ $v_2.v_2$
16	3	$\ldots w_5.w_5.w_{13}.w_2.w_2.$ w_3	$<S \Rightarrow aSB \Rightarrow aaSBB \Rightarrow aab$ $SABB \Rightarrow aab$	$<S \Rightarrow aSB \Rightarrow aa$ $SBB \Rightarrow aab$	$\ldots v_5.v_5.v_{13}.$ $v_2.v_2.v_3$
17	7	$\ldots w_5.w_5.w_{13}.w_2.w_2.$ $w_3.w_7$	$<S \Rightarrow aSB \Rightarrow aaSBB \Rightarrow aab$ $SABB \Rightarrow aabaSB$	$<S \Rightarrow aSB \Rightarrow aa$ $SBB \Rightarrow aabS$	$\ldots v_5.v_5.v_{13}.$ $v_2.v_2.v_3.v_7$

(Contd.)

Table 12.1 *(Continued)*

#	i	W-Concatenation	w-String	v-String	V-Concatenation
18	4	$\ldots w_5.w_5.w_{13}.w_2.w_2.$ $w_3.w_7.w_4$	$<S \Rightarrow aSB \Rightarrow aaSBB \Rightarrow aab$ $SABB \Rightarrow aabaSBA$	$<S \Rightarrow aSB \Rightarrow aa$ $SBB \Rightarrow aabSA$	$\ldots v_5.v_5.v_{13}.v_2.$ $v_2.v_3.v_7.v_4$
19	5	$\ldots w_5.w_5.w_{13}.w_2.w_2.$ $w_3.w_7.w_4.w_5$	$<S \Rightarrow aSB \Rightarrow aaSBB \Rightarrow aab$ $SABB \Rightarrow aabaSBAB$	$<S \Rightarrow aSB \Rightarrow aa$ $SBB \Rightarrow aabSAB$	$\ldots v_5.v_5.v_{13}.v_2.$ $v_2.v_3.v_7.v_4.v_5$
20	5	$\ldots w_5.w_5$	$<S \Rightarrow aSB \Rightarrow aaSBB \Rightarrow aab$ $SABB \Rightarrow aabaSBABB$	$<S \Rightarrow aSB \Rightarrow aa$ $SBB \Rightarrow aab$ $SABB$	$\ldots v_5.v_5$
...	...	...	...	...	...
30	5	$\ldots w_5.w_5.w_{13}.w_2.w_2.$ $w_3.w_2.w_{10}.w_5.w_4.$ $w_5.w_5$	$<S \Rightarrow aSB \Rightarrow aaSBB \Rightarrow aab$ $SABB \Rightarrow aabaSBABB$ $\Rightarrow aabaλBABB$	$<S \Rightarrow aSB \Rightarrow aa$ $SBB \Rightarrow aab$ $SABB \Rightarrow aaba$ $SBABB$	$\ldots v_5.v_5.v_{13}.v_2.$ $v_2.v_3.v_2.v_{10}.v_5.$ $v_4.v_5.v_5$
...	...	...	....	...	...
39	12	$\ldots w_5.w_5.w_{13}.w_2.w_2.$ $w_3.w_2.w_{12}.w_{11}.$ $w_{12}.w_{12}$	$<S \Rightarrow aSB \Rightarrow aaSBB \Rightarrow aab$ $SABB \Rightarrow aabaSBABB$ $\Rightarrow aabaBABB \Rightarrow aababab$	$<S \Rightarrow aSB \Rightarrow aa$ $SBB \Rightarrow aab$ $SABB \Rightarrow aaba$ $SBABB \Rightarrow aaba$ $BABB$	$\ldots v_5.v_5.v_{13}.v_2.$ $v_2.v_3.v_2.v_{12}.$ $v_{11}.v_{12}.v_{12}$
40	14	$\ldots w_{14}$	$<S \Rightarrow aSB \Rightarrow aaSBB \Rightarrow aab$ $SABB \Rightarrow aabaSBABB \Rightarrow aaba$ $BABB \Rightarrow aababab>$	$<S \Rightarrow aSB \Rightarrow aa$ $SBB \Rightarrow aab$ $SABB \Rightarrow aaba$ $SBABB \Rightarrow aaba$ $BABB \Rightarrow aababab>$	$\ldots v_{14}$

Thus, the solution obtained for the MPCP is as follows:

$$w_1.w_7.w_{13}.w_2.w_7.w_5.w_{13}.w_2.w_2.w_8.w_5.w_5.w_{13}.w_2.w_2.w_3.w_7.w_4.w_5.w_5.w_{13}.w_2.w_2.w_3.w_2.w_{10}.$$
$$w_5.w_4.w_5.w_5.w_{13}.w_2.w_2.w_3.w_2.w_{12}.w_{11}.w_{12}.w_{12}.w_{14}$$

=

$$<S \Rightarrow aSB \Rightarrow aaSBB \Rightarrow aabSABB \Rightarrow aabaSBABB \Rightarrow aabaλBABB \Rightarrow aababab>$$

=

$$v_1.v_7.v_{13}.v_2.v_7.v_5.v_{13}.v_2.v_2.v_8.v_5.v_5.v_{13}.v_2.v_2.v_3.v_7.v_4.v_5.v_5.v_{13}.v_2.v_2.v_3.v_2.v_{10}.v_5.v_4.v_5.$$
$$v_5.v_{13}.v_2.v_2.v_3.v_2.v_{12}.v_{11}.v_{12}.v_{12}.v_{14}$$

Having established the one-to-one relationship between derivations in unrestricted grammars and (M)PCP solutions, we can now reason as follows. If (M)PCP was solvable, that is, if there was an algorithm which, given any instance of (M)PCP, would either give a solution or halt and declare that there is no solution to the given problem without ever going into an infinite loop, then we can convert this algorithm to a member computer for the language of the grammar. That is, the converted algorithm, given any string, would be able to determine if the given string belongs to the language of the grammar *or not.*

However, we know (from Chapter 11) that the language of an unrestricted grammar is a recursively enumerable language. If all recursively enumerable languages had such membership computers, they would all be recursive. This is a contradiction since we have already shown using diagonalization in Chapter 11 that not all recursively enumerable languages are recursive. Therefore there cannot be an algorithm for solving (M)PCP.

A formal proof of the equivalence of derivation in an unrestricted grammar and concatenation in MPCP would have to apply mathematical induction on the length of the derivation; the details of the inductive proof are rather uninteresting and are omitted. Similarly, one can also show that solving an MPCP through a series of concatenations mimics the computation history of a Turing machine. Starting with the initial configuration of the machine, the first string to be concatenated represents the start state of the machine with the input string being written on its tape. Each subsequent concatenation corresponds to a state transition and represents the new state of the Turing machine as well as its tape contents to the left and right of the reading head (as shown in Chapter 10, Sec. 10.3). A solution to MPCP is obtained if the last concatenation represents the Turing machine reaching a final state. As we will see below, if every instance of MPCP was solvable, without getting into an infinite loop, the corresponding Turing machine would not ever suffer from the halting problem.

12.3 Understanding the Halting Problem of Turing Machines

Let us consider a Turing machine M_{oracle} with two final states q_{True} and q_{False} as shown in Fig. 12.2. Its input is any statement, say a theorem in mathematics, suitably encoded. If the input statement is true, the machine halts in q_{True} and if it is false, it halts in q_{False}. In other words, it is an ultimate oracle that can answer any question in the world (albeit only those worded so as to have a Boolean answer)! We will now show that such a machine (also known as a universal *decider machine*) cannot exist.

If such a machine existed, mankind could figure out the truth (or falsehood) of any statement in any field (as long as the statement could be encoded and represented in a formal language). The impact of

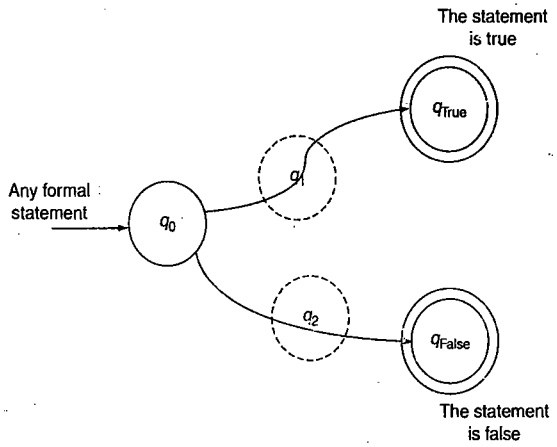


FIGURE 12.2 The fictitious universal oracle M_{oracle}.

such a machine on various fields including scientific research, law and justice, religion and spirituality would be beyond imagination. However, a well-known result in mathematics from Kurt Gödel, called *Gödel's Incompleteness Theorem* showed in the last century that there exist statements in any formal system (e.g., number theory, predicate logic, etc.) that can be neither proved nor disproved. And Gödel was able to prove this theorem!

Why is it so? Why is it not possible to establish the truth condition of an arbitrary statement mathematically?

Alan Turing showed soon after Gödel's theorem was published that the reason for such a limitation was that while any computable function can be computed by a Turing machine, there exist problems that are not computable because Turing machines suffer from the halting problem. The universal oracle M_{oracle} cannot be built because it would go into an infinite loop and never halt when asked to establish the truth condition of certain statements; the same statements that Gödel said suffer from the incompleteness of the formal system in which they are represented.

How can the machine M_{oracle} not reach one of its two final states after going through an arbitrarily large number of transitions from its start state, given that it is deterministic and has only a finite number of states?

1. It can halt in a non-final state, in which case it has rejected the input (i.e., the input statement is false).
2. It goes into a loop among its states and never halts.

Let us see by going through a proof by contradiction why it is not possible to construct M_{oracle} such that it never goes into an infinite loop. Let us assume that it is possible to construct the universal oracle machine M_{oracle}. Let us apply a construction by modifying the final states as follows. Make the final state q_{True} non-final and add a new state q_{new} and an infinite loop between q_{True} and q_{new} as shown in Fig 12.3. On reaching the state q_{True}, for any symbol s on the tape, the machine goes to the new state q_{new} moving left on the tape and for any next symbol s on the tape (which may be different from the symbol to its right), it comes back to q_{True}, moving right on the tape. Thus, the new machine M_{mod} goes into an infinite loop whenever the original machine halts in its final state q_{True}.

Now we consider a particular kind of input statement to the unmodified universal oracle M_{oracle}, namely, "Will a given Turing machine M halt and accept a given input w?" We encode the given Turing machine M as a binary string (see Sec. 10.9 on universal Turing machines in Chapter 10) and append it to the given string w to generate the input string for M_{oracle}. If M halts on input w, our oracle M_{oracle} will halt in q_{True}; if M does not halt on w or halts and rejects w, then M_{oracle} halts in state q_{False}.

Note that if such an oracle M_{oracle} could be built, no Turing machine would suffer from the halting problem. We could easily construct them so that they would first invoke (as a "subroutine") the machine M_{oracle} to find out if there was a halting problem *before* processing a given input. This would be similar in a sense to a modern operating system checking an executable program for viruses before running it.

Continuing with the proof by contradiction, consider now the behavior of M_{mod} given w and M as input. If M halts and accepts w, M_{mod} goes into an infinite loop! And if M either rejects w or fails to halt on w, M_{mod} halts in state q_{False}!

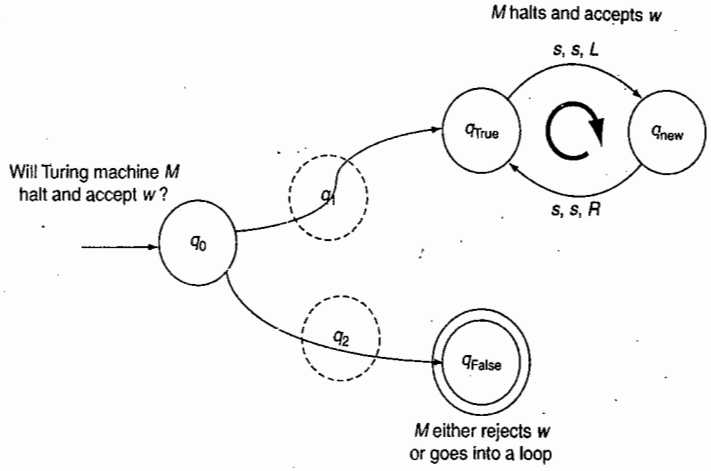


FIGURE 12.3 Modified universal oracle M_{mod}.

Finally, we play the trick of "a snake eating its own tail," that is, we give the binary encoding of M_{mod} itself (along with some input string w) as the input to M_{mod}. In other words, the input data to this "program" is the "program" itself. We are thereby asking the question: will M_{mod} halt on the input w and we are asking this question to M_{mod} itself! Given the definition of M_{oracle} and the modification we did to arrive at the design of M_{mod}, we are now forced to conclude that

1. M_{mod} will halt in its state q_{False} whenever M_{mod} rejects its input or fails to halt.
2. M_{mod} fails to halt and goes into an infinite loop through its state q_{True} whenever M_{mod} halts and accepts its input.

In other words, M_{mod} halts whenever it does not halt and it does not halt whenever it halts. This is obviously impossible and ridiculous. It is a contradiction. Therefore, our assumption that we can construct a universal oracle M_{oracle} must have been wrong. There is no such *decider machine* and Turing machines do suffer from the halting problem.

12.4 *The Idea of* Problem Reduction

We can also make an argument known as problem reduction to show that PCP is unsolvable. If PCP was always solvable, that is, if there was an algorithm for PCP that did not suffer from the halting problem (i.e., it did not go into an infinite loop for any inputs), then the correspondence we established above would show that unrestricted grammars (i.e., their parsers or generators) would not suffer from the decidability problem and therefore even Turing machines would not have the halting problem. But we showed above using the M_{oracle} and M_{mod} machines that such is

not the case. Turing machines do suffer from the halting problem and therefore PCP is unsolvable. This technique of proof is called *problem reduction*. A problem P_1 is shown to be unsolvable by reducing another problem P_2 to P_1 where P_2 is known to be unsolvable. If P_1 were to be solvable, then P_2 would also become solvable via the reduction, thus showing us that P_1 is indeed unsolvable. Similar arguments are used in complexity analysis to show for example that a given problem is NP-Complete (see Sec. 12.9).

12.5 The Halting Problem and Recursive Languages

To recap, we now relate the halting problem of Turing machines to recursively enumerable languages. We know from the diagonalization argument that recursive languages are a proper subset of recursively enumerable languages (see Chapter 11). This is because only acceptor Turing machines exist for recursively enumerable languages that are not recursive and this is due to the halting problem of Turing machines. Recursive languages, on the other hand, have decider Turing machines which do not suffer from the halting problem.

Earlier, in Chapter 11, we saw that if the complement of a recursively enumerable language is also recursively enumerable then the language is recursive. As such, there are recursively enumerable languages whose complements are not recursively enumerable.

Why are their complements not enumerable?

Given a string w_{good} which belongs to the closure of the alphabet Σ^*, if w_{good} belongs to the language L, then there is an acceptor Turing machine $M_{acceptor}$ which halts and accepts w_{good}. If a string w_{bad} does not belong to L, a Turing machine for L may or may not halt for w_{bad}. If there is a machine $M_{decider}$ which always halts for every w_{good} as well as every w_{bad}, then the language is recursive. If all machines for the language are such that they halt and accept every w_{good}, halt and reject some w_{bad} but fail to halt for some other w_{bad}, then the language is recursively enumerable but not recursive.

An *acceptor* $M_{acceptor}$ is a Turing machine which halts and accepts every string w_{good} that belongs to the language but may not halt and reject when presented with a string w_{bad} that does not belong to the language (i.e., it may go into an infinite loop). A *decider* machine (or a *membership function*) $M_{decider}$ is a Turing machine which halts and accepts every w_{good} and halts and rejects every w_{bad}. $M_{acceptor}$ machines exist for all recursively enumerable languages but $M_{decider}$ machines exist only for recursive languages.

If there was no halting problem, every Turing machine would always halt given any input w_{good} or w_{bad}. We could always construct a machine $M_{decider}$ for every recursively enumerable language such that it goes to appropriate final states q_{True} and q_{False} for w_{good} and w_{bad}, respectively. As seen in Chapter 6 (Sec. 6.1.4) in the case of finite automata, it would be easy to modify this machine by exchanging its two final states to construct an acceptor for the complement of the language. By combining the two acceptors, we obtain a decider for the language. This decider would work around the halting problem of the Turing machine by first checking (using this miracle solution to the halting problem) whether the machine would halt. The machine would be run on the input only when there is a guarantee that it would halt on that input. This way, all recursively enumerable languages would have a membership function and become recursive. This would be contradictory with the result we obtained through diagonalization in Chapter 11. Hence, there cannot be a solution to the halting problem of Turing machines.

12.6 *The Ideas of* Computability and Decidability

A mathematical function f: $D \rightarrow R$ from a domain set D to a range set R is *computable* if there exists a Turing machine M that, given any input string w from the domain D, goes from its start state to a final state and halts there with $f(w)$ written as the output on the tape. Being a mathematical function, f would also require that there is only one value for the output $f(w)$ for a given input w, but that is not our primary concern here; we could very well be talking about computable relations, not just functions.

If the function f is a Boolean function, then the output $f(w)$ written on the tape is not significant; what matters is whether the machine halts in an accepting final state q_{True} or *halts* in a rejecting final or non-final state q_{False}. Such Boolean functions are called *decidable functions*.

As noted in Chapter 1, we can convert any function to a Boolean function by including the answer in the input. For example, instead of asking "what is 3 + 4?" we can ask "is 3 + 4 equal to 7?," "is 3 + 4 equal to 6?" etc. Considering the set of all such strings, we can now say the function of adding two numbers or the problem of addition is in fact a formal language of all strings of the form "is $n_1 + n_2$ equal to n_3?" wherein actually $n_1 + n_2 = n_3$. If the sum n_3 is not right, the string does not belong to the language of addition. This is the equivalence of classes of problems and formal languages and this is why we rarely talk about problem solving in this subject. Rather, we talk about problems in terms of their statements, that is, strings belonging to the corresponding formal languages.

If a type of problem or, equivalently, a formal language has a decider Turing machine, then it is said to be computable or decidable. In other words, all recursive languages are computable or decidable. For recursively enumerable languages, on the other hand, its acceptor machine can only recognize a correct solution to the problem, that is, a string that belongs to the language. It cannot compute a solution to the problem, that is, it cannot always reject a string that does not belong to the language by recognizing that it represents an incorrect solution to the problem. It is as though there are too many problems in the world and not as many solutions! The PCP is an example of a problem that is not computable. There is no algorithm for solving it and we say it is undecidable. We will see below other examples of practical problems that are undecidable.

In summary, we know that:

1. PCP is an unsolvable problem, that is, it is not computable because if it was, then all recursively enumerable languages would become recursive. This is a contradiction since we already know of the existence of languages such as L_{Diag} and $L_{Non\text{-}RE}$ (see Chapter 11) found by the method of diagonalization.
2. Turing machines suffer from the halting problem for the same reason. If there was no halting problem, we could convert every acceptor machine to a decider, thereby making every recursively enumerable language recursive.
3. The halting problem of Turing machines shows why certain functions are not computable (and also explains the reason for Gödel s incompleteness theorem).
4. This limitation of the very idea of computation is applicable to many practical problems such as the PCP in string matching, the problem of equivalence of two context-free grammars, and many others that we list in the following section.

12.7 Undecidable Problems

The *halting problem of Turing machines* is the canonical unsolvable problem. There is no algorithm that takes an arbitrary Turing machine and its input string and says whether that machine will halt on that input. We have also seen that the PCP is undecidable.

What other problems are undecidable or not computable or algorithmically unsolvable?

Several other problems can be shown to be undecidable by reducing the halting problem of Turing machines to them. If the problems were decidable, the halting problem would also become decidable because of the reduction. This we know is not true. Therefore, those problems must also be undecidable.

Using the technique of problem reduction, we can show that the following problems are undecidable:

1. Various problems in predicting the behaviors of Turing machines such as the state entry problem in a Turing machine: Given a Turing machine M and input w, will M enter a particular state q_i in processing w?
2. *Blank-Tape Halting Problem*: Given a blank tape as input, will a Turing machine M halt?
3. Given an unrestricted grammar G, is G.language empty?
4. Given a Turing machine M, is M.language infinite?
5. Given a Turing machine M, is M.language regular or context-free?
6. A generalization of the above problems, called *Rice's Theorem*, says that any non-trivial property of recursively enumerable languages is undecidable! A trivial property is one that is either true for all elements of a set or false for all elements of the set. For example, the property that all recursively enumerable languages have an acceptor Turing machine is trivial. A non-trivial property here is one which is true for some recursively enumerable languages and false for other recursively enumerable languages. For example, the property that only some of them are recursive or that only some of them are context-free is non-trivial.
7. Equivalence of context-free grammars: Suppose two designers have independently developed two different grammars for a language (e.g., a new programming language). Do the two grammars represent the same language? That is, given G_1 and G_2 as two context-free grammars, the problem of answering is G_1.language = G_2.language is undecidable.
8. The problem of determining, given two context-free grammars G_1 and G_2, whether the intersection of G_1.language and G_2.language is null, is undecidable, that is, we can't even determine whether there is a common string between the two languages.
9. Given a context-free grammar G, determining whether G is ambiguous is undecidable
10. Given a context-free grammar G, determining whether G.language = Σ^* is undecidable.
11. Equivalence of pushdown automata is also undecidable. Just like their context-free grammar counterparts, we cannot write an algorithm that determines if the languages of two pushdown automata are the same.
12. We were able to minimize a deterministic finite automaton (see Chapter 3). There is no minimization algorithm for a pushdown automaton. Finding an equivalent pushdown automaton with a minimum number of states is undecidable.
13. Even the membership problem in an unrestricted grammar is undecidable. Given a grammar G and a string w, it is undecidable whether w is a member of G.language! In other words, unrestricted grammars are too powerful to even construct a parser that accepts all strings in the language of the grammar and rejects all non-members.

14. On the same lines, the language of all strings accepted by corresponding Turing machines is undecidable. That is, we cannot determine whether a given string is accepted by a given Turing machine (since, if we could do this, there would be no halting problem of the Turing machine).
15. The problem of inference or semantic entailment in first-order predicate logic is undecidable. Predicate logic is often used to represent knowledge and make inferences in verifying the correctness of a program and in applications such as expert systems and other intelligent applications. Unfortunately, the problem of determining whether a given statement is true based on a set of axioms according to the rules of deduction in predicate logic is undecidable.

These are examples of the *limitations of algorithmic computation.* One might as well have called them the *limitations of computation,* since we have defined computation as algorithmic; we don't know of any other kind of computation. Of course, it is still possible to develop approximate or partial solutions to these problems. One can also put a time limit on the execution of a computing machine to obtain a partial solution; if the machine has indeed gone into an infinite loop, its execution would be truncated. However, it is quite possible that for some inputs the machine was still computing a solution without being stuck in an infinite loop when the timeout killed the computation.

12.8 Other Models of Computation

The Turing machine is not the only model of computation. Several other models have been developed, each of which can be shown to be exactly equivalent in their computing power to the Turing machine. For example, Post machines and Markov algorithms are equivalent to recursively enumerable languages and Turing machines. There is no model that is more powerful than Turing machines (as already noted in the Church–Turing thesis in Chapter 10).

The theory of recursive functions explores recursive definitions of all computable functions. For example, addition of two natural numbers m and n can be defined as

```
add(m, 0) = m;
add(m, n+1) = successor(add(m, n));
```

where `successor()` is a function that returns the next higher number in the sequence of natural numbers. Such functions are called *primitive recursive functions.* Similarly, multiplication can be defined recursively as

```
multiply(m, 0) = 0;
multiply(m, n+1) = add(m, multiply(m, n));
```

It turns out that the set of all primitive recursive functions is countably infinite but the set of all possible functions is uncountably infinite. To define more complex computable functions that are not primitive recursive, a *minimum function* that selects the least possible value from its input set is necessary. With the minimum function introduced, we can define more complex functions known as *μ-recursive functions*. It turns out that they are exactly equivalent to recursively enumerable languages, that is, the languages of Turing machines. In other words, every Turing computable function is a *μ-recursive function*. Similarly, a formalism known as *lambda calculus* is also equivalent to Turing machines and computable functions.

An example of a function that is computable, that is, μ-recursive but not primitive recursive, is the Ackermann function:

```
A(m, n) = n + 1 if m = 0
A(m - 1, 1) if m > 0, n = 0
A(m - 1, A(m, n - 1)) if m > 0, n > 0
```

Ackermann function grows very rapidly. For example,

```
A(2, 2) = 7
A(3, 2) = 29
```

and $A(4, 2) = 2^{65536} - 3$.

12.9 Computational Complexity: Efficiency of Computation

Consider functions which are computable, that is, those for which a Turing machine decider exists so that we know whether a given answer to a problem is right or wrong. Such computing problems can be solved by developing a software program that is equivalent to the Turing machine. Although a well-designed program in a modern programming language is often much more efficient than the Turing machine, it is important to know how efficient it can be. Are there different levels of inherent difficulty or *complexity* in various computable problems because of which their programs cannot be more efficient than a limit or bound for a given problem?

This question is studied extensively in the subject of *computational complexity* which is usually a part of *analysis and design of algorithms*. Complexity theory classifies computing problems into complexity classes based on the inherent difficulty of problems. Problems that belong to the same complexity class can be solved within a specified amount of time and memory resources for that particular class. A brief account of the essential ideas in the study of the complexity of computing problems is presented here in the context of computability and undecidability. While computability is about the theoretical limits of computation, complexity analysis is about the practical limitations of computing.

Usually, the difficulty or complexity of computing is not an issue for small problems. For example, if we are searching for a data element in a small collection, sorting a short sequence of data elements, or finding a path in a graph with a small number of nodes, the efficiency of the algorithm we choose for the problem does not make a big difference. This is somewhat similar to every finite language being regular, and, therefore easily manageable. As the size of the problem increases, on the other hand, the complexity of the problem itself and the efficiency of the algorithm applied for solving the problem become so critical that an incorrect understanding of the complexity or inappropriate choice of algorithm can be the difference between a great computing solution and an infeasible or *intractable* one.

Computational complexity is measured usually in terms of the resources such as time and space (i.e., amount of memory) required to solve the problem. Instead of measuring actual time in seconds or the actual amount of memory in bytes, the minimum number of steps (i.e., transitions) or the number of tape cells required to solve a problem in a standard deterministic Turing machine is used. This number, known as *time complexity* if it is the number of steps and *space complexity* if it is the number of memory cells, is usually expressed as a function of the size of the input data given to the algorithm (the size being denoted by n). Time or space complexity can differ between different instances of the same problem.

For example, some sorting algorithms are very efficient when the input data is already (almost) sorted; some take the longest when the input is randomly ordered or sorted in reverse order. To account for such differences among problem instances and among different algorithms for solving the same problem, computational complexity is measured in three ways:

1. **Best case complexity:** What is the minimum number of steps required to compute a solution for the best possible input for the algorithm and problem?
2. **Worst case complexity:** What is the minimum number of steps required to compute a solution for the worst possible input for the algorithm and problem? This is usually dependant on the algorithm since one algorithm can be more efficient than another for the same problem. However, problems themselves can be shown to have a lower bound on their complexity in the sense that no algorithm for that problem can be more efficient, in the worst case, than the lower bound.
3. **Average case complexity:** What is the minimum number of steps required to compute a solution on an average for all possible inputs for the algorithm and problem?

Once again, it may be noted that all of the above measures are defined in terms of the number of steps involved in the computation, not in terms of actual time or cost. While this abstracts out factors such as the speed of a currently available computing machine or the overheads of a particular operating system, it still includes certain fundamental assumptions about the computing model involved. Essentially, these measures of complexity are based on Turing's model of a computing machine with its stored table of instructions (e.g., a compiled program) loaded in memory which is itself accessed by a reading head that moves sequentially along a linear sequence of memory cells. A unit of computation for a given problem is based on such restrictions imposed by the model of computing. A unit is assumed to take a constant (but unknown) amount of time to execute.

A key question is as n grows large, *how fast does the complexity grow?* The ideal case is linear, that is, the number of steps involved grows linearly with the problem size n. Rarely, we encounter problems that can be solved even faster, such as, for example binary search in an array which can be performed in time (or number of steps) that is logarithmically related to n, or $\log n$. Some other problems such as sorting take $n \log n$. In general, an algorithm that, in the worst case, takes time (or number of steps) that is a polynomial in n is manageable or tractable. For example, in the worst case, quick sort is quadratic in n; we say it is an n^2 algorithm. We ignore any constant factors such as the a and b in the actual number of steps which may be $an^2 + b$ since they do not grow as n increases. The difference between the constants and the problem size n is most significant for very large values of n. For example, if you consider the size of a popular social networking platform viewed as a graph, n is of the order of hundreds of millions. Only highly efficient algorithms are feasible for such large data sets; even a quadratic algorithm that operates globally on such a large graph would never compute anything useful in a reasonable amount of time no matter how fast and powerful the computing hardware is.

Although a problem or algorithm that takes n^3, for example, takes much longer to compute than another that takes only n or n^2, any polynomial algorithm is still manageable. Intractable problems are those that take an exponential amount of time due to a combinatorial explosion in the number and series of choices that arise in solving the problem. For example, a problem that takes 2^n or n^n is intractable in the sense that as n grows large, the computing solution takes too long to finish for it to be useful or practical. There are many such problems that are well known, for example, the Traveling Salesman Problem of finding an optimal route through n cities without visiting any city more than once.

The complexity class DTIME($f(n)$) is the set of all problems that can be solved in a deterministic Turing machine within $f(n)$ number of steps, where n is the size of the problem. For example, sorting is a DTIME($n \log n$) problem since algorithms are available that can sort a list of n data elements in $n \log n$ steps in the worst case. Similarly, the class of all problems that can be solved in polynomial time, that is, DTIME($f(n)$) where $f(n)$ is a polynomial function in n, is called the class P. Ignoring minor terms in the polynomial function, problems in the class P are usually described as n^2 problems, n^3 problems and so on.

On the same lines, EXPTIME is the class of problems that take time exponential in n to solve (e.g., 2^n). Problems in EXPTIME are considered intractable since exponential functions grow too fast as n increases. For example, if $n = 100$, 2^n is so large that it would take more than ten billion years to complete even with a computing machine that can process a trillion transitions per second!

Another class of problems known as NP problems or *non-deterministic polynomial problems* are those that take exponential time but would take only polynomial time if non-deterministic parallelism (of the kind that we saw in non-deterministic finite automata in Chapter 3) is available. The number of steps for this class of problems is a polynomial function in n given a non-deterministic Turing machine. We must note here that although non-deterministic Turing machines are equivalent to deterministic Turing machines in their computing power (i.e., which set of functions they can compute), they take fewer steps to compute than a deterministic machine. This is similar to non-deterministic finite automata and deterministic finite automata being equivalent while a typical non-deterministic finite automaton (NFA) has far fewer states than the corresponding deterministic finite automaton (DFA). There are a number of NP problems such as the Boolean Satisfiability problem in logic, the Hamiltonian Path problem and the vertex cover problem in graph theory, the integer programming problem in operations research and the protein structure prediction problem in biochemistry.

Complexity theory further classifies NP problems using the idea of problem reduction into the classes of *NP-Complete* and *NP-Hard* problems. In general, all problem reductions considered are expected to take only polynomial time in a deterministic Turing machine. NP-Complete problems are those NP problems for which, given a solution, its validity can be verified in polynomial time. NP-Complete problems are the hardest problems in the class NP. They are also NP-Hard problems (see below) such that any NP problem can be reduced to the problem in polynomial time. They are usually decision problems.

NP-Hard problems are at least as hard as the hardest known NP problem. A problem is NP-Hard if and only if there exists an NP-Complete problem that is polynomial-time reducible to it. However, NP-Hard problems need not even be in the class NP. For example, the halting problem is NP-Hard but not NP-Complete. NP-Hard problems are usually optimization problems such as the well-known Traveling Salesman Problem. Further, it has been shown that there exist NP-Hard problems that are neither NP-Complete nor undecidable.

Figure 12.4 shows the relationships among various complexity classes. Also shown in the figure is the new class PSPACE which is defined as the set of problems that take an amount of space (i.e., memory) that is polynomial in the size of the input. It turns out that the class PSPACE is larger than NP but smaller than EXPTIME. Moreover, it has been shown that the classes PSPACE and NPSPACE, that is, the set of problems that require a polynomial amount of space in a non-deterministic Turing machine, are the same.

It may also be noted that there are NP problems that are neither P nor NP-Complete. An example of such a problem is the graph isomorphism problem in graph theory which asks, given two graphs with the same number of vertices and edges, whether one graph can be transformed to the other without altering any edge.

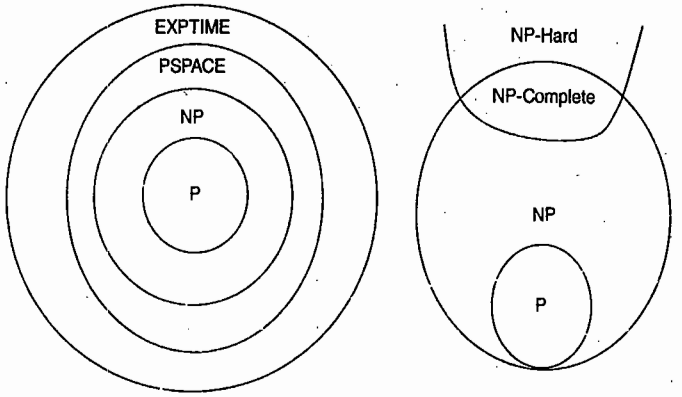


FIGURE 12.4 Complexity classes.

12.10 P = NP?

One of the biggest unsolved questions in computer science is whether P = NP. That is, whether problems in the class NP can be solved as efficiently as polynomial problems. In other words, the question is whether an efficient implementation of non-deterministic parallelism can be found so that all computable problems can be solved efficiently. You will remember from Chapter 3 that converting a non-deterministic automaton to a deterministic one increases the number of states in the machine, to a number that is, in the worst case, exponential in the number of states in the NFA. Although we were not at all concerned with the efficiency of computation in finite automata or other computing machines throughout this book, it must be noted here that the equivalent DFA obtained from the NFA is not efficient, since a larger number of states in an automaton corresponds to longer programs or more steps in an equivalent algorithm. The NFA itself is efficient only if the kind of non-deterministic parallelism that we imagined (in Chapter 3) can actually be implemented as efficiently as deterministic sequential processing.

The question of whether P = NP is significant not just from the point of view of the theory of computational complexity. From a practical point of view, if the two classes are indeed equal, the consequences for computer and information security and the entire information technology industry would be disastrous! If they are equal, our current reliance on their non-equivalence to build encryption technology would create a huge security loophole. It takes exponential time to decrypt and break an encrypted password or other coded data on which all on-line transactions, data management and Web technologies depend. If an efficient polynomial time algorithm could be devised to break and decode an encrypted message, no on-line system would be secure anymore.

Fortunately, no one has been able to show that P is in fact equal to NP. Even recent attempts at providing lengthy proofs which are yet to be accepted or rejected by peer communities of scholars have tried to show that P is *not equal* to NP.

12.11 Open Questions

In this book, we have raised and answered a number of interesting questions about computing and computers. We have seen what constitutes a most abstract and minimal computing machine, what are

the effects of adding various kinds of memory to such a machine, how to define or represent a language very compactly using grammars, whether there are problems that are not computable, why isn't everything computable, and so on. Before concluding, it is legitimate to ask whether there are any unanswered questions about the ideas of computing and computing machines (apart from P = NP).

Here are several questions some of which have been answered in this book and some of which, a discerning reader may be able to answer:

1. Why are deterministic and non-deterministic finite automata equivalent but not -deterministic and non-deterministic pushdown automata?
2. Why does it take non-determinism to deal with context-free grammars?
3. If we allow the input to be processed in two left-to-right passes instead of one, will the need for non-determinism in context-free grammars go away?
4. In non-determinism, who throws the dice to get it right every time? Is it magic?
5. Can we ever build a non-deterministic machine? Can unlimited parallelism implement non-determinism? Using multi-core processors, multi-threading or massively parallel computing?
6. Is there a different, new model of the very idea of computing that will change all of these equivalences and distinctions? Will there be a model of computing that is more powerful than the Turing machine?
7. Do these statements hold good in radically different computing architectures (e.g., quantum computing)?
8. Can the human mind handle non-determinism? Without a serialized backtracking implementation?

In the following section, we consider the question of whether the Turing machine (or any of its equivalents) is the right model of computing for making further progress in computer science and the development and application of computing solutions for the betterment of human life.

12.12 Recognition Versus Recall

We have seen that for undecidable problems there can never be an algorithmic solution or its implementation in a program. What if we change the very idea of computing? Let us rclook at the limitations of Turing computability and compare the abilities of algorithmic computing in Turing machines with the way the human mind appears to work.

The language L_{Diag} from Chapter 11 which is recursively enumerable (but its complement $L_{Non\text{-}RE}$ is not,) is a very peculiar language:

1. *It has an enumeration procedure*: Any finite string in this language can be enumerated (i.e., printed, displayed, etc.) by a proper enumeration procedure in finite time (although the set itself is infinite and can never be exhaustively enumerated in finite time).
2. *However, it has no membership function*: It is not recursive, it is not decidable and its complement is not enumerable. Therefore, given a string there is no algorithm (i.e., a Turing machine) that can tell you if the string belongs to the language.

It is as though the Turing machine can recall and list any member of this language (or set), but it is unable to recognize any of them. Normal human cognitive behavior suggests the opposite: Recognition is easier than recall. If asked to recall the names and faces of all our classmates from a long time ago, we would find it impossible to list their names or bring back all their faces to our mind fully and accurately. On the other hand, if we hear the name or see the face of one of them, in most cases, we can easily recognize the name or the face, even after many years. Computation, as we have modeled it using Turing machines, however, seems to find it easy to enumerate (recall) than verify (recognize). What does this suggest to us? Perhaps, we have been doing it all wrong: this is not the right model of *computation* if our objective is to figure out what is going on in the human mind! You will recall that this was one of Turing's key objectives when he set out to study abstract models of computing machines.

The visibly differing ease with which recognition and recall problems are solved by modern computers based on Turing's model and the human mind is the reason why computer-based solutions find it hard to solve problems that are natural to us such as recognizing an object or face, recognizing (hand) written texts and recognizing and comprehending speech in natural languages. Perhaps other human abilities such as learning, decision making, intuition and common sense as well! Decades of research and development in artificial intelligence have failed to produce a computing model with abilities that are anywhere close to an ordinary human being. Human beings seem to solve problems intuitively, quickly and approximately with common sense and semantics; computers based on Turing's model seem to be stuck in sequentially searching for provably correct and globally optimal solutions, while having no big picture or intuition or sense of what they are doing.

While in no way implying to undermine the enormous contribution made to our field by Alan Turing, it is legitimate to ask whether we will have a better model of computing: One that can explain human cognitive behavior better than the Turing machine and all of what we call computation today, including artificial intelligence, neural networks, genetic algorithms, swarm intelligence and so on.

Why are computers stuck at the level of syntax? Why do computers deal with data and syntax-level processing very effectively but are unable to get to the semantic level? Why can't they get to meaning, knowledge and understanding? Are we going to see a breakthrough in our lifetimes?

We have been following the path defined by Alan Turing, the father of computer science, for over 70 years with great success except when it comes to modeling or reproducing the cognitive behavior and natural abilities of the human mind. We have failed to realize Turing's own prediction in 1950 that by the year 2000, there would be computers with so much memory that they will have abilities akin to humans. While we definitely produced computers with larger memory units than predicted by Turing, their abilities have remained more or less at the same level of a dumb machine. Is there a better alternative? If Turing was our Newton, who is our Einstein going to be? Will we have a better model of computing?

12.13 Key Ideas

1. A practical problem in string matching known as the Post Correspondence Problem (PCP) has no algorithmic solution. It is an undecidable problem.
2. It can be shown that solving the Post Correspondence Problem corresponds to parsing or deriving a string in an unrestricted grammar.
3. If PCP is solvable, then we have a membership function for all unrestricted grammars. This would make every recursively enumerable language recursive. This is not the case as we know from the diagonalization argument (from Chapter 11).
4. There is no algorithmic solution to PCP because of the halting problem of Turing machines.
5. Turing machines suffer from the halting problem when presented with certain inputs that do not belong to the language of the machine. This situation arises when the problem is undecidable, that is, when the language is recursively enumerable but not recursive.

6. It is not possible to build a universal oracle machine that would answer any question and determine the truth of any given statement. This is why all formal mathematical systems suffer from Gödel's incompleteness.
7. If there was no halting problem for Turing machines, we could build a decider machine for every recursively enumerable language, thereby making the language recursive. Again, this is not possible.
8. Apart from PCP, there are a number of undecidable (or incomputable) problems including some practical problems in context-free grammars. They show the limitation of algorithmic computation.
9. Other models of computation such as Post machines and recursive functions are equivalent to Turing machines.
10. Computable problems are classified into a number of complexity classes on the basis of the inherent difficulty of solving various problems. A problem is shown to belong to a class by reducing a known problem in the class to the given problem.
11. At this time it is unknown whether the class of problems that can be solved in polynomial time is equivalent to the class of non-deterministic-polynomial problems, that is, whether P = NP.
12. Modern computers based on Turing's model of a stored-program computing machine are very good at storing and recalling large amounts of data but they are unable to recognize faces, voices, handwritings or other patterns. The human mind, on the other hand, is very good at recognition and rather poor in memorizing and recalling large amounts of data. This difference between abilities for recognition and recall appears to indicate that the human mind does not work according Turing's model of computing.

12.14 Exercises

1. What is the difference between computability and decidability?
2. What is the relation between computing problems and formal languages?
3. Let the alphabet set be {0, 1}. Let A and B lists be as defined below. Show that the Post Correspondence Problem does not have a solution for (A, B).

i	List A: w_i	List B: v_i
1	011	101
2	11	011
3	1101	110

4. Consider life on Planet M in the Andromeda galaxy. There, the Post Correspondence Problem is decidable! What are the consequences of this? Explain clearly.
5. Show that greater(x, y) = 1 if $x > y$ else = 0 is computable.

   Hint: Define the function recursively as shown for *add* in Sec. 12.8.
6. Show that $g(x, y) = x^y$ is primitive recursive.

   Hint: Define the function recursively as shown for *multiply* in Sec. 12.8.
7. Show that subtraction is computable.
8. Compute the values of Ackermann function $A(m, n)$ for each of $m = 0, 1, 2, 3$ and $n = 0, 1, 2, 3, 4, 5$. *Hint*: Apply the recursive definition of the function starting from the smallest values of m and n and moving to higher values incrementally.

The Science of Computing: A Summary

This has been a study of the essence of computing: What computing means, what its power and limitations are, the nature of basic computing machines and the design of computing languages. *Theory of Computing* which is also known as *Formal Languages and Automata Theory,* is about understanding what *computing* is and what *computers* can and cannot do. It is essentially a study of classes of equivalent triads of *languages,* their *grammars* and *computing machines* that process them. It is the theory behind compilers and the design of programming languages including data representation, XML and other computer languages. One can be a programmer or a software engineer without understanding this theory, but surely not a *computer science* graduate.

We began the study of this subject with the simplest computing device, a *Finite-State Automaton* with no memory and a finite number of states. It can only remember things by changing states. Yet it can handle infinite languages, those that are called *regular languages.* It is useful to remember one *mantra* in particular for designing finite automata: *What in the world does the machine need to remember?*

We learned that non-determinism does not add any power to these machines; they are merely easier for us to construct. Non-deterministic finite automata (NFAs) can be reduced to deterministic finite automata (DFAs) by *subset construction.* DFAs can be further minimized by marking distinguishable states and reducing indistinguishable states for which *Table Filling* is a useful book-keeping tool.

Regular languages and regular expressions are a simple but practical class of formal languages. Regular languages are so well-behaved they have all the good properties including set-theoretic closures, membership and other questions. We found that while regular expressions were very useful for specifying and processing simple patterns in data, practical problems such as matching nested parentheses or nested if-then-else statements could not be handled by this simplest class of languages. We were able to demonstrate this by applying the *Pumping Lemma for regular languages.*

Regular languages have a repeating pattern. This is the gist of the *Pumping Lemma* (based on the *Pigeonhole Principle*) that we use in a mock adversarial game to show that non-regular languages do not have such a pattern. If we use variables to denote and re-use such patterns, we arrive at the idea of a *grammar.* The simplest notation for a "grammar" for a regular language is the *regular expression* (*RegEx*). RegEx's can be converted to NFAs and DFAs can be converted to RegEx's using the idea of a *generalized transition graph* and *state elimination* (which we also called building "flyovers").

When asked to construct a DFA or to write a regular expression or a grammar, it is useful to always start with the simplest case and then add all the other bells, whistles, exceptions and boundary conditions. Remember that whenever a DFA must change states to remember something, its regular expression moves to the next concatenated element and the grammar introduces a new variable.

We introduced the larger and highly practical class of languages known as *context-free languages* and *grammars* and studied how they can be *parsed* or *derived* in a compiler. We need them to be in neat forms to make parsing convenient and efficient: hence *Chomsky Normal Form* (that always grows or shrinks the parser's stack by exactly one) and *Greibach Normal Form* (that always introduces exactly one terminal symbol). Our *mantra* to remember the proper order for cleaning up grammars was *lambda units are useless.* The *CYK algorithm* (also a *table filling* algorithm) is a neat way to implement a parser for context-free languages (although by no means the most efficient way), thereby enabling the construction of compilers for high-level programming languages.

Computing devices called *pushdown automata* (PDAs) are finite automata that are provided with a single stack memory. Their stack behavior of accessing only the top of the stack is equivalent to context-freeness in grammars. In the other direction, a leftmost derivation with a grammar in Greibach Normal Form is equivalent to a PDA's behavior.

Deterministic PDAs (and deterministic context-free languages) are not the same as non-deterministic PDAs [and context-free languages (CFLs)] because non-determinism, the ability to try out multiple options in parallel with potentially different stack contents to come out with "the lucky guess" cannot be simulated in a deterministic single-stack machine.

The *Pumping Lemma* for CFLs shows that they have two repeating patterns that dance in perfect tune with each other to generate infinitely many strings in the language. The lemma has the same use as before with a similar adversarial game to show that languages such as *ww* are not context-free.

Closure properties of CFLs showed a few surprises; their behaviors are not quite so "regular" any more. In later chapters, we learnt that although programming languages are for the most part context-free, they need some constructs that are not context-free. We understood how context-sensitive languages and grammars can be applied to model such constructs in modern programming languages.

Remember, an alphabet is a finite set of symbols; its closure is an infinite set of strings; formal languages are any possible subsets of this set of strings. Smaller languages can be "more complex" than their supersets. The largest language, that is, the closure of the alphabet is always a regular language!

We learnt that even before modern computing machines were invented, in 1936 to be precise, Alan Turing had described a most general and most powerful computing device known as the *Turing Machine*. We learnt how Turing machines could solve any problem that is solvable. Turing machines are the most generic computing devices known to us. They have infinite memory with a sequential access and a finite control, that is, a finite program endowed with an unlimited memory. A *computation* is what a Turing machine does and an *algorithm* is a computation that can be done with a guarantee by a Turing machine. All variations of Turing machines are exactly as powerful as the standard ones.

Turing's universal machine was also a stored-program computer, the architecture and organization of which have now been realized in the form of digital computers. A *Universal Turing Machine* is a most generic *programmable computer;* it can store the program of another Turing machine (encoded as a binary string) and simulate its execution. However, Turing machines suffer from *the halting problem:* They may try endlessly to decide certain inputs. This is because many languages are not *decidable.* There are only countably infinite (i.e., as many as positive integers) Turing machines but there are an uncountably infinite number (i.e., as many as real numbers) of languages. This is why not all *recursively enumerable languages* are *recursive.* This is why some problems are unsolvable: There can never be algorithms for seemingly practical problems such as determining if a context-free grammar (CFG) is ambiguous or if two grammars are the same or for the *Post Correspondence Problem.* The complete

picture of the relationships between the various classes of formal languages, grammars and computing machines was shown in Chapter 11 (see Fig. 11.12).

Along the way, we also studied the largest class of formal languages, namely, recursively enumerable languages and touched upon important ideas in the computational complexity of solvable problems. Apart from the theoretical limitations of computing shown by Gödel, Church and Turing, we also briefly considered how, in comparison to the human mind, modern computers based on Turing's model of computing are severely limited in their abilities to recognize while being incomparably better in memorizing and recalling large amounts of information.

Hopefully, you have obtained an overview of the science of computing, the very core of what we call computer science. You are now ready to take up further studies in more advanced and applied topics.

Appendix A

A Formal Language Quiz

A. True or False

1. There exists a regular expression for every finite language.
2. A set of strings can be regular even if some of its subsets are not regular.
3. There can be more than one grammar that corresponds to a deterministic finite automaton.
4. The language of every context-free grammar is a context-free language.
5. The language of an unrestricted grammar can be regular.
6. Two separate pushdown automata with one stack each are together equivalent to a single pushdown automaton with two stacks.
7. There are languages for which there are no Turing machine deciders.
8. Every formal language has a Turing machine acceptor.
9. Every formal language has an unrestricted grammar even if it does not have a Turing machine acceptor.
10. The alphabet of a formal language can be infinite.
11. A grammar can have more terminals than non-terminals.
12. A grammar can have more non-terminals than terminals.
13. Every language over a unary alphabet (i.e., with just one symbol) is regular.
14. There are only a finite number of languages over a unary alphabet.
15. Every subset of a regular language is accepted by a deterministic finite automaton.
16. Every language accepted by a pushdown automaton has an unambiguous grammar.
17. Every context-free grammar can be converted to a left-linear grammar.
18. There is a regular language that has an ambiguous grammar.
19. There is a regular language for which there is no pushdown automaton.
20. There is a context-free language that is not deterministic.
21. There is a regular language for which there is no deterministic pushdown automaton.
22. Every grammar for every context-free language not containing λ can be converted to Chomsky Normal Form.
23. There is a context-sensitive language that is not regular.
24. There are regular languages that cannot be pumped infinitely according to the Pumping Lemma for regular languages.

25. There can be a non-deterministic finite automaton with more states than an equivalent deterministic finite automaton.
26. Regular expressions can be used to specify completely the syntax of modern programming languages.
27. No instance of the Post Correspondence Problem can be solved because it is not computable.
28. Salesmen cannot travel deterministically or easily because the Traveling Salesman Problem is an NP-Hard problem.
29. Every computer program that accepts an infinite variety of inputs must have at least one loop in it.
30. The language ww is decidable.
31. The language $a^n b^n c^n d^n e^n f^n$, where $n \geq 0$ is decidable.
32. The language L_{Diag} is recursive.
33. We can construct an enumeration procedure for the language $L_{\text{Non-RE}}$ but not a Turing machine.
34. Given any two computer programs, there is a procedure to determine whether they compute the same thing.
35. Given any computer program, there is a procedure to minimize the program the same way we minimize deterministic finite automata.
36. For every right-linear grammar, there is an equivalent left-linear grammar.
37. A non-linear grammar can be normalized to a linear grammar by splitting the right-hand sides of its productions.
38. Every Greibach Normal Form grammar is linear.
39. The same grammar can be in both Chomsky and Greibach normal forms.
40. A Turing machine can simulate the computation carried out by any pushdown automaton.
41. Every string that is grammatical can be parsed in finite time.
42. Sentential forms do not shrink during a derivation using a context-free grammar.
43. There is an algorithm that takes a regular expression as input and automatically constructs a parser for the language of the regular expression.
44. There is an algorithm that takes a context-free grammar (CFG) as input and automatically constructs a parser for the language of the CFG.
45. There is an algorithm to determine if two context-free grammars are the same.
46. There is an algorithm to solve the Modified Post Correspondence Problem.
47. There is no algorithm to decide if a given deterministic finite automaton is minimal.
48. English can be used worldwide in almost any context. Therefore it is a context-free language.
49. A language has a membership function if, given any string over its alphabet, there is an algorithm to determine whether the string belongs to the language or not.
50. Post Correspondence Problem becomes decidable if there is a limit imposed on the number of concatenations.

B. Multiple Choice Questions

51. A language L_1 is a subset of another language L_2. Then, if
    (a) L_2 is regular, then L_1 is regular
    (b) L_1 is regular, then L_2 is regular
    (c) L_2 is context-free, then L_1 is context-free
    (d) L_2 is finite, then L_1 is regular
52. Consider the grammar $S \rightarrow aSb \mid \lambda$. The language of the grammar is
    (a) Regular
    (b) Ambiguous
    (c) Non-deterministic
    (d) Linear
53. L_1 is the set of all strings over $\{a, b\}$ with equal numbers of a and b. L_2 is the set of all strings over $\{a, b, c\}$ with equal numbers of a, b and c. Then,
    (a) L_1 and L_2 are both context-free.
    (b) L_1 is regular and L_2 is context-free.
    (c) Neither is context-free.
    (d) L_1 is context-free but not regular.
54. Which of the following characteristics of a modern programming language is best specified using a context-free grammar?
    (a) Valid variable names
    (b) Maximum length of an arithmetic expression
    (c) Operator precedence in arithmetic expressions
    (d) Data type conversion
55. L_1 is a language over $\{a, b\}$ and L_2 is defined as $\{w \mid w$ contains some string from L_1 as a substring$\}$. Then,
    (a) If L_1 is regular, then L_2 is not regular
    (b) If L_1 is context-free, then L_2 is not context-free
    (c) If L_1 is recursively enumerable, then L_2 is not recursively enumerable
    (d) None of the above
56. Which of the following sets is countably infinite?
    (a) The set of all prime numbers
    (b) The set of all real numbers between 0 and 100
    (c) The set of all formal languages over $\{a, b\}$
    (d) The set of all subsets of binary strings

57. Which of the following sets is uncountably infinite?

(a) The set of all 32-bit memory addresses

(b) The set of all possible computer programs

(c) The set of all formal languages over the binary alphabet

(d) The set of all regular languages over the binary alphabet

58. Which of the following is not uncountably infinite?

(a) The set of all computer programs that can ever be written

(b) The set of all functions

(c) The set of all formal languages over the ASCII alphabet

(d) The cross product of the set of all computer programs that can ever be written with the set of all possible inputs to each program

59. Which of the following is a regular expression for binary strings with no consecutive 1 s?

(a) $(01 + 10)^*$

(b) $(1 + \lambda)(01 + 0)^*$

(c) $(0 + 1)^*(0 + \lambda)$

(d) $(10 + 0)^*(1 + \lambda)^*$

60. Which of the following is the language of the grammar:

$S \rightarrow bS \mid aA \mid b$

$A \rightarrow bA \mid aB$

$B \rightarrow bB \mid aS \mid a$

(a) Number of as more than three times the number of bs

(b) Number of bs more than three times the number of as

(c) Number of as is a multiple of 3

(d) Number of bs is a multiple of 3

61. The smallest finite automaton that accepts all non-negative binary numbers divisible by 3 has

(a) 2 states

(b) 3 states

(c) 4 states

(d) 5 states

62. The regular expression $0^*(10^*)^*$ is the same as

(a) $(1^*0)^*$

(b) $0 + (0 + 10)^*$

(c) $(0 + 1)^*10$

(d) None of the above

63. The deterministic finite automaton shown in Fig. A.1 accepts all binary strings in which

(a) The number of 0 s is divisible by 2

(b) The length of the string is divisible by 3

(c) There are more 1 s than 0 s

(d) The number of 1 s is divisible by 3

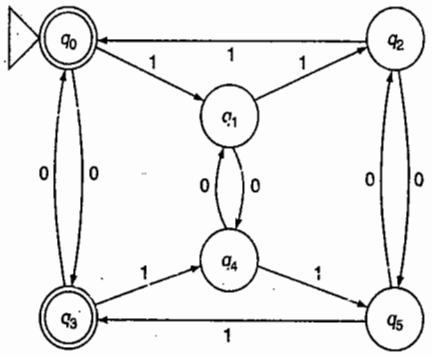


FIGURE A.1

64. Which of the following is true about the deterministic finite automaton (DFA) in Fig. A.1?

(a) It is a minimal DFA

(b) It does not accept strings not containing 1 s

(c) It accepts at least one string not containing 0 s

(d) Accepts only strings of odd length

65. Consider the deterministic finite automaton (DFA) shown in Fig. A.2 and the set of all binary strings of length 7 in which the first, fourth and the last symbols are all 1. The number of such strings accepted by the DFA is

(a) 1

(b) 5

(c) 7

(d) 8

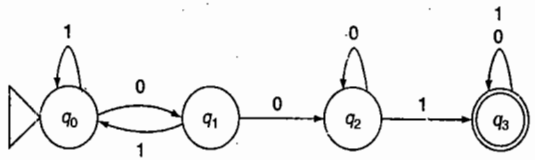


FIGURE A.2

66. Consider the Turing machine shown in Fig. A.3. This machine

(a) Is an acceptor for the language (0 + 1)1*

(b) Halts and rejects 10

(c) Does not halt on 011

(d) Is a decider for the language (0 + 1)1*

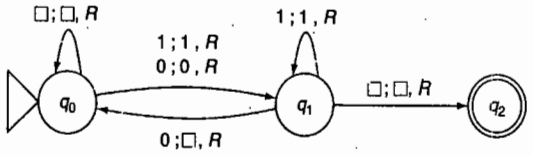


FIGURE A.3

67. Consider the non-deterministic finite automaton (NFA) shown in Fig. A.4. Let its language be L_1. Let L_2 be the language of the NFA obtained by changing the accepting state of the given NFA to a non-accepting state and the two non-accepting states to acccpting states. Which of the following is true?

(a) $L_2 = (0 + 1)^* - L_1$

(b) $L_2 = (0 + 1)^*$

(c) $L_2 = L_1$.Complement

(d) $L_2 = L_1$

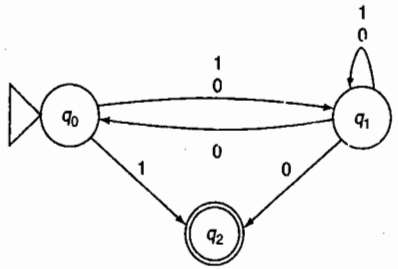


FIGURE A.4

68. Which of the following binary languages is regular?

(a) ww, where w is any binary string

(b) ww^R, where w is any binary string

(c) 0^{2i}, where i is a positive integer

(d) ww, where w belongs to (11*0)*

69. The language $a^m b^n c^{n+m}$, $m \geq 1$, $n \geq 1$ is

(a) Regular but not context-free

(b) Type II but not Type III language

(c) Type III language

(d) Type 0 but not Type II

70. A finite state machine with an output defined for each state

(a) Is a Mealy machine

(b) Accepts the closure of the alphabet

(c) Is a Moore machine

(d) Is nondeterministic

C. Fill in the Blanks

71. Simple grammars are a _____ of Greibach Normal Form grammars.

72. The set of Turing machine deciders is a _____ of the set of Turing machines acceptors.

73. The number of pushdown automata is _____ infinite.

74. The right-hand side of a production in a context-sensitive grammar is _____ the left-hand side.

75. The left-hand side of a production in a context-sensitive grammar cannot be _____.

76. In minimizing a deterministic finite automaton, a non-final state is always _____ from a final state.

77. Useless variables are either _____ or _____.

78. The correct order to clean up a grammar is: *lambda* _____ *are useless.*

79. A non-terminal that derives the null string λ is called _____.

80. Recursively enumerable languages whose complements are also recursively enumerable are called _____.

81. The languages of linear-bounded automata are _____.

82. A pushdown automaton that never uses its stack is a _____ automaton.

83. If only the rightmost symbol on the right-hand side of every production is a variable, the grammar is _____-linear.

84. A grammar that produces two or more rightmost derivations for the same string is _____.

85. Eliminating null productions can result in _____ productions.

86. Eliminating unit productions can result in _____ variables.

87. A set that cannot be mapped one-to-one to the set of natural numbers is _____ infinite.

88. Strings obtained by concatenating a string with its own reverse are called _____.

89. Any _____ property of recursively enumerable languages is undecidable.

90. The Pumping Lemma is not applicable to strings shorter than _____.

91. Every NP-Complete problem is _____ but not every _____ problem is NP-Complete.

92. If problem P_1 is known to be in the complexity class NP and P_1 can be reduced to problem P_2 in polynomial time, then P_2 is in complexity class _____.

93. Computing machines are very good at _____ but not good at _____.
94. The sequence of terminal symbols on the leaf nodes of a parse tree from left-to-right is called the _____ of the tree.
95. Grammars in _____ Normal Form give us _____ parse trees.
96. The leftmost symbol on the right-hand side of any production in _____ Form is not a variable.
97. Every multi-track Turing machine is equivalent to a _____ machine.
98. In _____ grammars, there can be only one production for every combination of a _____ on the left-hand side of the production and the _____ symbol at the beginning of its right-hand side.
99. The longest run in the string 1000111101001 is of length _____.
100. Ackermann function is _____-recursive but not _____-recursive.
101. According to the Pigeonhole Principle, $n + 1$ pigeons cannot fit into _____ holes.
102. Every infinite language must have strings that are _____ long.

Answer Key

A. True or False

1. True
2. True
3. True
4. True
5. True
6. False
7. True
8. False
9. False
10. False
11. True
12. True
13. False
14. False
15. False
16. False
17. False
18. True
19. False
20. True
21. False
22. True
23. True
24. True
25. True
26. False
27. False
28. False
29. True
30. True
31. True
32. False
33. False
34. False
35. False
36. True
37. False
38. False
39. True
40. True
41. False
42. True
43. True
44. True
45. False
46. False
47. False
48. False
49. True
50. True

B. Multiple Choice Questions

51. (d)
52. (d)
53. (d)
54. (c)
55. (d)
56. (a)
57. (c)
58. (a)
59. (b)
60. (c)
61. (b)
62. (d)
63. (d)
64. (c)
65. (c)
66. (a)
67. (b)
68. (c)
69. (b)
70. (c)

C. Fill in the Blanks

71. proper subset
72. proper subset
73. countably
74. at least as long as
75. null or λ or all terminals
76. distinguishable
77. non-generating, unreachable
78. units
79. nullable
80. recursive
81. context-sensitive languages
82. (non-deterministic) finite
83. right
84. ambiguous
85. unit
86. useless
87. uncountably
88. even palindromes
89. non-trivial
90. a constant m
91. NP-Hard, NP-Hard
92. NP
93. recall, recognition
94. yield
95. Chomsky, binary
96. Greibach Normal
97. standard or single-track
98. simple or s-, non-terminal or variable, terminal
99. 4
100. μ, primitive
101. n
102. infinitely

Theorems

The science of computing is primarily about classes of computing machines, formal languages and grammars. Throughout the chapters in this book, we have studied various properties and equivalences of the classes of automata, languages and grammars. We studied them through a problem-solving approach rather than a theorem-proving approach. We understood the key ideas conceptually, analogically and through a series of examples and exercises. However, significant statements about the classes are traditionally presented as formal theorems. Let us consolidate what we learned in the 12 chapters of this book by listing the most important theorems in the Theory of Computation. Since we have already understood the main concepts in the subject, we are better equipped to understand the theorems and appreciate their proofs, making effective use of concepts across chapters. This is especially useful in the case of theorems that establish the subset relations among classes of formal languages in the Chomsky hierarchy since the properties of the different classes of formal languages were studied in several different chapters. For example, ideas from Chapters 7, 8, 10 and 11 are needed together to establish the theorems about context-sensitive languages.

We will also discuss how the theorems can be proved. There are two reasons for proving theorems. One is to clearly understand why the statement is true; this makes our understanding of automata, languages and grammars more precise. Second, we need to acquire theorem-proving skills so that, given any new statement, we are able to figure out whether it is true by proving or disproving it without a doubt.

Methods of Proof

There are several ways to prove a positive statement, that is, a statement that is in fact true:

1. **Proof by Deduction (or Logical Inference):** In this method, we make logical deductions from a known set of axioms to show that the statement is valid. For example, to prove that *even numbers greater than 2 are not prime numbers*, we start with the definitions of even numbers and prime numbers. Even numbers are those that are divisible by 2. A prime number is one that is divisible only by itself and by 1. We know that every number (other than 0) is divisible by itself and by 1. Even numbers greater than 2 are divisible by themselves, by 1 and by 2. Therefore they are not prime numbers.

   Although deductive reasoning is the mainstay of formal logic, we cannot always use deductive proofs in this subject. In fact, other methods are more common.

2. **Proof by Construction:** In a proof by construction, we show the validity of a statement by providing a method, procedure or algorithm for constructing an automaton, grammar or other artifact. We show that the method of construction works for all possible cases and thereby argue that the statement is valid.

   For example, to show that *there is an NFA for every RegEx*, we provide a method for constructing an NFA given a RegEx, no matter how trivial or complex the RegEx is (see Theorem 3). Usually,

in such a proof, we also need to show that the artifact that is constructed is correct. In this example, we have to show that the constructed NFA is equivalent to the RegEx, that is, it accepts the same language as the set of strings that match the RegEx. We do this by following a proof by construction with another method of proof such as a proof by mathematical induction.

3. **Proof by (Mathematical) Induction:** A proof by induction is similar to a recursive definition. We show that the given statement is true for a larger value (such as $n + 1$) if it is true for a smaller value (such as n). We begin first by showing that the statement is true for a simple case (such as $n = 1$), so that there is a way to exit the recursive proof:

   the statement is true for $n + 1$
   because it is true for n
   because it is true for $n - 1$
   ...
   because it is true for 2
   because it is true for 1.

   We use the term *inductive basis* for the part which shows that the statement is true for a small value. *Inductive hypothesis* refers to the assumption that the statement is true for some value n and *inductive step* is the part of the proof that shows that the statement is true for $n + 1$ if the inductive hypothesis is true. For example, to show that *all numbers that are one less than an even power of 2 are divisible by 3*, that is, $2^{2n} - 1$, $n \geq 0$ is divisible by 3 (e.g., 0, 3, 15, 63, ...), we reason as follows:

   Inductive basis: For $n = 0$, $2^{2n} - 1 = 1 - 1 = 0$ which is divisible by 3.

   Inductive hypothesis: Let us assume that $2^{2n} - 1$ is divisible by 3. We normally do not write this step explicitly.

   Inductive step: Consider

$$\begin{aligned} 2^{2(n+1)} - 1 &= 2^{2n+2} - 1 \\ &= 2^2 \times 2^{2n} - 1 \\ &= 2^2 \times 2^{2n} - 2^2 \times 1 + 3 \\ &= 2^2 (2^{2n} - 1) + 3 \\ &= 4 \times (2^{2n} - 1) + 3 \end{aligned}$$

   This number is divisible by 3 since, according to the inductive hypothesis, $2^{2n} - 1$ is divisible by 3. Thus the statement is true for $n + 1$. Therefore, the statement is true for all values of $n \geq 0$.

   Strong Induction: There is a variation of induction where the inductive hypothesis is required to assume that the statement is true for all values less than or equal to n, not just the value n. The rest of the proof is similar to a regular induction. We will see examples of this when we prove some of the theorems.

4. **Proof by Enumeration (or Set Equivalence):** An interesting method using an enumeration or mapping can be used to prove statements about the equivalence or correspondence of two sets. To show that the two sets contain the same number of elements, we establish a one-to-one and onto function (i.e., a bijection) between the two sets. An example of such a proof was shown in Section 11.4 in Chapter 11 (see Fig. 11.2).

   Other specialized methods of proof may be applicable in particular domains. For example, a *combinatorial proof* is nothing but a deductive proof using the principles of counting in the study of permutations and combinations.

In some cases, the *contra-positive* of a given statement is easier to prove. When the given statement is of the form

If condition *then* statement

it can be rewritten using the rules of logical reasoning as

if not statement *then not* condition

For example, to show that

If a DFA has a loop in it *then* its language is infinite

(where the loop is along a path from the start state to one of its final states), we may prove its contra-positive

If the language of a DFA is finite *then* the DFA does not have a loop

It is important to remember that a positive statement cannot be proven by just giving one or any number of examples. It can only be proved by showing without a doubt that it is true for every possible value of its parameters.

Negative statements, that is, those that are in fact false, can often be proved more readily by the following methods:

5. **Proof by Contradiction:** In a proof by contradiction, we assume that the opposite of the statement is true. Based on the assumption, we use one of the other methods to show that it results in a contradiction. From this we conclude that our assumption was incorrect. Hence the opposite of what we assumed, that is, the given statement must be true.

   For example, to show that *there is no prime number greater than 2 that is even*, let us assume that the negation of the statement is true, that is, there exist even prime numbers greater than 2. Let n be such a prime number. Since n is even, it must be the case that $n = 2m$. As such, n is divisible by 2, by m and by 1. Therefore, it is not a prime number. This is a contradiction. Our assumption that there exist even prime numbers greater than 2 must be wrong. Hence the given statement is true.

6. **Proof by Counterexample:** To show that a negative statement is true, that is, to show that a particular statement is false, we simply need to give one counterexample. This is one of the easiest methods of proof. For example, if we are asked to prove or disprove the statement that *no number that is one less than an even power of 2 is divisible by 3*, we simply need to give a counterexample to show that the statement is false. Consider the number

$$15 = 16 - 1 = 2^4 - 1 = 2^{2\times 2} - 1$$

   which is divisible by 3. Thus, the given statement is false. There do exist such numbers. Similarly, if the statement is *all numbers that are one less than a power of 2 are divisible by 3*, we can give as a counterexample

$$7 = 8 - 1 = 2^3 - 1$$

   which is not divisible by 3.

A final point about proving theorems that must be remembered is the frequent requirement to prove a statement in both directions. If the given statement is of the form *X if and only if Y* (often abbreviated as *X iff Y*), we must separately prove *if X then Y* and *if Y then X*. Many of the equivalence statements that we encounter in this subject are of this type and need two parts in their proofs. Of course, the second proof is often the same as the first one so that we can omit its details. It is also important to not miss any case (e.g., $x < 0$) in any of the methods of proof.

We now consider a set of 36 theorems central to the Theory of Computation and prove them wherever necessary.

Theorem 1 ***Equivalence of NFA and DFA*:** For every non-deterministic finite automaton M_n, there is an equivalent deterministic finite automaton M_d whose language is the same as that of the non-deterministic automaton, that is, M_n.language = M_d.language.

Proof (by construction and strong induction): Consider an NFA M_n with its components M_n.alphabet (also denoted by Σ), M_n.states (also denoted by Q), M_n.startState (usually denoted by q_0), M_n.finalStates (also denoted by Q_F) and M_n.transitionFunction (a function δ from $Q \times \Sigma$ to 2^Q), that is, a mapping from the current state and the current input symbol to a set of new states.

We learned the method for constructing the DFA M_d from the NFA M_n in Chapter 3 (see Sections 3.3 and 3.4):

1. M_d.alphabet = M_n.alphabet
2. M_d.startState = $\{q_0\}$
   That is, the start state of the DFA is the set containing the start state of the NFA.
3. M_d.states = {A such that $A \subseteq M_n$.states}
   That is, each state of the DFA is a set of states of the NFA (as defined by the transition function).
4. M_d.finalStates = {A such that $A \in M_d$.states and at least one $q_1 \in A$, $q_1 \in Mn$.finalStates}
   That is, final states of the DFA are those that contain at least one final state of the NFA.
5. M_d.transitionFunction = {$(A, s) \rightarrow B$ such that $A, B \in M_d$.states, $s \in M_d$.alphabet, $q_1 \in A$, $C \subseteq B$, $(q_i, s) \rightarrow C \in M_n$.transition Function}

That is, a transition in the DFA corresponds to transitions in the NFA: for every NFA state q_i that belongs to the DFA state A, all the states C reachable from the state in the NFA for the given symbol s (including those that can be reached through one or more λ-transitions) are included in the DFA state B.

This construction of the DFA always terminates because the NFA has a finite number of states Q. States in the DFA are a subset of the power set of this finite set 2^Q, which is also a finite set.

How do we know that the resulting DFA is equivalent to the NFA? That is, when does the DFA reach a final state?

We will now show using a proof by strong induction on the length of the input string w that the DFA reaches one of its final states exactly when the NFA reaches one of its final states.

Basis for induction: Consider input strings of length one – $|w| = 1$. If the NFA reaches a final state for such a string, it contains a transition of the form

$$(q_0, w) \rightarrow q_j \text{ where } q_j \in M_n\text{.finalStates (or through one or more } \lambda\text{-transitions)}$$

By step 5 of the above construction, the DFA must contain a transition of the form

$$(\{q_0\}, w) \rightarrow B \text{ such that } q_j \in B \text{ and } B \in M_d\text{.finalStates}$$

Therefore the DFA also accepts w.

Inductive step: Let us assume that the equivalence is true for all input strings w whose length $|w| \leq n$. Consider now an input string w' of length $|w'| = n + 1$. Let $w' = w.s$ where $s \in M_n$.alphabet. Since w is of length n, the equivalence is true for w, that is, the set of states A reached by the NFA after processing w is the same as the state A reached by the DFA.

Let B be the set of states reachable by the NFA from any of the states in A after processing the symbol s (including any λ-transitions before or after processing the symbol). If one of these states is a final state of the NFA, then the input w' is accepted; otherwise it is rejected.

By construction, the DFA should also be in state B after processing the symbol s in state A. Therefore, the DFA will also accept the input w' if B is one of its final states, that is, if B contains a final state of the NFA (and reject otherwise). Thus, the DFA reaches a final state and accepts the input of length $n + 1$ exactly when the NFA reaches its corresponding final state.

By induction, the equivalence must be true for any value of n, that is, for input strings of any length. Thus, whenever the NFA accepts a string, the DFA also accepts it. This proves that the language of the NFA is a subset of the language of the DFA:

$$M_n\text{.language} \subseteq M_d\text{.language}$$

To complete the proof, we must in fact show that whenever the DFA reaches one of its final states to accept an input string, the NFA would also do so. This part of the proof is more or less the same as the above proof. The equivalence is a direct consequence of the way the DFA is constructed from the NFA and the construction guarantees the equivalence in both directions. Thus, we can also claim that

$$M_d\text{.language} \subseteq M_n\text{.language}$$

Therefore, the languages of the two machines are in fact the same and the two automata are equivalent:

$$M_n\text{.language} = M_d\text{.language}$$

Theorem 2 ***DFA minimization*:** The minimal DFA obtained by marking distinguishable states and collapsing indistinguishable states is equivalent to the original DFA (i.e., it accepts the same language).

Proof (by construction and strong induction): See Section 3.5 in Chapter 3 for the construction, that is, the method for minimizing a DFA. We once again use a strong induction on the length of the input string.

Basis for induction: Consider input strings of length one – $|w| = 1$. If the original DFA reaches a state q_i for such a string, that state being the only state reachable by the DFA after processing w (since the DFA is *deterministic*), two possibilities arise:

1. q_i is present in the minimal DFA as well. The minimal DFA also reaches the same state and accepts the input if q_i is a final state. If the DFA reached a non-final state, the minimal DFA also

reaches the same state and rejects the input. This is ensured in the construction by always distinguishing a final state from a non-final state (by the *fnf* marker, as shown in Tables 3.3 and 3.4 in Chapter 3).

2. q_i is not present in the minimal DFA; it has been collapsed with another state q_j. For example, the state q_3 would be collapsed into state q_0 in the DFA shown in Fig. A1.1 (in Appendix A) when that DFA is minimized. This is possible only when both q_i and q_j are final states or both are non-final states. Thus, even in this case, the minimal DFA accepts the input of length 1 if and only if the original DFA accepts it.

Thus, the equivalence is true for strings of length 1.

Inductive step: Let us assume that the equivalence is true for all input strings w whose length $|w| \leq n$. Consider now an input string w' of length $|w'| = n + 1$. Let $w' = w.s$ where $s \in M$.alphabet. Since w is of length n, the equivalence is true for w, that is, the state reached by the original DFA after processing w is the same as or is indistinguishable from the state reached by the minimized DFA.

Let the original DFA reach its state q_j from state q_i after processing s. If q_i and q_j are both present in the minimal DFA, then there is no issue. If q_j is a final state, both DFAs accept w'; otherwise, both of them reject the input.

Case I: q_i is present in the minimized DFA but q_j is not present: It must be the case that state q_j was collapsed into some other state q_k. From the construction, we know that q_j and q_k would be merged if and only if the two states were indistinguishable. In particular, either both are accepting (i.e., final) states or both are non-accepting states. If the original DFA accepts the input w' in state q_j, the minimal one will accept w' in state q_k; if the original DFA rejects the input in q_j, the minimal DFA also rejects it in state q_k. Thus, the sets of strings accepted by the original and minimized DFAs are the same.

Case II: q_i itself is not present in the minimized DFA: It must be the case that state q_i was collapsed into some other state q_l. From the construction, we know that q_i and q_l would be merged if and only if the two states were indistinguishable. That is, for any input symbol, including s, they should take the automaton to a pair of indistinguishable states q_x and q_y. In particular, q_x and q_y should both be final states or both non-final states. In one case both the original and minimized DFA accept w' and in the other, they both reject the input. Thus, the sets of strings accepted by the original and minimized DFAs are the same.

One more question remains to be answered. How can we be sure that the DFA obtained by the construction is indeed the most minimal DFA for the same language? A straightforward proof by contradiction suffices to show this. Let there be an even more minimal DFA that is equivalent to the original DFA. This must have at least one state lesser than the minimized DFA. Let the state missing from the new DFA be q_i. According to the construction used in minimizing the original DFA, state q_i was distinguishable from every other state in the minimized DFA. That is, there is at least one symbol for which they take the automaton to distinguishable states. Otherwise, it would have been already merged with some other state. Thus, if the state is removed, the DFA would no longer accept the same set of strings. Such a state cannot exist and the minimized DFA is the most minimal one.

For a related result, see also, Theorem 10: Myhill–Nerode Theorem.

Theorem 3 ***Equivalence of NFA and RegEx*:** For every regular expression there is an equivalent NFA whose language is the same as the set of strings that match the regular expression.

Idea of Proof: The method for constructing an NFA for a given RegEx was presented in Section 4.5 in Chapter 4. In particular, the mappings from primitive expressions to pieces of equivalent NFAs were shown in Table 4.1. It is easy to see that the corresponding NFAs accept the same sets of strings as the primitive regular expressions.

A formal proof of the equivalence is similar to the proofs given for the previous two theorems in using strong induction on top of the construction. However, instead of doing the induction on the length of the input string, this proof would have to carry out the induction on the number of operators in the RegEx. Table 4.1 shows the equivalences of all the RegEx's with zero or one operator thereby constituting the basis for the induction.

The inductive hypothesis assumes that a RegEx R with n operators is equivalent to the NFA constructed from it. The inductive step considers a new RegEx R' obtained by adding one more operator to R. This can be of the form: $R + r$, $r + R$, $R.r$, $r.R$ or R^*, where r is any RegEx with at most n operators. It is clear from the construction that in each case the corresponding NFA would accept the same set of strings. A "+" operator maps to parallel branches in the NFA; a "." to a series; and a "*" to a loop. Thus, the equivalence is established for RegEx's of any "length."

To be complete, the proof would also have to show that the NFA does not accept any other string. That is, the second part of the proof would have to show that any string accepted by the NFA also matches the RegEx. This again is obvious from the construction, as shown in Table 4.1.

The method of induction outlined above is sometimes called *structural induction* to indicate that the induction is not on a number or the length of a string but on the "length" or complexity of a structure that is built incrementally. In our present example, pieces of the NFA are put together in prescribed ways for each operator present in the RegEx. In this example, and in most others, the important ideas are in the construction or the mapping between the two classes whose equivalence is being proved. The rest of the proof is a straightforward application of mathematical induction.

Theorem 4 ***Equivalence of RegEx and Regular Language*:** For every regular language there is a regular expression that matches exactly the set of strings in the language.

Idea of Proof: A regular language is the language accepted by a DFA. From Theorem 1 we know that DFAs and NFAs are equivalent. Therefore, we can start with the languages accepted by NFAs. The construction for converting an NFA to a RegEx by eliminating states (i.e., constructing flyovers) in the NFA to obtain a generalized transition graph was illustrated in Chapter 4, Section 4.6. In particular, Table 4.2 showed the correspondence between configurations of the states in the NFA and the resulting regular expression. Figures 4.5 and 4.6 showed a general configuration for state elimination and the resulting non-trivial RegEx.

This construction is the idea behind a proof for the equivalence. As before, a complete proof would have to make a formal inductive argument on the number of states in the NFA while showing that each step in the state-elimination method results in an equivalent NFA. This is left as an exercise for interested readers.

Theorem 5 ***Equivalence of Right-Linear Grammar and NFA*:** For every right-linear grammar, there is an equivalent NFA that accepts the same language as the language of the grammar.

Proof (by induction): We need to show that the set of strings derived by the grammar is the same as the language of the NFA. We saw in Chapter 5, Section 5.6, in particular in Table 5.1, the correspondence between production rules in the grammar and transitions in the NFA. This is our construction (in which the state that corresponds to the start variable S is marked as the start state of the NFA).

Consider a derivation of a string w in a right-linear grammar. We show that any such derivation corresponds to a chain of transitions from the start state to a final state in the NFA where the labels on the successive transitions together constitute the string w.

Basis for induction: Consider input strings of length one – $|w| = 1$. They can be derived in two ways:

1. $S \Rightarrow w$
2. $S \Rightarrow V_1 \Rightarrow V_2 \Rightarrow \ldots \Rightarrow V_i \Rightarrow w$

In the first case, the start state of the NFA has a transition to a final state labeled w. Hence the NFA accepts the input w of length one. In the second case, the NFA has a chain of λ-transitions from the start state leading to a transition with the label w that takes it to a final state. Here again, the NFA accepts the input.

Inductive step: Let us assume that the equivalence is true for all input strings w whose length $|w| \leq n$. Consider now an input string w' of length $|w'| = n + 1$. Let $w' = w.s$, where s is a terminal symbol in the grammar. A successful derivation of w' in a right-linear grammar must be of the form:

$$S \Rightarrow w_1V_1 \Rightarrow w_1w_2V_2 \Rightarrow w_1w_2w_3V_3 \Rightarrow \ldots \Rightarrow w_1w_2w_3 \ldots w_nV_n \Rightarrow w_1w_2w_3 \ldots w_ns$$

where each w_i is the ith symbol in the input and V_i is any variable in the grammar. Optionally, there can be unit productions of the form $V_i \rightarrow V_j$ anywhere in the derivation. There can also be productions with more than one terminal symbol that could reduce the number of steps in the derivation. For example, if there is a production $V_i \rightarrow w_jw_{j+1}w_{j+2}V_{i+1}$, three terminal symbols are generated in a single application of this production. The NFA needs to consume the three input symbols and will need two intermediate states to do this. We can easily introduce such intermediate states by rewriting the above production as three different productions by introducing intermediate variables V_x and V_y:

$$V_i \rightarrow w_jV_x \quad V_x \rightarrow w_{j+1}V_y \quad V_y \rightarrow w_{j+2}V_{i+1}$$

In other words, we rewrite the right-linear grammar in Greibach Normal Form (for regular grammars!). In any case, there must be a production in the grammar of the form $V_n \rightarrow s$. From the inductive hypothesis and the construction, we know that the NFA will be in the state q_n corresponding to V_n (among other states, non-deterministically) after processing w. The production for V_n corresponds to a transition in the NFA from q_n to a final state with the input symbol s consumed in the transition (optionally through one or more λ-transitions that correspond to the application of unit productions in the derivation). Thus, the NFA accepts the same string w' that the grammar derived. This is true for strings of any length.

As usual, the second part of the proof must show that any string accepted by the NFA so constructed can also be derived by the grammar. The method for constructing a right-linear grammar from an NFA is simply the inverse of the above method for converting a grammar to an NFA (see Table 5.1). The rest of this part of the proof is very similar to the first part and is omitted.

Theorem 6 ***Equivalence of Right- and Left-Linear Grammars*:** For every right-linear grammar, there is an equivalent left-linear grammar whose language is the same as that of the right-linear grammar.

Idea of Proof: Consider a right-linear grammar (rewritten in Greibach Normal Form for regular grammars, as shown above):

$$S_R \rightarrow w_1V_{R1}$$
$$V_{R1} \rightarrow w_2V_{R2}$$
$$V_{R2} \rightarrow w_3V_{R3}$$
$$\ldots$$
$$V_{Rn-1} \rightarrow w_{n-1}V_{Rn-1}$$
$$V_{Rn-1} \rightarrow w_n$$

A derivation of a string w of length $|w| = n$ in such a right-linear grammar must be of the form

$$S_R \Rightarrow w_1V_{R1} \Rightarrow w_1w_2V_{R2} \Rightarrow w_1w_2w_3V_{R3} \Rightarrow \ldots \Rightarrow w_1w_2w_3 \ldots w_{n-1}V_{Rn-1} \Rightarrow w_1w_2w_3 \ldots w_n$$

Consider the corresponding left-linear grammar:

$$S_L \rightarrow V_{Ln-1}w_n$$
$$V_{Ln-1} \rightarrow V_{Ln-2}w_{n-1}$$
$$\ldots$$
$$V_{L3} \rightarrow V_{L2}w_3$$
$$V_{L2} \rightarrow V_{L1}w_2$$
$$V_{L1} \rightarrow w_1$$

with a corresponding derivation:

$$S_L \Rightarrow V_{Ln-1}w_n \Rightarrow V_{Ln-2}w_{n-1}w_n \Rightarrow V_{Ln-3}w_{n-2}w_{n-1}w_n \Rightarrow \ldots \Rightarrow V_{L1}w_2w_3 \ldots w_{n-1}w_n \Rightarrow w_1w_2w_3 \ldots w_n$$

The two derivations for the string w are equivalent and hence the corresponding grammars are also equivalent. A complete and formal proof of this theorem would make use of a DFA constructed from the right-linear grammar. The DFA is first reversed (as shown in Section 6.1.5 in Chapter 6), a right-linear grammar constructed from the reversed DFA and finally the productions of the grammar are reversed to obtain a left-linear grammar. The equivalence of their languages is argued once again by appealing to mathematical induction on the length of the input string.

Theorem 7 ***Closure Properties of Regular Languages*:** Regular languages are closed under union, concatenation, * closure, complementation, reversal, intersection, set-difference and homomorphism.

Proof (by construction): These statements were shown to be true in Chapter 6, Section 6.1. Each of them was a proof by construction along with a simple deductive argument.

Theorem 8 ***Decidability of Questions about Regular Languages*:** Algorithms exist for determining whether a given regular language is empty, finite or infinite; whether a string belongs to a regular language; and whether two regular languages are the same.

Proof: Algorithms (or methods) for answering each of these questions were presented in Chapter 6, Section 6.2.

Theorem 9 ***Pumping Lemma for Regular Languages*:** Every infinite regular language L has a constant m, specific to that language, such that all strings w, $|w| \geq m$, belonging to L can be split into $w = xyz$, where $|xy| \leq m$ *and* $|y| \geq 1$ and for all $i = 0, 1, 2, \ldots$ the strings xy^iz belong to L.

Proof: The Pumping Lemma was shown to be true in Chapter 6 (see Section 6.5 and, in particular, Fig. 6.4). The essence of the proof is the Pigeonhole Principle (see Section 6.4). There aren't enough states in a DFA to handle very long strings from an infinite regular language without going through a loop. Once a loop exists, it can be traversed any number of times, thereby repeating the pattern of symbols along the loop any number of times to generate infinitely many strings. All such strings must belong to the language. Similarly, a RegEx for an infinite language must have at least one * closure and a regular grammar for the language must to have at least one cyclic (or recursive) production that brings back the same variable into the sentential form.

Theorem 10 ***Myhill–Nerode Theorem*:** A language L is regular if and only if the equivalence relation of distinguishing extensions on strings in the language has a finite number of equivalence classes.

Proof (by construction and contradiction): Let us define the equivalence relation R for a language L for a pair of strings over its alphabet as follows: Two strings w_1 and w_2 are *distinguishable* if there is a string x such that exactly one of $w_1.x$ and $w_2.x$ belongs to L. Two strings are *indistinguishable* if there is no extension x that distinguishes them. They are also called *left-equivalent* strings.

First, let us prove that if a language is regular then the number of equivalence classes for it is finite. If a language is regular, there exists a DFA that accepts the language. Since the DFA is a decider for the regular language (i.e., it computes the membership function for the language), and it is deterministic, any input string over the alphabet takes the DFA to exactly one state. For each state in the DFA, consider the set of input strings for which the DFA reaches the state from the start state. Since there are only a finite number of such states in the DFA, we get a finite number of such sets of strings. Thus, all the strings over the alphabet can be partitioned into a finite number of sets each corresponding to a particular state in the DFA.

Next, we need to show that strings in the same partition are *indistinguishable*. Let us try a proof by contradiction. Consider any two strings w_1 and w_2 in the same partition, that is, from the same set that corresponds to a particular state of the DFA. Let them be distinguishable. Then there exists a string x such that only one of $w_1.x$ and $w_2.x$ belongs to the language. Only one of them is accepted by the DFA. But this is impossible in a DFA. It does not matter whether the DFA got to the present state by consuming w_1 or w_2 from its input. From the present state, if x takes it to a final state, then both $w_1.x$ and $w_2.x$ will be accepted. If, on the other hand, x takes the DFA to a non-final state, then both $w_1.x$ and $w_1.x$ will be rejected. Therefore w_1 and w_2 are not distinguishable. This is a contradiction. Therefore, our assumption that they are distinguishable is incorrect. Thus, there are only a finite number of equivalence classes of strings that correspond to the set of states in the DFA.

In the reverse direction, we need to show that if the number of equivalence classes is finite, then the language is regular. Using an argument similar to the first part of the proof, we construct a DFA in which each state corresponds to one of the equivalence classes. This is possible since any two strings belonging to the same equivalence class are such that the same string x either takes them both to acceptance or both to rejection. We omit the details of this second part of the proof.

It may be noted that the Myhill–Nerode theorem is the basis for minimizing a DFA. The number of states in a minimal DFA that accepts L is equal to the number of equivalence classes in the relation R.

Theorem 11 ***Substitution Rule for Grammars*:** If a grammar has a set of productions

$$A \rightarrow uBv \text{ and } B \rightarrow x \mid y \mid z$$

where u, v, x, y, z stand for any piece of a sentential form with terminal and non-terminal symbols, then the grammar obtained by eliminating the B productions by substituting the right-hand sides of those productions wherever B occurs in other productions is equivalent to the original grammar:

$$A \rightarrow uxv \mid uyv \mid uzv$$

Proof (by deduction and induction): Consider a string belonging to the language of the grammar and the sequence of sentential forms by which the start symbol S derives the string. If the production

$$A \rightarrow uBv$$

is used at least once in the derivation, the sentential form before and after applying the production once must look like *rAs* and *ruBvs*, where r and s are also any sequences of symbols. For the string to be derived successfully, the variable B must be eliminated from the sentential form by applying one of the productions for B. At some point, when B is eliminated, we must get *ruxvs* or *ruyvs* or *ruzvs*. One of these will eventually derive the given string.

In the new grammar where the production for B has been substituted into the A productions, we get directly from *rAs* the same sentential forms, either *ruxvs* or *ruyvs* or *ruzvs*, one of which will derive the given string. Thus, if the original grammar derives a string, so does the new one. Their languages are the same.

In fact, to conclude the last statement, we need a second part of the proof where we argue in the reverse direction. That is, if the new grammar derives a string, then the original one also derives it. The argument is very similar to the one above.

Why do we need induction then? We have shown what happens when the production under consideration is applied once during a derivation. The same production of course can be applied more than once to derive a string. If needed, we can apply a proof by induction on the number of times the production is applied to complete the proof.

Theorem 12 ***Equivalence of Cleaned-Up Grammars*:** The grammar obtained by eliminating λ-productions, unit productions and useless variables from a context-free grammar is equivalent to the original grammar.

Proof (by deduction): The methods for eliminating λ-productions, unit productions and useless variables were presented in Chapter 7 in Section 7.10. Let us first see why eliminating useless variables does not alter the language of the grammar. Useless variables are either unreachable or non-generating. An unreachable variable can never be derived by the start symbol S and therefore never appears in any sentential form for any string in the language. As such, removing it has no effect on the language of the grammar. A non-generating variable, whenever it occurs in a sentential form, can never be eliminated. Such a sentential form is a dead end; it can never generate any string in the language of the grammar. Hence, it too can be removed without any effect.

Removal of λ-productions and unit productions are done by applying the substitution rule. As such, we can use Theorem 11 to argue that the resulting grammar has the same language as the original grammar.

Theorem 13 ***Chomsky Normal Form*:** Every context-free grammar whose language does not contain λ can be converted to an equivalent grammar that is in Chomsky Normal Form (CNF).

Proof: Any context-free grammar can be converted to CNF by first cleaning it up and then rewriting the production rules to comply with the requirements of the CNF. The method was presented in Chapter 7, Section 7.10. The first part was Theorem 12 and the second part is merely a repeated application of the substitution rule. As such, by combining the proofs of Theorems 12 and 11, we get the proof that the resulting grammar in CNF has the same language as the original grammar.

Theorem 14 ***Greibach Normal Form*:** Every context-free grammar whose language does not contain λ can be converted to an equivalent grammar that is in Greibach Normal Form (GNF).

Proof: On similar lines, we can also show that the resulting GNF grammar also has the same language as the original grammar.

Theorem 15 ***Equivalence of Non-Deterministic PDA and CFL*:** The language of every non-deterministic pushdown automaton is a context-free language. Conversely, for every CFL, there is an equivalent NPDA that accepts the language.

Proof (by construction and induction): A proof of this theorem is a direct consequence of the constructions suggested in Chapter 8 for converting a PDA to a CFG (see Section 8.6) and for converting a CFG to a PDA (see Section 8.5). The constructions are designed to specifically ensure the equivalence of a transition in the NPDA to the application of a production in deriving a string in the CFG. To enable this, productions in the CFG must be rewritten in one of the following two forms:

$T \rightarrow a$ Pops T off the stack (shrinks by 1)
$T \rightarrow aXY$ Pops T off the stack and pushes Y first and then X (grows by 1)

As noted in Section 8.5, this is a type of grammar that is a combination of the ideas of Chomsky Normal Form and Greibach Normal Form. The first production above is in both normal forms; the second one is in GNF but has a restriction similar to CNF that there can't be more than two variables on the right-hand side.

It is clear from Theorems 13 and 14 that any CFG can be rewritten in the above form. Given such a grammar and the methods of construction, it is also clear that applications of productions in the grammar and transitions in the NPDA mirror each other. The NPDA reaches its final state to accept a string exactly when the grammar derives the string from its start symbol S. To complete the proof in either direction, we merely need to add a mathematical induction on the number of productions applied (or the number of transitions made).

Theorem 16 ***Closure Properties of CFL*:** Context-free languages are closed under union, concatenation and * closure.

Proof (by construction): These statements were shown to be true in Chapter 9, Section 9.1. Each of them was a proof by construction along with a simple deductive argument.

Corollary 16.1: CFLs are not closed under complementation, intersection or set-difference.

Proof (by counterexample): A counterexample was presented in Section 9.1 for each of complementation, intersection and set-difference.

Theorem 17 ***Pumping Lemma for CFL*:** Every infinite context-free language L has a constant m, specific to that language, such that all strings w, $|w| \geq m$, belonging to L can be split into $w = uvxyz$, where $|vxy| \leq m$ and $|vy| \geq 1$ and for all $i = 0, 1, 2, \ldots$, the strings uv^ixy^iz belong to L.

Proof: This Pumping Lemma was shown to be true in Chapter 9 (see Section 9.5 and, in particular, Fig. 9.2). The essence of the proof is the Pigeonhole Principle (see Section 6.4) because of which at least one variable must appear more than once in a sentential form that derives long strings in the language. This allows us to interchange the productions for two different appearances of the same variable, thereby generating infinitely many strings belonging to the language.

Theorem 18 ***Ogden's Lemma*:** Every infinite context-free language L has a constant m, specific to that language, such that all strings w, $|w| \geq m$, belonging to L can be marked (or distinguished) at any m or more symbols and split into $w = uvxyz$, where vxy has at most m marked symbols, vy contains at least one marked symbol and for all $i = 0, 1, 2, \ldots$, the strings uv^ixy^iz belong to L.

Proof: This is a stronger version of the Pumping Lemma (Theorem 17). Essentially, it says that we need not consider the entire string w but just the symbols at any m (or more) chosen positions in the string w. The proof itself is identical to that of the Pumping Lemma (Theorem 17) except that any unmarked symbol is ignored and only marked ones are considered when applying the pigeonhole principle to the nodes in the parse tree.

Theorem 19 ***Decidable Questions about CFL*:** Algorithms exist to determine if a given context-free language is empty, finite or infinite; an algorithm exists to determine if a given string belongs to the language of a given context-free grammar.

Proof: These were proven in Chapter 9, Section 9.2, wherein a method was provided for answering each of the questions. However, no algorithm exists for determining whether a CFG is ambiguous or whether two CFGs are equivalent (see the comments for Theorem 36).

Theorem 20 ***Equivalence of Variations of Turing Machines*:** Turing machines with stay-option, multi-track tape, semi-infinite tape, multi-tape, multi-dimensional tape, non-deterministic Turing machines are all equivalent to standard Turing machines.

Proof (by construction):

20.1 ***Stay-option Turing machine*:** A transition in a stay-option Turing machine has a third option S (stay in the same cell) apart from R (move right) and L (move left). We construct a standard Turing machine from a stay-option Turing machine as follows. For each transition of the form

$$(q_i, a) \rightarrow (q_j, b, S)$$

which would be shown as $a; b, S$ on an arrow from state q_i to state q_j in a diagram for the Turing machine, we replace it with two transitions:

$$(q_i, a) \rightarrow (q_k, b, R) \text{ and } (q_k, x) \rightarrow (q_j, x, L) \text{ for each } x \in \Sigma \text{ and } \square$$

where q_k is a new state introduced to handle the intermediate move to the right and then back to the left. The second transition above is in fact a set of transitions obtained by replacing the x successively with every symbol in the alphabet Σ.

It is easy to see that one such replacement of a transition with the stay-option does not change the set of strings accepted by the machine. The same symbol a is replaced by the same symbol b and the state is

changed eventually to the same state q_j. Moreover, since the tape of a Turing machine is infinitely long, there is always a cell to the right of the current cell to accommodate the intermediate transition to the right and then back to the left. To complete the proof, we need to apply induction on the number of such transitions replaced. The details are mechanical and are omitted (see Exercise 31 in Chapter 10).

20.2 ***Multi-track Turing machine*:** In a multi-track Turing machine, the tape has n parallel tracks so that the reading head can read from or write into each cell n symbols in a single transition. To show that such a machine is equivalent to the standard Turing machine, we need to show how a standard machine can simulate the computation of the multi-track machine. A multi-track machine can of course trivially simulate the behaviors of a standard machine by using only one of its tracks.

The idea behind the simulation is to interlace the multiple tracks into a single track and adjust the transitions so that the standard machine can perform an equivalent series of transitions. The contents of a single cell in the multi-track machine will be placed in n consecutive cells of the standard single-track tape. A transition in the multi-track machine:

$$(q_i, a[]) \rightarrow (q_j, b[], R)$$

where $a[]$ and $b[]$ are like arrays of length n that index into the n tracks in the current cell, will be replaced by a sequence of transitions:

$$(q_i, a[1]) \rightarrow (q_k, b[1], R), (q_k, a[2]) \rightarrow (q_{k+1}, b[2], R), \ldots, (q_{k+n-1}, a[n]) \rightarrow (q_j, b[n], R)$$

where $q_k \ldots q_{k+n-1}$ are new intermediate states and only the last transition goes to state q_j. The intermediate transitions serve to replace the symbols $a[i]$ by the corresponding $b[i]$ while moving right to access each successive symbol originally found on the multiple tracks. Figure B.1 shows the mapping from the multiple tracks to the single track of the standard Turing machine.

If the given transition has a left move, then the intermediate transitions will have to move right to access the symbols from the multiple tracks and, finally, we need $2n$ more intermediate transitions to move back to the beginning of the multi-track cell to the left (e.g., in Fig. B.1, the reading head has to move 4 cells to the right and then 8 cells back to the left).

As always, to complete the proof we need to add an induction on the number of such changes made to replace multi-track transitions with sequences of single-track transitions (see Exercise 27 in Chapter 10).

20.3 ***Semi-infinite tape Turing machine*:** A semi-infinite tape has a definite end on the left and is infinitely long only to the right. To show that such a machine can simulate the computation of a standard Turing machine, we can make it a two-track semi-infinite tape where the second track is used to simulate

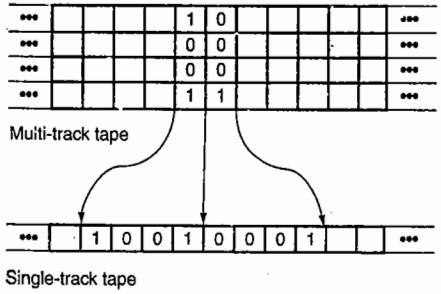


FIGURE B.1 Multi-track simulation on a single track tape.

the contents of the "left half" of the standard Turing tape. Transitions can be adjusted suitably to show that the machine is equivalent, that is, it accepts the same language as the standard Turing machine (see Exercise 30 in Chapter 10).

20.4 ***Multi-tape Turing machine*:** A multi-tape machine has more than one tape with independent reading heads for each tape. Now that we know how multi-track machines are equivalent to the standard single-track machine, we can simulate the computations of a multi-tape machine on a multi-track machine. We use as many tracks as there are tapes in the multi-tape machine and additional tracks to note down the position of each reading head as it moves along its own tape in the multi-tape machine. As before, we need to adjust the transitions suitably to show that the two machines are equivalent (see Exercise 28 in Chapter 10).

20.5 ***Multi-dimensional Turing machine*:** Similarly, a multi-dimensional tape can also be simulated on a single tape by extending the way we interlaced multiple tracks onto a single track (Fig. B.1). The reading head in a multi-dimensional tape machine can move in other dimensions in addition to left and right; it may move up, down, to the front, to the back and so on. All such moves can be accommodated by suitable adjustments to the transitions to show the equivalence (see Exercise 29 in Chapter 10).

20.6 ***Non-deterministic Turing machine*:** A non-deterministic Turing machine can have more than one transition from the same state for the same symbol in the current cell on the tape. The computations of a non-deterministic Turing machine can be simulated on a two-tape deterministic Turing machine by using one of the tapes to maintain the instantaneous descriptions (or snapshots) of all the parallel paths in which the non-deterministic machine is at a particular point in its computation. When the non-deterministic machine makes multiple transitions from its current state, the second tape is used as working memory to copy its instantaneous description as many times as needed to create that many new snapshots on the first tape. Each such snapshot is explored in turn (similar to a typical implementation of *breadth-first search*) and, in the end, if any one of them reaches a final state of the non-deterministic machine, the deterministic machine also reaches its final state to accept the input. In the worst case, the simulation may need to deal with an exponential number of snapshots to try all the combinatorial choices encountered by the nondeterministic machine. The details are quite tedious and are not included here (see Exercise 32 in Chapter 10).

By combining the above results, we can argue that even a non-deterministic, multi-dimensional, multi-track, multi-tape, semi-infinite Turing machine with stay-option is equivalent to a standard, deterministic, single-dimensional, single-track, infinitely long, single-tape Turing machine without the stay-option!

Theorem 21 ***Countability of Turing Machines*:** The set of all possible Turing machines is countably infinite.

Proof (by enumeration or set-equivalence): As shown in Chapter 10, Section 10.9, any standard Turing machine can be encoded completely in a single binary string. However, not every binary string is a valid encoding of a Turing machine. Therefore, the set of all possible Turing machines is a proper subset of the set of all binary strings. The set of all binary strings is countably infinite since it can be placed in one-to-one correspondence with the set of natural numbers by a proper enumeration procedure that lists binary strings ordered by length first and then by lexicographic order as follows:

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

With such an enumeration, we can talk about the i^{th} binary string and thus, the set is countably infinite (see also, Section 11.3 in Chapter 11).

Theorem 22 ***Uncountability of Languages*:** The set of all formal languages over a given non-empty alphabet is uncountably infinite.

Proof (by diagonalization and contradiction): A proof based on the diagonalization argument was presented in Chapter 11 (see Section 11.6 and Fig. 11.4).

Theorem 23 ***Existence of Non-RE Languages*:** There are formal languages that are not recursively enumerable; there are formal languages for which no Turing machine acceptor exists.

Proof (by diagonalization and contradiction): A proof for the existence of the language $L_{\text{Non-RE}}$ based on the diagonalization argument was presented in Chapter 11 (see Section 11.6).

Theorem 24 ***Recursive Languages are a Proper Subset of RE Languages*:** Every recursive language is recursively enumerable; there exist recursively enumerable languages that are not recursive.

Proof (by deduction): Recursive languages have Turing machine acceptors and therefore are a subset of recursively enumerable languages. However, there exist languages such as L_{Diag} that are recursively enumerable but not recursive, as shown in Chapter 11 (see Section 11.7). Thus, recursive languages are a proper subset of recursively enumerable languages.

Corollary 24.1: There are recursively enumerable languages whose complements are not recursively enumerable.

This is shown by the example of the language L_{Diag}.

Corollary 24.2: If a language and its complement are both recursively enumerable then they are both recursive.

This can be shown easily by combining the Turing machine acceptors for each language (which exist because they are both recursively enumerable) into a single Turing machine that becomes a Turing machine decider as shown in Fig. 11.5 in Chapter 11. The combined machine computes the membership function for either language thereby making them both recursive.

Theorem 25 ***Context-Sensitive Languages are a Proper Subset of Recursive Languages*:** Every context-sensitive language is recursive; there exist recursive languages that are not context-sensitive.

Proof (by deduction, diagonalization and contradiction): A context-sensitive language has a corresponding linear-bounded automaton (see Theorem 31) that computes its membership function (i.e., acts as a decider machine). Therefore, every context-sensitive language is recursive. A more formal proof can be provided by showing that the length of any derivation in a context-sensitive language is limited by a function of the length of the string being derived. The limit on the length and the resulting finiteness of the set of strings of that length are used to show that a membership function exists for any context-sensitive language, thereby making it recursive.

In the other direction, it can be shown that there exist recursive languages that are not context-sensitive. First, it is assumed that all recursive languages are context-sensitive. Then it is shown that every context-sensitive grammar (which corresponds to a context-sensitive language by definition) can be encoded as a binary string. Such binary strings are enumerated and a diagonalization argument is put forth to construct a recursive language that is not identical to any enumerated context-free language. The contradiction shows the existence of a recursive language that is not context-sensitive. The details of the proof are identical to the diagonalization argument used to show the existence of L_{Diag} (see Theorem 24 and Section 11.7).

Theorem 26 ***Context-Free Languages are a Proper Subset of Context-Sensitive Languages*:** Every CFL is context-sensitive; there exist context-sensitive languages that are not context-free.

Proof (by deduction): Every context-free grammar (which corresponds to a context-free language by definition) is also trivially a context-sensitive grammar with the additional constraint that it has just a single symbol on the left-hand side of every production rule. Thus CFLs are a subset of CSLs. To show that they are a proper subset, we need to show that there exists at least one context-sensitive language that is not context-free. An example of such a language is $a^n b^n c^n$ which is not a context-free language, as shown by the Pumping Lemma (see Example 9.1 in Chapter 9) and is a context-sensitive language, as shown by constructing a context-sensitive grammar for it (see Example 11.1 in Chapter 11). Thus, CFLs are a proper subset of CSLs.

Theorem 27 ***Deterministic CFL is a Proper Subset of CFL*:** Every deterministic CFL is context-free; there exist context-free languages that are not deterministic.

Proof (by deduction): Deterministic CFL is defined as the class of languages accepted by deterministic PDAs. Since every DPDA is trivially an NPDA, it follows (from Theorem 15) that every DCFL is also a CFL. Thus, the set of DCFLs is a subset of CFLs. To show that it is a proper subset, we need to show that exists at least one CFL that is not a DCFL.

Consider the language of even palindromes ww^R (see Example 7.3 in Chapter 7 and Example 8.3 in Chapter 8) over the alphabet $\{a, b\}$ specified by the CFG:

$$S \rightarrow aSa \mid bSb \mid \lambda$$

Consider the strings $w_1 = aaaa$ and $w_2 = aaabbaaa$ which belong to this language. A deterministic PDA cannot be constructed to accept both strings. For w_1 it has to push just two a s and start popping thereafter; for w_2 it has to push the third a also onto the stack. A deterministic PDA cannot do both for the same prefix aaa of the two input strings. Non-determinism is essential to accept both strings. This language is context-free but not deterministic. Therefore, the set of DCFLs is a proper subset of CFLs.

Theorem 28 ***Linear Languages are a Proper Subset of CFL*:** Every linear CFL is context-free; there exist context-free languages that are not linear.

Proof (by deduction): A linear language is represented by a linear grammar, that is, a grammar where the right-hand side of any production can have at most one variable. By definition, a linear grammar is a CFG and therefore, every linear language is a CFL. To show that it is a proper subset of CFLs, we need to show that there exists at least one CFL that is not linear.

Consider the language of proper nesting of parentheses (see Example 7.11 in Chapter 7 and Exercise 2 in Chapter 8) specified by the non-linear grammar:

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

It is impossible to construct a linear grammar for this language because it contains strings such as *abab* and *aababb*. Any sentential form that derives such strings must contain more than one variable at some point in the derivation (since it can also be shown using the Pumping Lemma that the language is not regular). This language is context-free but not linear. Therefore, the set of linear languages is a proper subset of context-free languages.

Theorem 29 ***Regular Languages are a Proper Subset of both DCFL and of Linear Languages*:** Every regular language is both deterministic and linear (and therefore context-free); there exist DCFLs that are not regular; there are linear languages that are not regular.

Proof (by deduction): Every regular language has a right-linear grammar (from Theorem 5) which is linear. Thus regular languages are a subset of linear languages. Every regular language also has a DFA that accepts it. Thus regular languages are also deterministic. To show that regular languages are a proper subset of linear languages, we need to show that there exists a linear language that is not regular. The language of simple nesting (see Example 7.2 in Chapter 7) specified by the grammar:

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

is such a linear language. We have already shown using the Pumping Lemma that this language is not regular (see Example 6.2 in Chapter 6). Thus, the language is linear but not regular.

Similarly, we need to show that there exists a DCFL that is not regular. The same language of simple nesting is also deterministic but not regular. Thus, regular languages are a proper subset of both linear languages and DCFLs. Remember that we have also seen that while some languages such as simple nesting are both linear and deterministic, there exist linear languages that are not deterministic and DCFLs that are not linear (see Fig. 11.11 in Chapter 11). Thus, the sets of linear languages and DCFLs overlap but are not identical.

Theorem 30 ***Finite Languages are a Proper Subset of Regular Languages*:** Every finite language is regular; there exist regular languages that are not finite.

Proof: Every finite language is regular because we can construct a RegEx for every finite language. Let the finite language L have n strings, that is,

$$L = \{w_1, w_2, w_3, \ldots, w_n\}$$

A RegEx for L is

$$w_1 + w_2 + w_3 + \ldots + w_n$$

where each w_i is actually a concatenation of all the symbols in it. Thus, by Theorem 4, the language is regular. However, not every regular language is finite. For example, a^* is regular but not finite. Therefore, the set of finite languages is a proper subset of regular languages.

Theorem 31 ***Equivalence of Linear-Bounded Automata and Context-Sensitive Languages*:** For every context-sensitive language, there is a linear-bounded automaton that accepts the language. Conversely, the language accepted by every linear-bounded automaton is a context-sensitive language.

Idea of Proof: Context-sensitive languages are defined as the languages of context-sensitive grammars. A proof of this theorem is based on the property of context-sensitive grammars, namely, that their production rules are non-shrinking. This implies that the sentential form can only increase in length during any derivation using a context-sensitive grammar (except when a null production is used to replace a variable by λ). However, because there can be unit productions or other productions whose left- and right-hand sides are of the same length, the sentential form can also remain the same length when a production is applied. Such a non-shrinking behavior of the sentential form is used to show that a limited length of the tape is sufficient to derive the string in a Turing machine. The limit on the length is a linear

function of the length of the input string (using a multi-track tape that is linear-bounded if necessary). Thus, the Turing machine can be restricted to a linear-bounded automaton.

In a similar proof, the second part of the theorem can be proved by constructing a context-sensitive grammar from the transitions of a linear-bounded automaton (LBA), thereby showing that its language is context-sensitive. It may also be noted here that linear-bounded automata are non-deterministic. It is not known whether deterministic linear-bounded automata are equivalent to non-deterministic LBAs.

Theorem 32 ***Equivalence of Unrestricted Grammars and RE*:** The language of every unrestricted grammar is a recursively enumerable language. Conversely, for every recursively enumerable language, there exists an equivalent unrestricted grammar.

Proof (by construction and deduction): A grammar provides a method for enumerating all the strings in its language. A grammar contains a finite number of productions each of which has a finite number of symbols in it. As such, the set of all strings that can be derived from the start symbol S by applying productions a given number of times is finite. The enumeration procedure for a given grammar can therefore be as follows:

```
i := 1;
repeat
      List all the strings that can be generated by applying any
      permutation of i productions;
      i := i + 1;
until no more strings are enumerated.
```

Note that the same production can be used any number of times in the above permutations. This is an enumeration procedure that enumerates all the strings in the language of the grammar in a proper order. It can be encoded as a Turing machine and therefore, by the definition of recursively enumerable languages, the language of an unrestricted grammar is recursively enumerable.

Similarly, in the other direction, it can be shown that any recursively enumerable language, that is, any language accepted by a Turing machine, is the language of an unrestricted grammar. The sequence of transitions made by the Turing machine to accept a string is mapped to the steps in the derivation of the string using the grammar. The details are rather similar to the correspondence between derivation and the Post Correspondence Problem shown in Chapter 12 (see Section 12.2, Fig. 12.1 and Table 12.1).

Theorem 33 ***Undecidability of the Halting Problem*:** The halting problem of Turing machines is undecidable.

Proof (by contradiction): A proof of this statement was provided in Chapter 12 (see Section 12.3 and Figs. 12.2 and 12.3). This can also be proved using the technique of problem reduction as outlined in Section 12.4. In essence, the proof shows that if there was no halting problem, then all recursively enumerable languages become recursive which contradicts Theorem 24. It may also be noted that undecidability of the halting problem is the reason why a universal oracle of the type shown in Fig. 12.2 does not exist.