Automata, Computability and Complexity: Theory and Applications

Elaine Rich

Automata, Computability, and Complexity

THEORY AND APPLICATIONS | Elaine Rich

Automata, Computability and Complexity illuminates the elegant theoretical underpinnings of computing and brings theory to life by demonstrating its influence on modern hardware and software system design.

Dr. Elaine Rich begins with finite state machines and regular languages, then turns to context-free languages and parsing techniques. Next, she introduces Turing machines and equivalent models of computation, as well as the crucial question of undecidability. Building on this material, she then considers theproblem of practical computability.

Throughout, the author shows how key concepts are applied, giving readers practical insight for using computing theory in their own work. Applications discussed include: programming languages, compilers, networking, natural language processing, artificial intelligence, computational biology, security, games, business rule modeling, markup languages, Web search, and more. An appendix offers a detailed refresher on the mathematical techniques associated with the principles of computing.

Coverage includes:

- Understanding computation, including decision procedures, nondeterminism, and functions on languages
- The hierarchy of language classes: regular, context-free, decidable, and semidecidable languages
- Exploring formal computation and the problems it can solve
- Discovering fundamental limitations on what can be computed
- Finite state machines, regular expressions, and regular grammars
- Context-free languages and grammars, and non-context-free languages
- Algorithms and decision procedures for regular and context-free languages
- Turing machines and undecidability: from the Church-Turing Thesis and the Halting Problem to the Chomsky Hierarchy
- Decidability and undecidability proofs
- Analyzing complexity: time and space complexity classes

About the Author

Elaine Rich, Senior Lecturer at The University of Texas at Austin, currently teaches Automata Theory, Artificial Intelligence, and other courses. She is the author of *Artificial Intelligence*, which has been honored on CACM's list of classic computer science textbooks. She holds a Ph.D. in Computer Science from Carnegie-Mellon University.

ISBN-13: 978-0-13-228806-4
ISBN-10: 0-13-228806-0

EAN

9 780132 288064 90000

PEARSON
Prentice Hall
Upper Saddle River, NJ 07458
www.prenhall.com

Originally published in 2007 by Pearson Education, Inc.

With minor revisions, July, 2019.

Table of Contents

Preface

This book has three goals:

1. To introduce students to the elegant theory that underlies modern computing.
2. To motivate students by showing them that the theory is alive. While much of it has been known since the early days of digital computers (and some of it even longer), the theory continues to inform many of the most important applications that are considered today.
3. To show students how to start looking for ways to exploit the theory in their own work.

The core of the book, as a standard textbook, is Parts I through V. They address the first of the stated goals. They contain the theory that is being presented. There is more material in them than can be covered in a one-semester course. Sections that are marked with a ✹ are optional, in the sense that later material does not, for the most part, depend on them. The Course Plans section, below, suggests ways of selecting sections that are appropriate for some typical computer science courses.

Then there are three appendices:

- Appendix 𝔄 reviews the mathematical concepts on which the main text relies. Students should be encouraged to review it during the first week of class.
- Appendix 𝔅 treats selected theoretical concepts in greater depth. In particular, it contains the details of some proofs that are only sketched in the main text. It also contains a section on working with logical formulas (both Boolean and first-order).
- Appendix ℭ addresses the second and third goals. Its chapters present applications of the techniques that are described in the main body of the book. It also contains some interesting historical material. Although it is long (at least in comparison to the space that is devoted to applications in most other books in this area), it only skims the surface of the applications that it presents. But my hope is that that is enough. The World Wide Web has completely changed our ability to access knowledge. What matters now is to know that something exists and thus to look for it. The short discussions that are presented in Appendix ℭ will, I hope, give students that understanding.

There is a Web site that accompanies this book: http://www.theoryandapplications.org/. It is organized into the same sections as the book, so you can easily follow the two in parallel. The symbol 💻 following a concept in the text means that additional material is available on the Web site.

Throughout the text, you'll find pointers to the material in these appendices, as well as to material on the book's Web site. There are also some standalone application notes. These pointers and notes are enclosed in boxes, and refer you to the appropriate appendix and page number or to the Web. The appendix references look like this:

This technique really is useful. ℭ 718.

Notation

It is common practice to write definitions in the following form:

A something is a ***special something*** if it possesses property *P*.

This form is used even though property *P* is not only a sufficient but also a necessary condition for being a special something. For clarity we will, in those cases, write "if and only if", abbreviated "iff", instead of "if". So we will write:

A something is a ***special something*** iff it possesses property *P*.

Throughout the book we will, with a few exceptions, use the following naming conventions:

		Examples
sets	capital letters, early in the alphabet, plus *S*	*A*, *B*, *C*, *D*, *S*
logical formulas	capital letters, middle of the alphabet	*P*, *Q*, *R*
predicates and relations	capital letters, middle of the alphabet	*P*, *Q*, *R*
logical constants	subscripted *X*'s and specific names	X_1, X_2, John, Smoky
functions	lower case letters or words	*f*, *g*, *convert*
integers	lower case letters, middle of the alphabet	*i*, *j*, *k*, *l*, *m*, *n*
string-valued variables	lower case letters, late in the alphabet	*s*, *t*, *u*, *v*, *w*, *x*, *y*
literal strings	written in courier font	`abc`, `aabbb`
language-valued variables	upper case letters starting with *L*	L, L_1, L_2
specific languages	nonitalicized strings	$\mathrm{A}^n\mathrm{B}^n$, WW
regular expressions	lower case Greek letters	α, β, γ
states	lower case letters, middle of the alphabet	*p*, *q*, *r*, *s*, *t*
nonterminals in grammar rules	upper case letters	*A*, *B*, *C*, *S*, *T*
working strings in grammatical derivations	lower case Greek letter	α, β, γ
strings representing a PDA's stack	lower case Greek letter	α, β, γ
other variables	lower case letters, late in the alphabet	*x*, *y*, *z*

Programs and algorithms will appear throughout the book, stated at varying levels of detail. We will use the following formats for describing them:

- Exact code in some particular programming language will be written the same way other strings are written.

- Algorithms that are described in pseudocode will be written as:

```
Until an even-length string is found do
   Generate the next string in the sequence.
```

When we want to be able to talk about the steps, they will be numbered, so we will write:

```
1.  Until an even-length string is found do:
    1.1.  Generate the next string in the sequence.
2.  Reverse the string that was found.
```

When comments are necessary, as for example in code or in grammars, they will be preceded by the string /*.

Course Plans

Appendix 𝔄 summarizes the mathematical concepts on which the rest of the book relies. Depending on the background of the students, it may be appropriate to spend one or more lectures on this material. At the University of Texas, our students have had two prior courses in logic and discrete structures before they arrive in my class, so I have found that it is sufficient just to ask the students to read Appendix 𝔄 and to work a selection of the exercises that are provided at the end of it.

Part I lays the groundwork for the rest of the book. Chapter 2 is essential, since it defines the fundamental structures: strings and languages. I have found that it is very useful to cover Chapter 3, which presents a roadmap for the rest of the material. It helps students see where we are going and how each piece of the theory fits into the overall picture of a theory of computation. Chapter 4 introduces three ideas that become important later in the book. I have found that it may be better to skip Chapter 4 at the beginning of my class and to return to each of its sections once or twice later, as the concepts are required.

If the optional sections are omitted, Chapters 5, 6, 8, 9, 11 - 14, 17 - 21, and, optionally, 23 and/or 24 cover the material in a standard course in Automata Theory. Chapter 15 (Context-free Parsing) contains material that many computer science students need to see and it fits well into an Automata Theory course. I used to include much of it in my class. But that material is often taught in a course on Programming Languages or Compilers. In that case, it makes sense to omit it from the Automata Theory course. In its place, I now cover the optional material in Chapter 5, particularly the section on stochastic finite automata. I also cover Chapter 22. I've found that students are more motivated to tackle the difficult material (particularly the design of reduction proofs) in Chapter 21 if they can see ways in which the theory of undecidability applies to problems that are, to them, more intriguing than questions about the behavior of Turing machines.

This text is also appropriate for a broader course that includes the core of the classic theory of automata plus the modern theory of complexity. Such a course might cover Chapters 2 – 3, 5, 8, 11, 13, 17 – 21, and 27 – 30, omitting sections as time pressures require.

This text is unique in the amount of space it devotes to applications of the core theoretical material. In order to make the application discussions coherent, they are separated from the main text and occur in the Appendices at the end of the book. But I have found that I can substantially increase student interest in my course by sprinkling application discussions throughout the term. The application references that occur in the main text suggest places where it makes sense to do that.

Resources for Instructors

I have created a set of materials that have been designed to make it easy to teach from this book. In particular, there are:

- a complete set of Powerpoint slides,
- solutions to many of the Exercises, and
- additional problems, many of them with solutions.

If you are teaching a class and using this book, please write to ear@cs.utexas.edu and I will share these materials with you.

I would like to invite instructors who use this book to send me additional problems that can be shared with other users.

Acknowledgements

This book would not have been possible without the help of many people. When I first began teaching CS 341, Automata Theory, at the University of Texas, I was given a collection of notes that had been written by Bob Wall and Russell Williams. Much of the material in this book has evolved from those notes. I first learned automata theory from [Hopcroft and Ullman 1969]. Over the years that I have taught CS 341, I have used several textbooks, most frequently [Lewis and Papadimitriou 1988] and [Sipser 2006]. Much of what I have written here has been heavily influenced by the treatment of this material in those books.

Several of my friends, colleagues, and students have provided examples, answered numerous questions, and critiqued what I have written. I am particularly indebted to Don Baker, Volker Bandke, Jim Barnett, Jon Bentley, Gary Bland, Jaime Carbonell, Alan Cline, Martin Cohn, Dan Connolly, Ann Daniel, Chris Edmonson-Yurkanan, Scott Fahlman, Warren Gish, Mohamed Gouda, Jim Hendler, Oscar Hernandez, David Jefferson, Ben Kuipers, Greg Lavender, Tim Maxwell, Andy Mills, Jay Misra, Luay Nakhleh, Gordon Novak, Gabriela Ochoa, Dewayne Perry, Brian Reid, Bob Rich, Mike Scott, Cathy Stacy, Peter Stone, Lynda Trader, and David Zuckerman. Luay Nakhleh, Dan Tamir, and Bob Wall have used drafts of this book in their classes. I thank them for their feedback and that of their students.

I would also like to thank all of the students and teaching assistants who have helped me understand both why this material is hard and why it is exciting and useful. A couple of years ago, Tarang Mittal and Mat Crocker finished my class and decided that they should create an organized automata theory tutoring program the following fall. They got the program going and it continues to make a big difference to many students. I'd like to thank Tarang and Mat and the other tutors: Jason Pennington, Alex Menzies, Tim Maxwell, Chris St. Clair, Luis Guimbarda, Peter Olah, Eamon White, Kevin Kwast, Catherine Chu, Siddharth Natarajan, Daniel Galvan, Elton Pinto, and Jack Djeu.

My students have helped in many other ways as well. Oscar Hernandez helped me with several of the application appendices and made the Powerpoint slides that accompany the book. Caspar Lam designed the Web site for the book. David Reaves took pictures. My quilt, Blue Tweed, appears on the book's cover and on the Web site and slides. David took all the pictures that we used.

I would not have been in a position to write this book without the support of my father, who introduced me to the elegance of mathematics, Andy van Dam for my undergraduate experience at Brown, and Raj Reddy for my graduate experience at CMU. I cannot thank them enough.

Special thanks go to my family and friends, particularly my husband, Alan Cline and my father, Bob Rich, for countless meals taken over by discussions of this material, proofreading more drafts than I can count, and patience while living with someone who is writing a book.

Credits

On the Cover: On the Cover:

A quilt, Blue Tweed (1996, 53" x 80". Cotton. Machine pieced and quilted), made by the author. Notice that your eye fills in the vertical lines, so they appear to run the length of the quilt, even though the colors in the middle of the quilt are all about the same. Quilt photography by David Reaves.

Photo Credits:

- Photograph of a fragment of the Antikythera Mechanism and two photographs of the reconstructed model of it, page 795: copyright of the Antikythera Mechanism Research Project.
- Photo of Prague orlog, page 795: photo by DIGITALY @ www.orloj.com
- Photo of abacus, page 797, David Reeves.
- Photo of Jacquard loom, page 797: Stan Sherer.
- Photo of Sony Aibo robot, page 801: Alan Cline.

Credits for Exercises:

- Alan Cline: Exercise 27.9).
- [Brachman and Levesque 2004]: Exercise 33.9).
- Jay Misra: Exercise 20.10).
- Luay Nakhleh: Exercises 8.17), 17.5), 17.12), 21.18), 21.21), 21.22).
- Cathy Stacy: Exercise 5.3).
- David Zuckerman: Exercises 18.5), 28.11), 28.16), 28.23)d), 28.26), 29.3), 30.1).

Other Credits:

- IBM 7090 example, page 2: Brian Reid.
- IBM 360 JCL, page 2: Volker Bandke, http://www.bsp-gmbh.com/hercules/herc_jcl.html.
- The Java example, page 3: Mike Scott.
- Example 5.10, page 47: from [Misra 2004].
- The poem, "The Pumping Lemma for DFAs", page 143: Martin Cohn 💻.
- The drawings generated by Lindenmayer systems, pages 402 - 404: Generated by Alan Cline in Matlab®.
- Graph showing the growth rates of functions, page 441: Generated by Alan Cline in Matlab®.
- Progression of closures given in Example 32.11, page 582: Alan Cline.
- Example 32.19 (Generalized Modus Tollens), page 586 Alan Cline.
- Analysis of iterative deepening, page 647: Alan Cline.
- Proofs in Section 37.1, pages 653-658: Alan Cline.
- Network protocol diagrams and corresponding state machines, pages 694-698693 : Oscar Hernandez.
- A very long English sentence, page 743: http://www.plainenglish.co.uk/longsentences.htm.
- Drawing of girl with cat, page **Error! Bookmark not defined.**: Lynda Trader.
- Drawing of bear with rifle, page **Error! Bookmark not defined.**: Lynda Trader.
- Sound wave for the word "cacophony", page 755: Alan Cline.
- Simplified HMM for speech understanding, page 757754: Jim Barnett
- Drawing of the Towers of Hanoi, page 798: Alan Cline
- Schematic diagram and finite state diagram of a binary multiplier, page 800: Oscar Hernandez.
- Diagram of the FSM robot controller, page 802: Peter Stone.

Part I: Introduction

1 Why Study the Theory of Computation?

In this book, we present a theory of what can be computed and what cannot. We also sketch some theoretical frameworks that can inform the design of programs to solve a wide variety of problems. But why do we bother? Why don't we just skip ahead and write the programs that we need? This chapter is a short attempt to answer that question.

1.1 The Shelf Life of Programming Tools

Implementations come and go. In the somewhat early days of computing, programming meant knowing how to write code like[1]:

```
ENTRY       SXA     4,RETURN
            LDQ     X
            FMP     A
            FAD     B
            XCA
            FMP     X
            FAD     C
            STO     RESULT
RETURN      TRA     0

A           BSS     1
B           BSS     1
C           BSS     1
X           BSS     1
TEMP        BSS     1
STORE       BSS     1
            END
```

In 1957, Fortran appeared and made it possible for people to write programs that looked more straightforwardly like mathematics. By 1970, the IBM 360 series of computers was in widespread use for both business and scientific computing. To submit a job, one keyed onto punch cards a set of commands in OS/360 JCL (Job Control Language). Guruhood attached to people who actually knew what something like this meant[2]:

```
//MYJOB     JOB (COMPRESS),'VOLKER BANDKE',CLASS=P,COND=(0,NE)
//BACKUP  EXEC PGM=IEBCOPY
//SYSPRINT DD  SYSOUT=*
//SYSUT1   DD  DISP=SHR,DSN=MY.IMPORTNT.PDS
//SYSUT2   DD  DISP=(,CATLG),DSN=MY.IMPORTNT.PDS.BACKUP,
//             UNIT=3350,VOL=SER=DISK01,
//             DCB=MY.IMPORTNT.PDS,SPACE=(CYL,(10,10,20))
//COMPRESS EXEC PGM=IEBCOPY
//SYSPRINT DD  SYSOUT=*
//MYPDS    DD  DISP=OLD,DSN=*.BACKUP.SYSUT1
//SYSIN    DD  *
COPY INDD=MYPDS,OUTDD=MYPDS
//DELETE2 EXEC PGM=IEFBR14
//BACKPDS  DD  DISP=(OLD,DELETE,DELETE),DSN=MY.IMPORTNT.PDS.BACKUP
```

By the turn of the millennium, gurus were different. They listened to different music and had never touched a keypunch machine. But many of them did know that the following Java method (when compiled with the appropriate libraries) allows the user to select a file, which is read in and parsed using whitespace delimiters. From the parsed file, the program builds a frequency map, which shows how often each word occurs in the file:

[1] This program was written for the IBM 7090. It computes the value of a simple quadratic $ax^2 + bx + c$.
[2] It safely reorganizes and compresses a partitioned dataset.

```
public static TreeMap<String, Integer> create() throws IOException
    public static TreeMap<String, Integer> create() throws IOException
      {  Integer freq;
         String word;
         TreeMap<String, Integer> result = new TreeMap<String, Integer>();
         JFileChooser c = new JFileChooser();
         int retval = c.showOpenDialog(null);
         if (retval == JFileChooser.APPROVE_OPTION)
              {   Scanner s = new Scanner( c.getSelectedFile());
                  while( s.hasNext() )
                  {   word = s.next().toLowerCase();
                     freq = result.get(word);
                     result.put(word, (freq == null ? 1 : freq + 1));
                  }
              }
         return result;
      }
}
```

Along the way, other programming languages became popular, at least within some circles. There was a time when some people bragged that they could write code like[3]:

```
(⌈/V)>(+/V)-⌈/V
```

Today's programmers can't read code from 50 years ago. Programmers from the early days could never have imagined what a program of today would look like. In the face of that kind of change, what does it mean to learn the science of computing?

The answer is that there are mathematical properties, both of problems and of algorithms for solving problems, that depend on neither the details of today's technology nor the programming fashion *du jour*. The theory that we will present in this book addresses some of those properties. Most of what we will discuss was known by the early 1970s (barely the middle ages of computing history). But it is still useful in two key ways:

- It provides a set of abstract structures that are useful for solving certain classes of problems. These abstract structures can be implemented on whatever hardware/software platform is available.
- It defines provable limits to what can be computed, regardless of processor speed or memory size. An understanding of these limits helps us to focus our design effort in areas in which it can pay off, rather than on the computing equivalent of the search for a perpetual motion machine.

In this book our focus will be on analyzing problems, rather than on comparing solutions to problems. We will, of course, spend a lot of time solving problems. But our goal will be to discover fundamental properties of the problems themselves:

- Is there any computational solution to the problem? If not, is there a restricted but useful variation of the problem for which a solution does exist?
- If a solution exists, can it be implemented using some fixed amount of memory?
- If a solution exists, how efficient is it? More specifically, how do its time and space requirements grow as the size of the problem grows?

[3] An expression in the programming language APL 💻. It returns 1 if the largest value in a three element vector is greater than the sum of the other two elements, and 0 otherwise [Gillman and Rose 1984, p. 326]. Although APL is not one of the major programming languages in use today, its inventor, Kenneth Iverson, received the 1979 Turing Award for its development.

- Are there groups of problems that are equivalent in the sense that if there is an efficient solution to one member of the group there is an efficient solution to all the others?

1.2 Applications of the Theory Are Everywhere

Computers have revolutionized our world. They have changed the course of our daily lives, the way we do science, the way we entertain ourselves, the way that business is conducted, and the way we protect our security. The theory that we present in this book has applications in all of those areas. Throughout the main text, you will find notes that point to the more substantive application-focused discussions that appear in Appendix ℭ. Some of the applications that we'll consider are:

- Languages, the focus of this book, enable both machine/machine and person/machine communication. Without them, none of today's applications of computing could exist.

> Network communication protocols are languages. ℭ 693. Most Web pages are described using the Hypertext Markup Language, HTML. ℭ 805. The Semantic Web, whose goal is to support intelligent agents working on the Web, exploits additional layers of languages, such as RDF and OWL, that can be used to describe the content of the Web. ℭ 703. Music can be viewed as a language. And specialized languages enable composers to create new electronic music. ℭ 776. Even very unlanguage-like things, such as sets of pictures, can be viewed as languages by, for example, associating each picture with the program that drew it. ℭ 808.

- Both the design and the implementation of modern programming languages rely heavily on the theory of context-free languages that we will present in Part III. Context-free grammars are used to document the languages' syntax and they form the basis for the parsing techniques that all compilers use.

> The use of context-free grammars to define programming languages and to build their compilers is described in ℭ 664.

- People use natural languages, such as English, to communicate with each other. Since the advent of word processing, and then the Internet, we now type or speak our words to computers. So we would like to build programs to manage our words, check our grammar, search the World Wide Web, and translate from one language to another. Programs to do that also rely on the theory of context-free languages that we present in Part III.

> A sketch of some of the main techniques used in natural language processing can be found in ℭ 739.

- Systems as diverse as parity checkers, vending machines, communication protocols, and building security devices can be straightforwardly described as finite state machines, which we'll describe in Chapter 5.

> A vending machine is described in Example 5.1. A family of network communication protocols are modeled as finite state machines in ℭ 693. An example of a simple building security system, modeled as a finite state machine, can be found in ℭ 717. An example of a finite state controller for a soccer-playing robot can be found in ℭ 801.

- Many interactive video games are (large, often nondeterministic) finite state machines.

> An example of the use of a finite state machine to describe a role playing game can be found in ℭ 789.

- DNA is the language of life. DNA molecules, as well as the proteins that they describe, are strings that are made up of symbols drawn from small alphabets (nucleotides and amino acids, respectively). So computational biologists exploit many of the same tools that computational linguists use. For example, they rely on techniques that are based on both finite state machines and context-free grammars.

For a very brief introduction to computational biology see ℭ 727.

- Security is perhaps the most important property of many computer systems. The undecidability results that we present in Part IV show that there cannot exist a general purpose method for automatically verifying arbitrary security properties of programs. The complexity results that we present in Part V serve as the basis for powerful encryption techniques.

For a proof of the undecidability of the correctness of a very simple security model, see ℭ 718. For a short introduction to cryptography, see ℭ 722.

- Artificial intelligence programs solve problems in task domains ranging from medical diagnosis to factory scheduling. Various logical frameworks have been proposed for representing and reasoning with the knowledge that such programs exploit. The undecidability results that we present in Part IV show that there cannot exist a general theorem prover that can decide, given an arbitrary statement in first order logic, whether or not that statement follows from the system's axioms. The complexity results that we present in Part V show that, if we back off to the far less expressive system of Boolean (propositional) logic, while it becomes possible to decide the validity of a given statement, it is not possible to do so, in general, in a reasonable amount of time.

For a discussion of the role of undecidability and complexity results in artificial intelligence, see ℭ 758. The same issues plague the development of the Semantic Web. ℭ 703.

- Clearly documented and widely accepted standards play a pivotal role in modern computing systems. Getting a diverse group of users to agree on a single standard is never easy. But the undecidability and complexity results that we present in Parts IV and V mean that, for some important problems, there is no single right answer for all uses. Expressively weak standard languages may be tractable and decidable, but they may simply be inadequate for some tasks. For those tasks, expressively powerful languages, that give up some degree of tractability and possibly decidability, may be required. The provable lack of a one-size-fits-all language makes the standards process even more difficult and may require standards that allow alternatives.

We'll see one example of this aspect of the standards process when we consider, in ℭ 703, the design of a description language for the Semantic Web.

- Many natural structures, including ones as different as organic molecules and computer networks, can be modeled as graphs. The theory of complexity that we present in Part V tells us that, while there exist efficient algorithms for answering some important questions about graphs, other questions are "hard", in the sense that no efficient algorithm for them is known nor is one likely to be developed.

We'll discuss the role of graph algorithms in network analysis in ℭ 701.

- The complexity results that we present in Part V contain a lot of bad news. There are problems that matter for which no efficient algorithm is likely ever to be found. But practical solutions to some of these problems exist. They rely on a variety of approximation techniques that work pretty well most of the time.

An almost optimal solution to an instance of the traveling salesman problem with 1,904,711 cities has been found, as we'll see in Section 27.1. Randomized algorithms can find prime numbers efficiently, as we'll see in Section 30.2.4. Heuristic search algorithms find paths in computer games, ℭ 790, and move sequences for champion chess-playing programs, ℭ 785.

2 Languages and Strings

In the theory that we are about to build, we are going to analyze problems by casting them as instances of the more specific question, "Given some string *s* and some language *L*, is *s* in *L*?" Before we can formalize what we mean by that, we need to define our terms.

An ***alphabet***, often denoted Σ, is a finite set. We will call the members of Σ ***symbols*** or ***characters***.

2.1 Strings

A ***string*** is a finite sequence, possibly empty, of symbols drawn from some alphabet Σ. Given any alphabet Σ, the shortest string that can be formed from Σ is the empty string, which we will write as ε. The set of all possible strings over an alphabet Σ is written Σ*. This notation exploits the Kleene star operator, which we will define more generally below.

Example 2.1 Alphabets

Alphabet name	*Alphabet symbols*	*Example strings*
The English alphabet	`{a, b, c, …, z}`	ε, `aabbcg`, `aaaaa`
The binary alphabet	`{0, 1}`	ε, `0`, `001100`
A star alphabet	{★, ✪, ✫, ✬, ✭, ✡, ✰}	ε, ✪✪, ✪✭✭✰★✰
A music alphabet	{𝅝, 𝅗𝅥, 𝅘𝅥, 𝅘𝅥𝅮, 𝅘𝅥𝅯, 𝅘𝅥𝅰, |}	ε, 𝅝|𝅗𝅥𝅗𝅥|𝅗𝅥𝅘𝅥𝅘𝅥|

In running text, we will indicate literal symbols and strings by writing them `like this`.

2.1.2 Functions on Strings

The ***length*** of a string *s*, which we will write as |*s*|, is the number of symbols in *s*. For example:

$|\varepsilon| = 0$
$|\texttt{1001101}| = 7$

For any symbol *c* and string *s*, we define the function $\#_c(s)$ to be the number of times that the symbol *c* occurs in *s*. So, for example, $\#_{\texttt{a}}(\texttt{abbaaa}) = 4$.

The ***concatenation*** of two strings *s* and *t*, written *s* || *t* or simply *st*, is the string formed by appending *t* to *s*. For example, if *x* = `good` and *y* = `bye`, then *xy* = `goodbye`. So $|xy| = |x| + |y|$.

The empty string, ε, is the identity for concatenation of strings. So $\forall x\ (x\, \varepsilon = \varepsilon\, x = x)$.

Concatenation, as a function defined on strings, is associative. So $\forall s, t, w\ ((st)w = s(tw))$.

Next we define string ***replication***. For each string *w* and each natural number *i*, the string w^i is defined as:

$w^0 = \varepsilon$
$w^{i+1} = w^i\, w$

For example:

$\texttt{a}^3 = \texttt{aaa}$
$(\texttt{bye})^2 = \texttt{byebye}$
$\texttt{a}^0\texttt{b}^3 = \texttt{bbb}$

Finally we define string ***reversal***. For each string w, the reverse of w, which we will write w^R, is defined as:

if $|w| = 0$ then $w^R = w = \varepsilon$
if $|w| \geq 1$ then $\exists a \in \Sigma\ (\exists u \in \Sigma^*\ (w = ua))$. (i.e., the last character of w is a.) Then define $w^R = a\ u^R$.

Theorem 2.1 Concatenation and Reverse of Strings

Theorem: If w and x are strings, then $(w\ x)^R = x^R\ w^R$. For example, $(\texttt{nametag})^R = (\texttt{tag})^R\ (\texttt{name})^R = \texttt{gateman}$.

Proof: The proof is by induction on $|x|$:

Base case: $|x| = 0$. Then $x = \varepsilon$, and $(wx)^R = (w\ \varepsilon)^R = (w)^R = \varepsilon\ w^R = \varepsilon^R\ w^R = x^R\ w^R$.

Prove: $\forall n \geq 0\ (((|x| = n) \rightarrow ((w\ x)^R = x^R\ w^R)) \rightarrow ((|x| = n + 1) \rightarrow ((w\ x)^R = x^R\ w^R)))$.

Consider any string x, where $|x| = n + 1$. Then $x = u\ a$ for some character a and $|u| = n$. So:

$(w\ x)^R$	$= (w\ (u\ a))^R$	rewrite x as ua
	$= ((w\ u)\ a)^R$	associativity of concatenation
	$= a\ (w\ u)^R$	definition of reversal
	$= a\ (u^R\ w^R)$	induction hypothesis
	$= (a\ u^R)\ w^R$	associativity of concatenation
	$= (ua)^R\ w^R$	definition of reversal
	$= x^R\ w^R$	rewrite ua as x

■

2.1.3 Relations on Strings

A string s is a ***substring*** of a string t iff s occurs contiguously as part of t. For example:

`aaa`	is a substring of	`aaabbbaaa`
`aaaaaa`	is not a substring of	`aaabbbaaa`

A string s is a ***proper substring*** of a string t iff s is a substring of t and $s \neq t$.

Every string is a substring (although not a proper substring) of itself. The empty string, ε, is a substring of every string.

A string s is a ***prefix*** of t iff $\exists x \in \Sigma^*\ (t = sx)$. A string s is a ***proper prefix*** of a string t iff s is a prefix of t and $s \neq t$. Every string is a prefix (although not a proper prefix) of itself. The empty string, ε, is a prefix of every string. For example, the prefixes of `abba` are: ε, `a`, `ab`, `abb`, `abba`.

A string s is a ***suffix*** of t iff $\exists x \in \Sigma^*\ (t = xs)$. A string s is a ***proper suffix*** of a string t iff s is a suffix of t and $s \neq t$. Every string is a suffix (although not a proper suffix) of itself. The empty string, ε, is a suffix of every string. For example, the suffixes of `abba` are: ε, `a`, `ba`, `bba`, `abba`.

2.2 Languages

A ***language*** is a (finite or infinite) set of strings over a finite alphabet Σ. When we are talking about more than one language, we will use the notation Σ_L to mean the alphabet from which the strings in the language L are formed.

Example 2.2 Defining Languages Given an Alphabet

Let Σ = {a, b}. Σ* = {ε, a, b, aa, ab, ba, bb, aaa, aab, …}.

Some examples of languages over Σ are:

∅, {ε}, {a, b}, {ε, a, aa, aaa, aaaa, aaaaa}, {ε, a, aa, aaa, aaaa, aaaaa, …}

2.2.2 Techniques for Defining Languages

We will use a variety of techniques for defining the languages that we wish to consider. Since languages are sets, we can define them using any of the set-defining techniques that are described in Section 32.2. For example, we can specify a characteristic function, i.e., a predicate that is *True* of every element in the set and *False* of everything else.

Example 2.3 All a's Precede All b's

Let *L* = {*x* ∈ {a, b}* : all a's precede all b's in *w*}. The strings ε, a, aa, aabbb, and bb are in *L*. The strings aba, ba, and abc are not in *L*. Notice that some strings trivially satisfy the requirement for membership in *L*. The rule says nothing about there having to be any a's or any b's. All it says is that any a's there are must come before all the b's (if any). If there are no a's or no *b*'s, then there can be none that violate the rule. So the strings ε, a, aa, and bb trivially satisfy the rule and are in *L*.

Example 2.4 Strings That End in a

Let *L* = {*x* : ∃*y* ∈ {a, b}* (*x* = *y*a)}. The strings a, aa, aaa, bbaa, and ba are in *L*. The strings ε, bab, and bca are not in *L*. *L* consists of all strings that can be formed by taking some string in {a, b}* and concatenating a single a onto the end of it.

Example 2.5 The Perils of Using English to Describe Languages

Let *L* = {*x*#*y*: *x*, *y* ∈ {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}* and, when *x* and *y* are viewed as the decimal representations of natural numbers, *square*(*x*) = *y*}. The strings 3#9 and 12#144 are in *L*. The strings 3#8, 12, and 12#12#12 are not in *L*. But what about the string #? Is it in *L*? It depends on what we mean by the phrase, "when *x* and *y* are viewed as the decimal representations of natural numbers". Is ε the decimal representation of some natural number? It is possible that an algorithm that converts strings to numbers might convert ε to 0. In that case, since 0 is the square of 0, # is in *L*. If, on the other hand, the string-to-integer converter fails to accept ε as a valid input, # is not in *L*. This example illustrates the dangers of using English descriptions of sets. They are sometimes ambiguous. We will strive to use only unambiguous terms. We will also, as we discuss below, develop other definitional techniques that do not present this problem.

Example 2.6 The Empty Language

Let *L* = {} = ∅. *L* is the language that contains no strings.

Example 2.7 The Empty Language is Different From the Empty String

Let *L* = {ε}, the language that contains a single string, ε. Note that *L* is different from ∅.

All of the examples we have considered so far fit the definition that we are using for the term *language*: a set of strings. They're quite different, though, from the everyday use of the term. Everyday languages are also languages under our definition:

Example 2.8 English Isn't a Well-Defined Language

Let L = {w: w is a sentence in English}.

Examples:	Kerry hit the ball.	/* Clearly in L.
	Colorless green ideas sleep furiously.[4]	/* The syntax is correct but what could it mean?
	The window needs fixed.	/* In some dialects of L.
	Ball the Stacy hit blue.	/* Clearly not in L.

The problem with languages like this is that there is no clear agreement on what strings they contain. We will not be able to apply the theory that we are about to build to any language for which we cannot first produce a formal specification. Natural languages, like English or Spanish or Chinese, while hard to specify, are of great practical importance, though. As a result, substantial effort has been expended in creating formal and computationally effective descriptions of them that are good enough to be used as the basis for applications such as grammar checking and text database retrieval.

To the extent that formal descriptions of natural languages like English can be created, the theory that we are about to develop can be applied, as we will see in Parts II and III and ℭ 739.

Example 2.9 A Halting Problem Language

Let L = {w: w is a C program that halts on all inputs}. L is substantially more complex than, for example, $\{x \in \{\texttt{a,b}\}^* :$ all `a`'s precede all `b`'s$\}$. But, unlike English, there does exist a clear formal specification of it. The theory that we are about to build will tell us something very useful about L.

We can use the relations that we have defined on strings as a way to define languages.

Example 2.10 Using the Prefix Relation

We define the following languages in terms of the prefix relation on strings:

$L_1 = \{w \in \{\texttt{a}, \texttt{b}\}^*$: no prefix of w contains $\texttt{b}\}$
$= \{\varepsilon, \texttt{a}, \texttt{aa}, \texttt{aaa}, \texttt{aaaa}, \texttt{aaaaa}, \texttt{aaaaaa}, \ldots\}$.

$L_2 = \{w \in \{\texttt{a}, \texttt{b}\}^*$: no prefix of w starts with $\texttt{b}\}$
$= \{w \in \{\texttt{a}, \texttt{b}\}^*$: the first character of w is $\texttt{a}\} \cup \{\varepsilon\}$.

$L_3 = \{w \in \{\texttt{a}, \texttt{b}\}^*$: every prefix of w starts with $\texttt{b}\}$
$= \varnothing$.

L_3 is equal to $\varnothing$ because ε is a prefix of every string. Since ε does not start with `b`, no strings meet L_3's requirement.

Recall that we defined the replication operator on strings: For any string s and integer n, $s^n = n$ copies of s concatenated together. For example, $(\texttt{bye})^2 = \texttt{byebye}$. We can use replication as a way to define a language, rather than a single string, if we allow n to be a variable, rather than a specific constant.

Example 2.11 Using Replication to Define a Language

Let $L = \{\texttt{a}^n : n \geq 0\}$. $L = \{\varepsilon, \texttt{a}, \texttt{aa}, \texttt{aaa}, \texttt{aaaa}, \texttt{aaaaa}, \ldots\}$.

Languages are sets. So, if we want to provide a computational definition of a language, we could specify either:

- a language generator, which enumerates (lists) the elements of the language, or

[4] This classic example of a syntactically correct but semantically anomalous sentence is from [Chomsky 1957].

- a language recognizer, which decides whether or not a candidate string is in the language and returns *True* if it is and *False* if it isn't.

For example, the logical definition, $L = \{x : \exists y \in \{\texttt{a}, \texttt{b}\}^* \ (x = y\texttt{a})\}$, can be turned into either a language generator (enumerator) or a language recognizer.

In some cases, when considering an enumerator for a language L, we may care about the order in which the elements of L are generated. If there exists a total order D of the elements of Σ_L (as there does, for example, on the letters of the Roman alphabet or the symbols for the digits 0 – 9), then we can use D to define on L a useful total order called ***lexicographic order*** (written $<_L$):

- Shorter strings precede longer ones: $\forall x \ (\forall y \ ((|x| < |y|) \rightarrow (x <_L y)))$, and
- Of strings that are the same length, sort them in dictionary order using D.

When we use lexicographic order in the rest of this book, we will assume that D is the standard sort order on letters and numerals. If D is not obvious, we will state it.

We will say that a program ***lexicographically enumerates*** the elements of L iff it enumerates them in lexicographic order.

Example 2.12 Lexicographic Enumeration

Let $L = \{x \in \{\texttt{a},\texttt{b}\}^* :$ all `a`'s precede all `b`'s$\}$. The lexicographic enumeration of L is:

ε, `a`, `b`, `aa`, `ab`, `bb`, `aaa`, `aab`, `abb`, `bbb`, `aaaa`, `aaab`, `aabb`, `abbb`, `bbbb`, `aaaaa`, …

In Parts II, III, and IV of this book, we will consider a variety of formal techniques for specifying both generators (enumerators) and recognizers for various classes of languages.

2.2.3 What is the Cardinality of a Language?

How large is a language? The smallest language over any alphabet is $\varnothing$, whose cardinality is 0. The largest language over any alphabet Σ is Σ^*. What is $|\Sigma^*|$? Suppose that $\Sigma = \varnothing$. Then $\Sigma^* = \{\varepsilon\}$ and $|\Sigma^*| = 1$. But what about the far more useful case in which Σ is not empty?

Theorem 2.2 The Cardinality of Σ^*

Theorem: If $\Sigma \neq \varnothing$ then Σ^* is countably infinite.

Proof: The elements of Σ^* can be lexicographically enumerated by a straightforward procedure that:
- Enumerates all strings of length 0, then length 1, then length 2, and so forth.
- Within the strings of a given length, enumerates them in dictionary order.

This enumeration is infinite since there is no longest string in Σ^*. By Theorem 32.1, since there exists an infinite enumeration of Σ^*, it is countably infinite.

■

Since any language over Σ is a subset of Σ^*, the cardinality of every language is at least 0 and at most $\aleph_0$. So all languages are either finite or countably infinite.

2.2.4 How Many Languages Are There?

Let Σ be an alphabet. How many different languages are there that are defined on Σ? The set of languages defined on Σ is $\mathcal{P}(\Sigma^*)$, the power set of Σ^*, or the set of all subsets of Σ^*. If $\Sigma = \varnothing$ then Σ^* is $\{\varepsilon\}$ and $\mathcal{P}(\Sigma^*)$ is $\{\varnothing, \{\varepsilon\}\}$. But, again, what about the useful case in which Σ is not empty?

Theorem 2.3 An Uncountably Infinite Number of Languages

Theorem: If $\Sigma \neq \varnothing$ then the set of languages over Σ is uncountably infinite.

Proof: The set of languages defined on Σ is $\mathcal{P}(\Sigma^*)$. By Theorem 2.2, Σ^* is countably infinite. By Theorem 32.4, if S is a countably infinite set, $\mathcal{P}(S)$ is uncountably infinite. So $\mathcal{P}(\Sigma^*)$ is uncountably infinite.

■

2.2.5 Functions on Languages

Since languages are sets, all of the standard set operations are well-defined on languages. In particular, we will find union, intersection, difference, and complement to be useful. Complement will be defined with Σ^* as the universe unless we explicitly state otherwise.

Example 2.13 Set Functions Applied to Languages

Let: $\Sigma = \{\texttt{a}, \texttt{b}\}$.
L_1 = {strings with an even number of `a`'s}.
L_2 = {strings with no `b`'s} = {ε, `a`, `aa`, `aaa`, `aaaa`, `aaaaa`, `aaaaaa`, …}.

$L_1 \cup L_2$ = {all strings of just `a`'s plus strings that contain `b`'s and an even number of `a`'s}.
$L_1 \cap L_2$ = {ε, `aa`, `aaaa`, `aaaaaa`, `aaaaaaaa`…}.
$L_2 - L_1$ = {`a`, `aaa`, `aaaaa`, `aaaaaaa`, …}.
$\neg(L_2 - L_1)$ = {strings with at least one `b`} $\cup$ {strings with an even number of `a`'s}.

Because languages are sets of strings, it makes sense to define operations on them in terms of the operations that we have already defined on strings. Three useful ones to consider are concatenation, Kleene star, and reverse.

Let L_1 and L_2 be two languages defined over some alphabet Σ. Then their ***concatenation***, written L_1L_2 is:

$$L_1L_2 = \{w \in \Sigma^* : \exists s \in L_1\ (\exists t \in L_2\ (w = st))\}.$$

Example 2.14 Concatenation of Languages

Let L_1 = {`cat`, `dog`, `mouse`, `bird`}.
L_2 = {`bone`, `food`}.

L_1L_2 = {`catbone`, `catfood`, `dogbone`, `dogfood`, `mousebone`, `mousefood`, `birdbone`, `birdfood`}.

The language {ε} is the identity for concatenation of languages. So, for all languages L, $L\{\varepsilon\} = \{\varepsilon\}L = L$.

The language $\varnothing$ is a zero for concatenation of languages. So, for all languages L, $L\,\varnothing = \varnothing\, L = \varnothing$. That $\varnothing$ is a zero follows from the definition of the concatenation of two languages as the set consisting of all strings that can be formed by selecting some string s from the first language and some string t from the second language and then concatenating them together. There are no ways to select a string from the empty set.

Concatenation, as a function defined on languages, is associative. So, for all languages L_1, L_2, and L_3:

$$((L_1L_2)L_3 = L_1(L_2L_3)).$$

It is important to be careful when concatenating languages that are defined using replication. Recall that we used the notation $\{\texttt{a}^n : n \geq 0\}$ to mean the set of strings composed of zero or more `a`'s. That notation is a shorthand for a longer, perhaps clearer expression, $\{w : \exists n \geq 0\ (w = \texttt{a}^n)\}$. In this form, it is clear that n is a variable bound by an existential quantifier. We will use the convention that the scope of such quantifiers is the entire expression in which they occur. So multiple occurrences of the same variable letter are the same variable and must take on the same value. Suppose that $L_1 = \{\texttt{a}^n : n \geq 0\}$ and $L_2 = \{\texttt{b}^n : n \geq 0\}$. By the definition of language concatenation, $L_1L_2 = \{w : w$

consists of a (possibly empty) a region followed by a (possibly empty) b region}. $L_1L_2 \neq \{a^nb^n : n \geq 0\}$, since every string in $\{a^nb^n : n \geq 0\}$ must have the same number of b's as a's. The easiest way to avoid confusion is simply to rename conflicting variables before attempting to concatenate the expressions that contain them. So $L_1L_2 = \{a^nb^m : n, m \geq 0\}$. In Chapter 6 we will define a convenient notation that will let us write this as a*b*.

Let L be a language defined over some alphabet Σ. Then the ***Kleene star*** of L, written L^* is:

$$L^* = \{\varepsilon\} \cup \{w \in \Sigma^* : \exists k \geq 1 \ (\exists w_1, w_2, \ldots w_k \in L \ \ (w = w_1 \, w_2 \ldots w_k))\}.$$

In other words, L^* is the set of strings that can be formed by concatenating together zero or more strings from L.

Example 2.15 Kleene Star

Let L = {dog, cat, fish}. Then:

L^* = {ε, dog, cat, fish, dogdog, dogcat, …, fishdog, …, fishcatfish, fishdogfishcat, …}.

Example 2.16 Kleene Star, Again

Let $L = \{w \in \{a, b\}^* : \#_a(w) \text{ is odd}\}$. Then $L^* = \{w \in \{a, b\}^* : \#_b(w) \text{ is even}\}$. The constraint on the number of a's almost disappears in the description of L^* because strings in L^* are formed by concatenating together any number of strings from L. If an odd number of strings are concatenated together, the result will contain an odd number of a's. If an even number (including none) are used, the result will contain an even number of a's.

L^* always contains an infinite number of strings as long as L is not equal to either $\varnothing$ or {ε} (i.e., as long as there is at least one nonempty string any number of which can be concatenated together). If $L = \varnothing$, then $L^* = \{\varepsilon\}$, since there are no strings that could be concatenated to ε to make it longer. If $L = \{\varepsilon\}$, then L^* is also {ε}.

It is sometimes useful to require that at least one element of L be selected. So we define:

$$L^+ = L\,L^*.$$

Another way to describe L^+ is that it is the closure of L under concatenation. Note that $L^+ = L^* - \{\varepsilon\}$ iff $\varepsilon \notin L$.

Example 2.17 L^+

Let $L = \{0, 1\}^+$ be the set of binary strings. L does not include ε.

Let L be a language defined over some alphabet Σ. Then the ***revers e*** of L, written L^R is:

$$L^R = \{w \in \Sigma^* : w = x^R \text{ for some } x \in L\}.$$

In other words, L^R is the set of strings that can be formed by taking some string in L and reversing it.

Since we have defined the reverse of a language in terms of the definition of reverse applied to strings, we expect it to have analogous properties.

Theorem 2.4 Concatenation and Reverse of Languages

Theorem: If L_1 and L_2 are languages, then $(L_1\,L_2)^R = L_2^R\,L_1^R$.

Proof:	If x and y are strings, then $\forall x\ (\forall y\ ((xy)^R = y^R x^R))$	Theorem 2.1
	$(L_1 L_2)^R = \{(xy)^R : x \in L_1 \text{ and } y \in L_2\}$	Definition of concatenation of languages
	$= \{y^R x^R : x \in L_1 \text{ and } y \in L_2\}$	Lines 1 and 2
	$= L_2^R\, L_1^R$	Definition of concatenation of languages

■

We have now defined the two important data types, string and language, that we will use throughout this book. In the next chapter, we will see how we can use them to define a framework that will enable us to analyze computational problems of all sorts (not just ones you may naturally think of in terms of strings).

2.2.6 Assigning Meaning to the Strings of a Language

Sometimes we are interested in viewing a language just as a set of strings. For example, we'll consider some important formal properties of the language we'll call $A^nB^n = \{a^n b^n : n \geq 0\}$. In other words, A^nB^n is the language composed of all strings of `a`'s and `b`'s such that all the `a`'s come first and the number of `a`'s equals the number of `b`'s. We won't attempt to assign meanings to any of those strings.

But some languages are useful precisely because their strings do have meanings. We use natural languages like English and Chinese because they allow us to communicate ideas. A program in a language like Java or C^{++} or Perl also has a meaning. In the case of a programming language, one way to define meaning is in terms of some other (typically closer to machine architecture) language. So, for example, the meaning of a Java program can be described as a Java Virtual Machine program. An alternative is to define a program's meaning in a logical language.

Philosophers and linguists (and others) have spent centuries arguing about what sentences in natural languages like English (or Sanskrit or whatever) mean. We won't attempt to solve that problem here. But if we are going to work with formal languages, we need a precise way to map each string to its meaning (also called its ***semantics***). We'll call a function that assigns meanings to strings a ***semantic interpretation function***. Most of the languages we'll be concerned with are infinite because there is no bound on the length of the strings that they contain. So it won't, in general, be possible to define meanings by a table that pairs each string with its meaning.

We must instead define a function that knows the meanings of the language's basic units and can combine those meanings, according to some fixed set of rules, to build meanings for larger expressions. We call such a function, which can be said to "compose" the meanings of simpler constituents into a single meaning for a larger expression, a ***compositional semantic interpretation function***. There arguably exists a mostly compositional semantic interpretation function for English. Linguists fight about the gory details of what such a function must look like. Everyone agrees that words have meanings and that one can build a meaning for a simple sentence by combining the meanings of the subject and the verb. For example, speakers of English would have no trouble assigning a meaning to the sentence, "I gave him the fizding," provided that they are told what the meaning of the word "fizding" is. Everyone also agrees that the meaning of idioms, like "I'm going to give him a piece of my mind," cannot be derived compositionally. Some other issues are more subtle.

> Languages whose strings have meaning pervade computing and its applications. Boolean logic and first-order logic are languages. Programming languages are languages. ℭ 664. Network protocols are languages. ℭ 693. Database query languages are languages. ℭ 805. HTML is a language for defining Web pages. ℭ 805. XML is a more general language for marking up data. ℭ 805. OWL is a language for defining the meaning of tags on the Web. ℭ 714. BNF is a language that can be used to specify the syntax of other languages. ℭ 664. DNA is a language for describing proteins. ℭ 727. Music is a language based on sound. ℭ 776.

When we define a formal language for a specific purpose, we design it so that there exists a compositional semantic interpretation function. So, for example, there exist compositional semantic interpretation functions for programming languages like Java and C^{++}. There exists a compositional semantic interpretation function for the language of Boolean logic. It is specified by the truth tables that define the meanings of whichever operators (e.g., $\wedge$, $\vee$, $\neg$, and $\rightarrow$) are allowed.

One significant property of semantic interpretation functions for useful languages is that they are generally not one-to-one. Consider:

- English: The sentences, “Chocolate, please,”, “I’d like chocolate,”, “I’ll have chocolate,” and “I guess chocolate today,” all mean the same thing, at least in the context of ordering an ice cream cone.

- Java: The following chunks of code all do the same thing:

```
int x = 4;            int x = 4;            int x = 4;            int x = 4;
x++;                  ++x;                  x = x + 1;            x = x --1;
```

The semantic interpretation functions that we will describe later in this book, for example for the various grammar formalisms that we will introduce, will not be one-to-one either.

2.3 Exercises

1) Consider the language $L = \{1^n2^n : n > 0\}$. Is the string 122 in L?

2) Let $L_1 = \{a^nb^n : n > 0\}$. Let $L_2 = \{c^n : n > 0\}$. For each of the following strings, state whether or not it is an element of L_1L_2:
   a) ε.
   b) aabbcc.
   c) abbcc.
   d) aabbcccc.

3) Let L_1 = {peach, apple, cherry} and L_2 = {pie, cobbler, ε}. List the elements of L_1L_2 in lexicographic order.

4) Let $L = \{w \in \{a, b\}^* : |w| \equiv_3 0\}$. List the first six elements in a lexicographic enumeration of L. (See page 571 for the definintion of $\equiv_3$.)
5) Consider the language L of all strings drawn from the alphabet {a, b} with at least two different substrings of length 2.
   a) Describe L by writing a sentence of the form $L = \{w \in \Sigma^* : P(w)\}$, where Σ is a set of symbols and P is a first-order logic formula. You may use the function $|s|$ to return the length of s. You may use all the standard relational symbols (e.g., =, ≠, <, etc.), plus the predicate *Substr*(s, t), which is *True* iff s is a substring of t.
   b) List the first six elements of a lexicographic enumeration of L.

6) For each of the following languages L, give a simple English description . Show two strings that are in L and two that are not (unless there are fewer than two strings in L or two not in L, in which case show as many as possible).
   a) $L = \{w \in \{a, b\}^*$: exactly one prefix of w ends in a$\}$.
   b) $L = \{w \in \{a, b\}^*$: all prefixes of w end in a$\}$.
   c) $L = \{w \in \{a, b\}^* : \exists x \in \{a, b\}^+ (w = axa)\}$.

7) Are the following sets closed under the following operations? If not, what are their respective closures?
   a) The language {a, b} under concatenation.
   b) The odd length strings over the alphabet {a, b} under Kleene star.
   c) $L = \{w \in \{a, b\}^*\}$ under reverse.
   d) $L = \{w \in \{a, b\}^*$: w starts with a$\}$ under reverse.
   e) $L = \{w \in \{a, b\}^*$: w ends in a$\}$ under concatenation.

8) For each of the following statements, state whether it is *True* or *False*. Prove your answer.
   a) $\forall L_1, L_2$ ($L_1 = L_2$ iff $L_1^* = L_2^*$).
   b) $(\varnothing \cup \varnothing^*) \cap (\neg\varnothing - (\varnothing\varnothing^*)) = \varnothing$ (where $\neg\varnothing$ is the complement of $\varnothing$).

c) Every infinite language is the complement of a finite language.
d) $\forall L\ ((L^R)^R = L)$.
e) $\forall L_1, L_2\ ((L_1\ L_2)^* = L_1^*\ L_2^*)$.
f) $\forall L_1, L_2\ ((L_1^* L_2^* L_1^*)^* = (L_2 \cup L_1)^*)$.
g) $\forall L_1, L_2\ ((L_1 \cup L_2)^* = L_1^* \cup L_2^*)$.
h) $\forall L_1, L_2, L_3\ ((L_1 \cup L_2)\ L_3 = (L_1\ L_3) \cup (L_2\ L_3))$.
i) $\forall L_1, L_2, L_3\ ((L_1\ L_2) \cup L_3 = (L_1 \cup L_3)\ (L_2 \cup L_3))$.
j) $\forall L\ ((L^+)^* = L^*)$.
k) $\forall L\ (\varnothing L^* = \{\varepsilon\})$.
l) $\forall L\ (\varnothing \cup L^+ = L^*)$.
m) $\forall L_1, L_2\ ((L_1 \cup L_2)^* = (L_2 \cup L_1)^*)$.

3 The Big Picture: A Language Hierarchy

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

3.1 Defining the Task: Language Recognition

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

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

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

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

3.2 The Power of Encoding

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

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

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

3.2.1 Everything is a String

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

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

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

Example 3.1 Pattern Matching on the Web

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

Example 3.2 Question-Answering on the Web

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

The techniques that we will describe in the rest of this book are widely used in the construction of systems that work with natural language (e. g., English or Spanish or Chinese) text and speech inputs. ℭ 739.

Example 3.3 Does a Program Always Halt?

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

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

Example 3.4 Primality Testing

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

Prime numbers play an important role in modern cryptography systems. ℭ 722. We'll discuss the complexity of PRIMES in Section 28.1.7 and again in Section 30.2.4.

Example 3.5 Verifying Addition

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

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

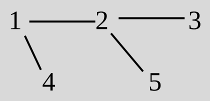
- The language to be decided:

  INTEGERSUM = {*w* of the form: <$integer_1$>+<$integer_2$>=<$integer_3$> :
  each of the substrings <$integer_1$>, <$integer_2$>, and <$integer_3$> is an element of {`0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`}$^+$
  and $integer_3$ is the sum of $integer_1$ and $integer_2$}.

- Examples of strings in *L*: `2+4=6` `23+47=70`.
- Examples of strings not in *L*: `2+4=10` `2+4`.

Example 3.6 Graph Connectivity

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



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

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

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

- The language to be decided:

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

Example 3.7 Protein Sequence Alignment

- Problem: given a protein fragment *f* and a complete protein molecule *p*, could *f* be a fragment from *p*?
- Encoding of the problem: represent each protein molecule or fragment as a sequence of amino acid residues. Assign a letter to each of the 20 possible amino acids. So a protein fragment might be represented as `AGHTYWDNR`.
- The language to be decided: {<*f*, *p*> : *f* could be a fragment from *p*}.

The techniques that we will describe in the rest of this book are widely used in computational biology. ℭ 727.

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

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

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

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

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

This idea is the basis for the design of binary adders, as well as larger circuits, like multipliers, that exploit them. ℭ 800.

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

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

We'll return to this issue in Section 27.3.1.

3.2.2 Casting Problems as Decision Problems

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

Example 3.8 Casting Addition as Decision

- Problem: given two nonnegative integers, compute their sum.
- Encoding of the problem: we transform the problem of adding two numbers into the problem of checking to see whether a third number is the sum of the first two. We can use the same encoding that we used in Example 3.5.

- The language to be decided:

  INTEGERSUM = {*w* of the form: <*integer*$_1$>+<*integer*$_2$>=<*integer*$_3$>,
  where each of the substrings <*integer*$_1$>, <*integer*$_2$>, and <*integer*$_3$> is an element of
  `{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}`$^+$ and *integer*$_3$ is the sum of *integer*$_1$ and *integer*$_2$}.

Example 3.9 Casting Sorting as Decision

- Problem: given a list of integers, sort it.
- Encoding of the problem: we transform the problem of sorting a list into the problem of examining a pair of lists and deciding whether the second corresponds to the sorted version of the first.
- The language to be decided:

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

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

Example 3.10 Casting Database Querying as Decision

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

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

- Example of a string in *L*:

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

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

```
Given a string of the form <integer_1>+<integer_2>=<integer_3> do:
1. Let $x$ = convert-to-integer(<integer_1>).
2. Let $y$ = convert-to-integer(<integer_2>).
3. Let $z = P(x, y)$.
4. If $z$ = convert-to-integer(<integer_3>) then accept. Else reject.
```

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

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

3.3 A Machine-Based Hierarchy of Language Classes

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

3.3.1 The Regular Languages

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

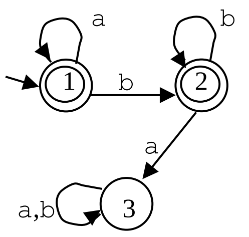


Figure 3.1 A simple FSM

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

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

3.3.2 The Context-Free Languages

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

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

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

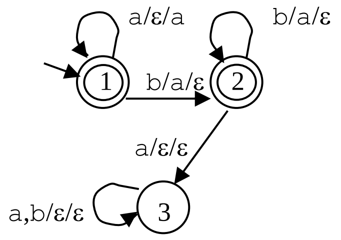


Figure 3.2 A simple PDA that accepts A^nB

Using a very similar sort of PDA, we can build a machine to accept Bal and other languages whose strings are composed of properly nested substrings. For example, a ***palindrome*** is a string that reads the same right-to-left as it does left-to right. We can easily build a PDA to accept the language PalEven = $\{ww^R : w \in \{$a, b$\}^*\}$, the language of even-length palindromes of a's and b's. The PDA for PalEven simply pushes all the characters in the first half of its input string onto the stack, guesses where the middle is, and then starts popping one character for each remaining input character. If there is a guess that causes the pushed string (which will be popped off in reverse order) to match the remaining input string, then the input string is in PalEven.

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

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

3.3.3 The Decidable and Semidecidable Languages

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

But it is easy to write a program to accept $A^nB^nC^n$. So, if we want a class of machines that can capture everything we can write programs to compute, we need a model that is stronger than the PDA. To meet this need, we will introduce a third kind of machine. We will get rid of the stack and replace it with an infinite tape. The tape will have a single read/write head. Only the tape square under the read/write head can be accessed (for reading or for writing). The read/write head can be moved one square in either direction on each move. The resulting machine is called a ***Turing***

machine. We will also change the way that input is given to the machine. Instead of streaming it, one character at a time, the way we did for FSMs and PDAs, we will simply write the input string onto the tape and then start the machine with the read/write head just to the left of the first input character. We show the structure of a Turing machine in Figure 3.3. The arrow under the tape indicates the location of the read/write head.

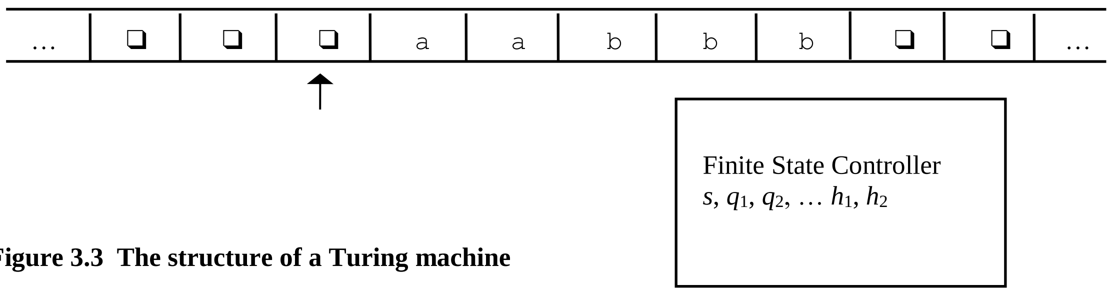


Figure 3.3 The structure of a Turing machine

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

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

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

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

- A language *L* is ***decidable*** iff there exists a Turing machine *M* that halts on all inputs, accepts all strings that are in *L*, and rejects all strings that are not in *L*. In other words, *M* can always say yes or no, as appropriate.

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

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

3.3.4 The Computational Hierarchy and Why It Is Important

We have now defined four language classes:

- regular languages, which can be accepted by some finite state machine,
- context-free languages, which can be accepted by some pushdown automaton,
- decidable (or simply D) languages, which can decided by some Turing machine that always halts, and
- semidecidable (or SD) languages, which can be *semi*-decided by some Turing machine that halts on all strings in the language.

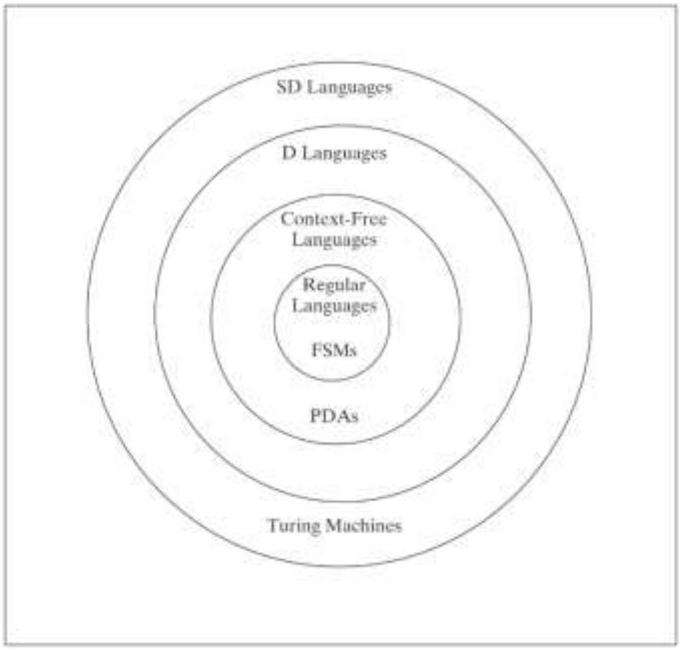


Figure 3.4: A hierarchy of language classes

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

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

- ***Computational efficiency***: Finite state machines run in time that is linear in the length of the input string. A general context-free parser based on the idea of a pushdown automaton requires time that grows as the cube of the length of the input string. A Turing machine may require time that grows exponentially with the length of the input string.

- ***Decidability***: There exist procedures to answer many useful questions about finite state machines. For example, does an FSM accept some particular string? Is an FSM minimal (i.e., is it the simplest machine that does the job it does)? Are two FSMs identical? A subset of those questions can be answered for pushdown automata. None of them can be answered for Turing machines.

- ***Clarity***: There exist tools that enable designers to draw and analyze finite state machines. Every regular language can also be described using the (often very convenient) regular expression pattern language that we will define in Chapter 6. Every context-free language, in addition to being recognizable by some pushdown automaton, can (as we will see in Chapter 11) be described with a context-free grammar. For many important kinds of languages, context-free grammars are sufficiently natural that they are commonly used as documentation tools. No corresponding tools exist for the broader classes of decidable and semidecidable languages.

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

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

Although stated in the context of the World Wide Web, the rule of least power applies far more broadly. We're appealing to a generalization of it here. We'll return to a discussion of it in the specific context of the Semantic Web in ℭ 703.

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

3.4 A Tractability Hierarchy of Language Classes

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

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

- ***P***, which contains those languages that can be decided in time that grows as some polynomial function of the length of the input.

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

- ***PSPACE***, which contains those languages that can be decided by a machine whose space requirement grows as some polynomial function of the length of the input.

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

$$\text{P} \subseteq \text{NP} \subseteq \text{PSPACE}$$

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

3.5 Exercises

1) Consider the following problem: Given a digital circuit *C*, does *C* output 1 on all inputs? Describe this problem as a language to be decided.

2) Using the technique we used in Example 3.8 to describe addition, describe square root as a language recognition problem.

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

3) Consider the problem of encrypting a password, given an encryption key. Formulate this problem as a language recognition problem.

4) Consider the optical character recognition (OCR) problem: Given an array of black and white pixels and a set of characters, determine which character best matches the pixel array. Formulate this problem as a language recognition problem.

5) Consider the language $A^nB^nC^n = \{a^nb^nc^n : n \geq 0\}$, discussed in Section 3.3.3. We might consider the following design for a PDA to accept $A^nB^nC^n$: As each `a` is read, push two `a`'s onto the stack. Then pop one `a` for each `b` and one `a` for each `c`. If the input and the stack come out even, accept. Otherwise reject. Why doesn't this work?

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

4 Computation

Our goal in this book is to be able to make useful claims about problems and the programs that solve them. Of course, both problem specifications and the programs that solve them take many different forms. Specifications can be written in English, or as a set of logical formulas, or as a set of input/output pairs. Programs can be written in any of a wide array of common programming languages. As we said in the last chapter, in this book we are, for the most part, going to depart from those standard methods and, instead:

- Define problems as languages to be decided, and
- Define programs as state machines whose input is a string and whose output is *Accept* or *Reject*.

Both because of this change in perspective and because we are going to introduce two ideas that are not common in everyday programming practice, we will pause, in this chapter, and look at what we mean by computation and how we are going to go about it. In particular, we will examine three key ideas:

1. Decision procedures.
2. Nondeterminism.
3. Functions on languages (alternatively, programs that operate on other programs).

Once we have finished this discussion, we will begin our examination of the language classes that we outlined in Chapter 3.

4.1 Decision Procedures

Recall that a ***decision problem*** is one in which we must make a yes/no decision. An ***algorithm*** is a detailed procedure that accomplishes some clearly specified task. A ***decision procedure*** is an algorithm to solve a decision problem. Put another way, it is a program whose result is a Boolean value. Note that, in order to be guaranteed to return a Boolean value, a decision procedure must be guaranteed to halt on all inputs.

This book is about decision procedures. We will spend most of our time discussing decision procedures to answer questions of the form:

- Is string s in language L?

But we will also attempt to answer other questions, in particular ones that ask about the machines that we will build to answer the first group of questions. So we may ask questions such as:

- Given a machine (an FSM, a PDA, or a Turing machine), does it accept any strings?
- Given two machines, do they accept the same strings?
- Given a machine, is it the smallest (simplest) machine that does its job?

If we have in mind a decision problem to which we want an answer, there are three things we may want to know:

1. Does there exist a decision procedure, i.e., an algorithm, to answer the question? A decision problem is ***decidable*** iff the answer to this question is yes. A decision problem is ***undecidable*** iff the answer to this question is no. A decision problem is ***semidecidable*** iff there exists an algorithm that halts and returns *True* iff *True* is the answer. When *False* is the answer, it may either halt and return *False* or it may loop. Some undecidable problems are semidecidable; some are not even that.
2. If any decision procedures exist, find one.
3. Again, if any decision procedures exist, what is the most efficient one and how efficient is it?

In the early part of this book, we will ask questions for which decision procedures exist and we will often skip directly to question 2. But, as we progress, we will begin to ask questions for which, provably, no decision procedure exists. It is because there are such problems that we have articulated question 1.

Decision procedures are programs. They must possess two correctness properties:

- The program must be guaranteed to halt on all inputs.
- When the program halts and returns an answer, it must be the correct answer for the given input.

Let's consider some examples.

Example 4.1 Checking for Even Numbers

Is the integer x even? This one is easy. Assume that / performs (truncating) integer division. Then the following program answers the question:

```
even(x: integer) =
    If (x / 2) ∗ 2 = x then return True else return False.
```

Example 4.2 Checking for Prime Numbers

Is the positive integer x prime? Given an appropriate string encoding, this problem corresponds to the language PRIMES that we defined in Example 3.4. Defining a procedure to answer this question is not hard, although it will require a loop and so it will be necessary to prove that the loop always terminates. Several algorithms that solve this problem exist. Here's an easy one:

```
prime(x: positive integer) =
    For i = 2 to ceiling(sqrt(x)) do:
        If (x / i) ∗ i = x then return False.
    Return True.
```

The function *ceiling*(x), also written $\lceil x \rceil$ returns the smallest integer that is greater than or equal to x. This program is guaranteed to halt. The natural numbers between 0 and *ceiling*(*sqrt*(x)) - 2 form a well-ordered set under $\leq$. Let *index* correspond to *ceiling*(*sqrt*(x)) - i. At the beginning of the first pass through the loop, the value of *index* is *ceiling*(*sqrt*(x)) - 2. The value of *index* decreases by one each time through the loop. The loop ends when that value becomes 0. It's worth pointing out that, while this program is simple and it is easy to prove that it is correct, it is not the most efficient program that we could write. We'll have more to say about this problem in Sections 28.1.7 and 30.2.4.

For our next few questions we need a definition. The sequence:

$$F_n = 2^{2^n} + 1,\ n \geq 0,$$

defines the Fermat numbers 💻. The first few Fermat numbers are:

$$F_0 = 3,\ F_1 = 5,\ F_2 = 17,\ F_3 = 257,\ F_4 = 65{,}537,\ F_5 = 4{,}294{,}967{,}297.$$

Example 4.3 Checking for Small Prime Fermat Numbers

Are there any prime Fermat numbers less than 1,000,000? There exists a simple decision procedure to answer this question:

```
fermatSmall() =
    i = 0.
    Repeat:
        candidate = (2 ∗∗ (2 ∗∗ i)) + 1.
        If candidate is prime then return True.
        i = i + 1.
    until candidate ≥ 1,000,000.
    Return False.
```

This algorithm is guaranteed to halt because the value of *candidate* increases each time through the loop and the loop terminates when its value exceeds a fixed bound. We will skip the proof that the correct answer is returned.

Example 4.4 Checking for Large Prime Fermat Numbers

Are there any prime Fermat numbers greater than 1,000,000? This question is different in one important way from the previous one. Does there exist a decision procedure to answer this question? What about:

```
fermatLarge() =
    i = 0.
    Repeat:
        candidate = (2 ** (2 ** i)) + 1.
        If candidate > 1,000,000 and is prime then return True.
        i = i + 1.
    Return False.
```

What can we say about this program? If there is a prime Fermat number greater than 1,000,000 *fermatLarge* will find it and will halt. But suppose that there is no such number. Then the program will loop forever. *FermatLarge* is not capable of returning *False* even if *False* is the correct answer. So, is *fermatLarge* a decision procedure? No. A decision procedure must halt and return the correct answer, whatever that is.

Can we do better? Is there a decision procedure to answer this question? Yes. Since this question takes no arguments, it has a simple answer, either *True* or *False*. So either

```
fermatYes() =
    Return True.
```

or

```
fermatNo() =
    Return False.
```

correctly answers the question. Our problem now is, "Which one?" No one knows. Fermat himself was only able to generate the first five Fermat numbers, and, on that basis, conjectured that all Fermat numbers are prime. If he had been right, then *fermatYes* answers the question. However, it now seems likely that there are no prime Fermat numbers greater than 65,537. A substantial effort 💻 continues to be devoted to finding one, but so far the only discoveries have been larger and larger composite Fermat numbers. But there is also no proof that a larger prime one does not exist nor is there an algorithm for finding one. We simply do not know.

Example 4.5 Checking for Programs That Halt on a Particular Input

Now consider a problem that is harder and that cannot be solved by a simple constant function such as *fermatYes* or *fermatNo*. Given an arbitrary Java program *p* that takes a string *w* as an input parameter, does *p* halt on some particular value of *w*? Here's a candidate for a decision procedure:

```
haltsOnw(p: program, w: string) =
    1. Simulate the execution of p on w.
    2. If the simulation halts return True else return False.
```

Is *haltsOnw* a decision procedure? No, because it can never return the value *False*. Yet *False* is sometimes the correct answer (since there are (*p*, *w*) pairs such that *p* fails to halt on *w*). When *haltsOnw* should return *False*, it will loop forever in step 1. Can we do better? No. It is possible to prove, as we will do in Chapter 19, that no decision procedure for this question exists.

Define a ***semidecision procedure*** to be a procedure that halts and returns *True* whenever *True* is the correct answer. But, whenever *False* is the correct answer, it may return *False* or it may loop forever. In other words, a semidecision procedure knows when to say yes but it is not guaranteed to know when to say no. A ***semidecidable problem*** is a problem for which a semidecision procedure exists. Example 4.5 is a semidecidable problem. While some semidecidable problems are also decidable, that one isn't.

Example 4.6 Checking for Programs That Halt on All Inputs

Now consider an even harder problem: Given an arbitrary Java program that takes a single string as an input parameter, does it halt on all possible input values? Here's a candidate for a decision procedure:

```
haltsOnAll(program) =
    1. For i = 1 to infinity do:
            Simulate the execution of program on all possible input strings of length i.
    2. If all of the simulations halt return True else return False.
```

HaltsOnAll will never halt on any program since, to do so, it must try running the program on an infinite number of strings. And there is not a better procedure to answer this question. We will show, in Chapter 21, that it is not even semidecidable.

The bottom line is that there are three kinds of questions:

- those for which a decision procedure exists.
- those for which no decision procedure exists but a semidecision procedure exists.
- those for which not even a semi-decision procedure exists.

As we move through the language classes that we will consider in this book, we will move from worlds in which there exist decision procedures for just about every question we can think of to worlds in which there exist some decision procedures and perhaps some semidecision procedures, all the way to worlds in which there do not exist even semidecision procedures.

But keep in mind throughout that entire progression what a *decision* procedure is. It is an algorithm that is *guaranteed* to halt on all inputs.

4.2 Determinism and Nondeterminism

Imagine adding to a programming language the function *choose*, which may be written in either of the following forms:

-
  ```
  choose  (action 1;;
                  action 2;;
             …
          action n  )
  ```

- *choose* (x from S: $P(x)$)

In the first form, *choose* is presented with a finite list of alternatives, each of which will return either a successful value or the value *False*. *Choose* will:

- Return some successful value, if there is one.
- If there is no successful value, then choose will:
  - Halt and return *False* if all the actions halt and return *False*.
  - Fail to halt if any of the actions fails to halt. We want to define *choose* this way since any path that has not halted still has the potential to return a successful value.

In the second form, *choose* is presented with a set *S* of values. *S* may be finite or it may be infinite if it is specified by a generator. *Choose* will:

- Return some element *x* of *S* such that *P*(*x*) halts with a value other than *False*, if there is one.
- If there is no such element, then *choose* will:
  - Halt and return *False* if it can be determined that, for all elements *x* of *S*, *P*(*x*) is not satisfied. This will happen if *S* is finite and there is a procedure for checking *P* that always halts. It may also happen, even if *S* is infinite, if there is some way, short of checking all the elements, to determine that no elements that satisfy *P* exist.
  - Fail to halt if there is no mechanism for determining that no elements of *S* that satisfy *P* exist. This may happen either because *S* is infinite or because there is no algorithm, guaranteed to halt on all inputs, that checks for *P* and returns *False* when necessary.

In both forms, the job of *choose* is to find a successful value (which we will define to be any value other than *False*) if there is one. When we don't care which successful value we find (or how we find it), *choose* is a useful abstraction, as we will see in the next few examples.

We will call programs that are written in our new language that includes *choose* ***nondeterministic***. We will call programs that are written without using *choose* ***deterministic***.

Real computers are, of course, deterministic. So, if *choose* is going to be useful, there must exist a way to implement it on a deterministic machine. For now, however, we will be noncommittal as to how that is done. It may try the alternatives one at a time, or it may pursue them in parallel. If it tries them one at a time, it may try them in the order listed, in some random order, or in some order that is carefully designed to maximize the chances of finding a successful value without trying all the others. The only requirement is that it must pursue the alternatives in some fashion that is guaranteed to find a successful value if there is one. The point of the *choose* function is that we can separate the design of the choosing mechanism from the design of the program that needs a value and calls *choose* to find it one.

Example 4.7 Nondeterministically Choosing a Travel Plan

Suppose that we regularly plan medium length trips. We are willing to drive or to fly and rent a car or to take a train and use public transportation if it is available when we get there, as long as the total cost of the trip and the total time required are reasonable. We don't care about small differences in time or cost enough to make it worth exhaustively exploring all the options every time. We can define the function *trip-plan* to solve our problem:

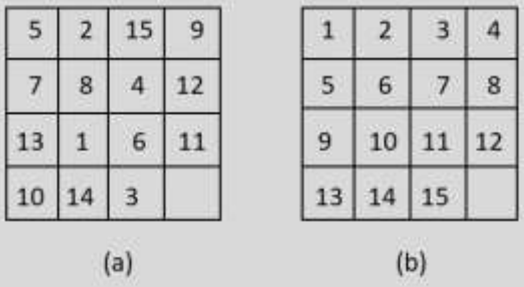
```
trip-plan(start, finish) =
   Return (choose(fly-major-airline-and-rent-car(start, finish);;
                  fly-regional-airline-and-rent-car(start, finish);;
                  take-train-and-use-public-transportation(start, finish);;
                  drive(start, finish)   )).
```

Each of the four functions *trip-plan* calls returns with a successful value iff it succeeds in finding a plan that meets the cost and time requirements. Probably the first three of them are implemented as an Internet agent that visits the appropriate Web sites, specifies the necessary parameters, and waits to see if a solution can be found. But notice that *trip-plan* can return a result as soon as at least one of the four agents finds an acceptable solution. It doesn't care whether the four agents can be run in parallel or are tried sequentially. It just wants to know if there's a solution and, if so, what it is.

A good deal of the power of choose comes from the fact that it can be called recursively. So it can be used to describe a search process, without having to specify the details of how the search is conducted.

Example 4.8 Nondeterministically Searching a Space of Puzzle Moves

Suppose that we want to solve the 15-puzzle. We are given two configurations of the puzzle, for example the ones shown here labeled (a) and (b). The goal is to begin in configuration (a) and, through a sequence of moves, reach configuration (b). The only allowable move is to slide a numbered tile into the blank square.



Using *choose*, we can easily write *solve-15*, a program that finds a solution if there is one. The idea is that *solve*-15 will guess at a first move. From the board configuration that results from that move, it will guess at a second move. From there, it will guess at a third move, and so on. If it reaches the goal configuration, it will report the sequence of moves that got it there.

Using the second form of *choose* (in which values are selected from a set that can be generated each time a new choice must be made), we can define *solve*-15 so that it returns an ordered list of board positions. The first element of the list corresponds to the initial configuration. Following that, in order, are the configurations that result from each of the moves. The final configuration will correspond to the goal. So the result of a call to *solve*-15 will describe a move sequence that corresponds to a solution to the original problem. We'll invoke *solve*-15 with a list that contains just the initial configuration. So we define:

```
solve-15(position-list) =
    /* Explore moves available from the last board configuration to have been generated.
    current = last(position-list).
    If current = solution then return (position-list).
    /* Assume that successors(current) returns the set of configurations that can be generated by one
          legal move from current. Then choose picks one with the property that, once it has been appended
          to position-list, solve-15 can continue and find a solution. We assume that append destructively
          modifies its first argument.
    choose (x from successors(current): solve-15(append(position-list, x))).
    Return position-list.
```

If there is a solution to a particular instance of the 15-puzzle, *solve-15* will find it. If we care about how efficiently the solution is found, then we can dig inside the implementation of *choose* and try various strategies, including:

- checking to make sure we don't generate a board position that has already been explored, or
- sorting the successors by how close they are to the goal.

But if we don't care about how *choose* works, we don't have to.

15-puzzle configurations can be divided into two equivalence classes. Every configuration can be transformed into every other configuration in the same class and into none of the configurations in the other class.

Many decision problems can be solved straightforwardly using *choose*.

Example 4.9 Nondeterministically Searching for a Satisfying Assignment

A wff in Boolean logic is ***satisfiable*** iff it is true for at least one assignment of truth values to the literals it contains. Now consider the following problem, which we'll call SAT: Given a Boolean wff *w*, decide whether or not *w* is satisfiable.

To see how we might go about designing a program to solve the SAT problem, consider an example wff $w = P \wedge (Q \vee R) \wedge \neg(R \vee S) \rightarrow Q$. We can build a program that considers the predicate symbols (in this case *P*, *Q*, *R*, and *S*) in some order. For each one, it will pick one of the two available values, *True* or *False*, and assign it to all occurrences of that predicate symbol in *w*. When no predicate symbols remain, all that is necessary is to use the truth table definitions of the logical operators to simplify *w* until it has evaluated to either *True* or *False*. If *True*, then we have found an assignment of values to the predicates that makes *w* true; *w* is satisfiable. If *False,* then this path fails to find such an assignment and it fails. This procedure must halt because *w* contains only a finite number of predicate symbols, one is eliminated at each step, and there are only two values to choose from at each step. So either some path will return *True* or all paths will eventually halt and return *False*.

The following algorithm returns *True* if the answer to the question is yes and *False* if the answer to the question is no:

```
decideSAT(w: Boolean wff) =
    If there are no predicate symbols in w then:
        Simplify w until it is either True or False.
        Return w.
    Else:
        Find P, the first predicate symbol in w.
        /* Let w/P/x mean the wff w with every instance of P replaced by x.
        Return choose (decideSAT(w/P/True);;
                       decideSAT(w/P/False)).
```

One way to envision the execution of a program like *solve*-15 or *decideSAT* is as a search tree. Each node in the tree corresponds to a snapshot of *solve*-15 or *decideSAT* and each path from the root to a leaf node corresponds to one computation that *solve*-15 or *decideSAT* might perform. For example, if we invoke *decideSAT* on the input $P \wedge \neg R$, the set of possible computations can be described by the tree in Figure 4.1. The first level in the tree corresponds to guessing a value for *P* and the second level corresponds to guessing a value for *R*.

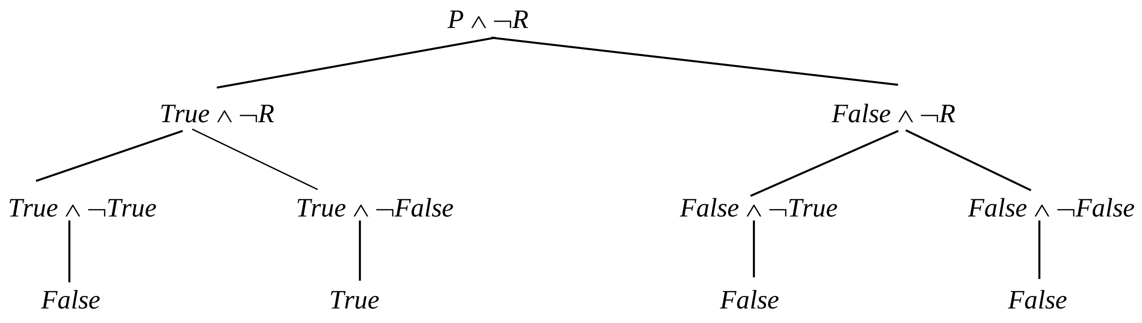


Figure 4.1 A search tree created by ***decideSAT*** **on the input** $P \wedge \neg R$**.**

Since there exists at least one computational path that succeeds (i.e., returns a value other than *False*), *decideSAT* will pick the value returned by one such path and return it. So *decideSAT* will return *True*. It may do so after exploring all four of the paths shown above (if it is unlucky choosing an order in which to explore the paths). Or it may guess correctly and find the successful path without considering any of the others.

Efficient algorithms for solving Boolean satisfiability problems are important in a wide variety of domains. No general and efficient algorithms are known. But, in 𝔅 612, we'll describe ordered binary decision diagrams (OBDDs), which are used in SAT solvers that work, in practice, substantially more efficiently than *decideSAT* does.

One of the most important properties of programs that exploit *choose* is clear from the simple tree that we just examined: Guesses that do not lead to a solution can be effectively ignored in any analysis that is directed at determining the program's result.

Does adding *choose* to our programming language let us solve any problems that we couldn't solve without it? The answer to that question turns out to depend on what else the programming language already lets us do.

Suppose, for example, that we are describing our programs as finite state machines (FSMs). One way to add *choose* to the FSM model is to allow two or more transitions, labeled with the same input character, to emerge from a single state. We show a simple example in Figure 4.2.

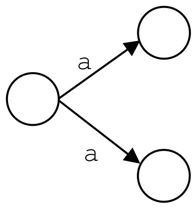


Figure 4.2 A nondeterministic FSM with two competing transitions labeled `a`

We'll say that an nondeterministic FSM *M* (i.e., one that may exploit *choose*) accepts iff at least one of its paths accepts. It will reject iff all of its paths reject. So *M*'s job is to find an accepting path if there is one. If it succeeds, it can ignore all other paths. If *M* exploits *choose* and does contain competing transitions, then one way to view its behavior is that it makes a guess and chooses an accepting path if it can.

While we will find it very convenient to allow nondeterminism like this in finite state machines, we will see in Section 5.4 that, whenever there is a nondeterministic FSM to accept some language *L*, there is also a (possibly much larger and more complicated) deterministic FSM that accepts *L*. So adding *choose* doesn't change the class of languages that can be accepted.

Now suppose that we are describing our programs as pushdown automata (PDAs). Again we will add *choose* to the model by allowing competing transitions coming out of a state. As we will see in Chapter 13, now the answer is that adding *choose* adds power. There are languages that can be accepted by PDAs that exploit *choose* that cannot be accepted by any PDA that does not exploit it.

Lastly, suppose that we are describing our programs as Turing machines or as code in a standard, modern programming language. Then, as we will see in Chapter 17, we are back to the situation we were in with FSMs. Nondeterminism is a very useful design tool that lets us specify complex programs without worrying about the details of how the search is managed. But, if there is a nondeterministic Turing machine that solves a problem, then there is a deterministic one (one that does not exploit *choose*) that also solves the problem.

In the two cases (FSMs and Turing machines) in which adding *choose* does not add computational power to our model, we will see that it does add descriptive power. We'll see examples for which a very simple nondeterministic machine can do the work of a substantially more complex deterministic one. We'll present algorithms, for both FSMs and Turing machines, that construct, given an arbitrary nondeterministic machine, an equivalent deterministic one. Thus we can use nondeterminism as an effective design tool and leave the job of building a deterministic program to a compiler.

In Part V, we will take a different look at analyzing problems and the programs that solve them. There we will be concerned with the complexity of the solution: how much running time does it take or how much memory does it require? In that analysis, nondeterminism will play another important role. It will enable us to separate our solution to a problem into two parts:

1. The complexity of an individual path through the search tree that *choose* creates. Each such path will typically correspond to checking one complete guess to see if it is a solution to the problem we are trying to solve.
2. The total complexity of the entire search process.

So, although nondeterminism may at first seem at odds with our notion of effective computation, we will find throughout this book that it is a very useful tool in helping us to analyze problems and see how they fit into each of the models that we will consider.

For some problems, it is useful to extend *choose* to allow probabilities to be associated with each of the alternatives. For example, we might write:

```
choose ( (.5) action 1;;
         (.3) action 2;;
         (.2) action 3  )
```

For some applications, the semantics we will want for this extended form of *choose* will be that exactly one path should be pursued. Let Pr(n) be the probability associated with alternative n. Then *choose* will select alternative n with probability Pr(n). For other applications, we will want a different semantics: All paths should be pursued and a total probability should be associated with each path as a function of the set of probabilities associated with each step along the path. We will have more to say about how these probabilities actually work when we talk about specific applications.

4.3 Functions on Languages and Programs

In Chapter 2, we described some useful functions on languages. We considered simple functions such as complement, concatenation, union, intersection, and Kleene star. All of those were defined by straightforward extension of the standard operations on sets and strings. Functions on languages are not limited to those, however. In this section, we mention a couple of others, which we'll come back to at various points throughout this book.

Example 4.10 The Function *chop*

Define $chop(L) = \{w : \exists x \in L\ (x = x_1 c x_2 \wedge x_1 \in \Sigma_L^* \wedge x_2 \in \Sigma_L^* \wedge c \in \Sigma_L \wedge |x_1| = |x_2| \wedge w = x_1 x_2)\}$. In other words, *chop*(L) is all the odd length strings in L with their middle character chopped out.

Recall the language $A^nB^n = \{a^n b^n : n \geq 0\}$. What is *chop*($A^nB^n$)? The answer is $\varnothing$, since there are no odd length strings in A^nB^n.

What about $A^nB^nC^n = \{a^n b^n c^n : n \geq 0\}$? What is *chop*($A^nB^nC^n$)? Approximately half of the strings in $A^nB^nC^n$ have odd length and so can have their middle character chopped out. Strings in $A^nB^nC^n$ contribute strings to *chop*($A^nB^nC^n$) as follows:

n	**in $A^nB^nC^n$**	**in *chop*($A^nB^nC^n$)**
0	ε	
1	abc	ac
2	aabbcc	
3	aaabbbccc	aaabbccc
4	aaaabbbbcccc	
5	aaaaabbbbbccccc	aaaaabbbbccccc

So $chop(A^nB^nC^n) = \{a^{2n+1} b^{2n} c^{2n+1} : n \geq 0\}$.

Example 4.11 The Function *firstchars*

Define $firstchars(L) = \{w : \exists y \in L\ (y = cx \wedge c \in \Sigma_L \wedge x \in \Sigma_L^* \wedge w \in \{c\}^*)\}$. So we could compute *firstchars*(L) by looking at all the strings in L, finding all the characters that start such strings, and then, for each such character c, adding to *firstchars*(L) all the strings in $\{c\}^*$. Let's look at *firstchars* applied to some languages:

L	***firstchars*(*L*)**
$\varnothing$	$\varnothing$
{ε}	$\varnothing$
{a}	{a}*
A^nB^n	{a}*
{a, b}*	{a}* ∪ {b}*

Given some function f on languages, we may want to ask the question, "If L is a member of some language class C, what can we say about $f(L)$? Is it too a member of C? Alternatively, is the class C closed under f?"

Example 4.12 Are Language Classes Closed Under Various Functions?

Consider two classes of languages, INF (the set of infinite languages) and FIN (the set of finite languages). And consider four of the functions we have discussed: union, intersection, *chop* and *firstchars*. We will ask the question, "Is class C closed under function f?" The answers are (with the number in each cell pointing to an explanation below for the corresponding answer):

	FIN	*INF*
intersection	yes (2)	no (6)
chop	yes (3)	no (7)
firstchars	no (4)	yes (8)

(1) For any sets A and B, $|A \cup B| \leq |A| + |B|$.

(2) For any sets A and B, $|A \cap B| \leq min(|A|, |B|)$.

(3) Each string in L can generate at most one string in *chop*(L), so $|chop(L)| \leq |L|$.

(4) To show that any class C is not closed under some function f, it is sufficient to show a single counterexample: a language L where $L \in C$ but $f(L) \notin C$. We showed such a counterexample above: *firstchars*({a}) = {a}*.

(5) For any sets A and B, $|A \cup B| \geq |A|$.

(6) We show one counterexample: Let L_1 = {a}* and L_2 = {b}*. L_1 and L_2 are infinite. But $L_1 \cap L_2 = \{\varepsilon\}$, which is finite.

(7) We have already shown a counterexample: A^nB^n is infinite. But *Chop*(A^nB^n) = $\varnothing$, which is finite.

(8) If L is infinite, then it contains at least one string of length greater than 0. That string has some first character c. Then $\{c\}^* \subseteq$ *firstchars*(L) and $\{c\}^*$ is infinite.

In the rest of this book, we will discuss the four classes of languages: regular, context-free, decidable, and semidecidable, as described in Chapter 3. One of the questions we will ask for each of them is whether they are closed under various operations.

Given some function f on languages, how can we:
1. Implement f ?
2. Show that some class of languages is closed under f ?

The answer to question 2 is generally by construction. In other words, we will show an algorithm that takes a description of the input language(s) and constructs a description of the result of applying f to that input. We will then use that constructed description to show that the resulting language is in the class we care about. So our ability to answer both questions 1 and 2 hinges on our ability to define an algorithm that computes f, given a description of its input (which is one or more languages).

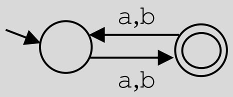
In order to define an algorithm A to compute some function f, we first need a way to define the input to A. Defining A is going to be very difficult if we allow, for example, English descriptions of the language(s) on which A is supposed to operate. What we need is a formal model that is exactly powerful enough to describe the languages on which we would like A to be able to run. Then A could use the description(s) of its input language(s) to build a new description, using the same model, of the result of applying f.

Example 4.13 Representing Languages So That Functions Can Be Applied

Suppose that we wish to compute the function union. It will be very hard to implement union if we allow input language description such as:

1. $\{w \in \{$`a`, `b`$\}^* : w$ has an odd number of characters$\}$.
2. $\{w \in \{$`a`, `b`$\}^* : w$ has an even number of `a`'s$\}$.
3. $\{w \in \{$`a`, `b`$\}^* :$ all `a`'s in w precede all `b`'s$\}$.

Suppose, on the other hand, that we describe each of these languages as a finite state machine that accepts them. So, for example, language 1 would be represented as



In Chapter 8, we will show an algorithm that, given two FSMs, corresponding to two regular languages, L_1 and L_2, constructs a new FSM that accepts the union of L_1 and L_2,.

If we use finite state machines (or pushdown automata or Turing machines) as input I to an algorithm A that computes some function f, then what A will do is to manipulate those FSMs (or PDAs or Turing machines) and produce a new one that accepts the language $f(I)$. If we think of the input FSMs (or PDAs or Turing machines) as programs, then A is a program whose input and output are other programs.

Lisp is a programming language that makes it easy to write programs that manipulate programs. ℭ 671.

Programs that write other programs are not particularly common, but they are not fundamentally different from programs that work with any other data type. Programs in any conventional programming language can be expressed as strings, so any program that can manipulate strings can manipulate programs. Unfortunately, the syntax of most programming languages makes it relatively difficult to design programs that can effectively manipulate other programs. As we will see later, the FSM, PDA, and Turing machine formalisms that we are going to focus on are reasonably easy to work with. Programs that perform functions on FSMs, PDAs, and Turing machines will be an important part of the theory that we are about to build.

Programs that write other programs play an important role in some application areas, including mathematical modeling of such things as oil wells and financial markets. ℭ 676.

4.4 Exercises

1) Describe in clear English or pseudocode a decision procedure to answer the question, "Given a list of integers N and an individual integer n, is there any element of N that is a factor of n?"

2) Given a Java program p and the input 0, consider the question, "Does p ever output anything?"
   a) Describe a semidecision procedure that answers this question.
   b) Is there an obvious way to turn your answer to part (a) into a decision procedure?

3) Recall the function *chop*(L), defined in Example 4.10. Let $L = \{w \in \{$`a`, `b`$\}^*: w = w^R\}$. What is *chop*(L)?

4) Are the following sets closed under the following operations? Prove your answer. If a set is not closed under the operation, what is its closure under the operation?
   a) $L = \{w \in \{$`a`, `b`$\}^* : w$ ends in `a`$\}$ under the function *odds*, defined on strings as follows: *odds*(s) = the string that is formed by concatenating together all of the odd numbered characters of s. (Start numbering the characters at 1.) For example, *odds*(`ababbbb`) = `aabb`.

b) FIN (the set of finite languages) under the function *oddsL*, defined on languages as follows:

$oddsL(L) = \{w : \exists x \in L \ (w = odds(x))\}$.

c) INF (the set of infinite languages) under the function *oddsL*.

d) FIN under the function *maxstring*, defined in Example 8.22.

e) INF under the function *maxstring*.

5) Let $\Sigma = \{\texttt{a}, \texttt{b}\}$. Let S be the set of all languages over Σ. Let f be a binary function defined as follows:

$f: S \times S \rightarrow S$.

$f(x, y) = x - y$.

Answer each of the following questions and justify your answer:

a) Is f one-to-one?

b) Is f onto?

c) Is f commutative?

6) Describe a program, using *choose*, to:

a) Play Sudoku 💻, described in ℭ 781.

b) Solve Rubik's Cube® 💻.