Foundations of Computing

An Accessible Introduction to Formal Languages

Charles D. Allison

Foundations of Computing

An Accessible Introduction to Formal Languages

Charles D. Allison

This book is for sale at http://leanpub.com/foundationsofcomputing

This version was published on 2023-12-20

Leanpub

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2021 Fresh Sources, Inc.

Cover art, ”Feather Moon”, by Doug Hamilton, doughamiltonart.com.

To my wife, Sandra, for a lifetime of support and joy.

Contents

Foreword . . . . **i**

Preface . . . . **ii**

1. Introduction **1**
- 1.1 Formal Languages 2
- 1.2 Finite State Machines 5
  - Exercises 11
  - Chapter Summary 11

I Regular Languages 13

2. Finite Automata **14**
- Where Are We? 14
- Chapter Objectives 14
- 2.1 Deterministic Finite Automata 15
  - Exercises 22
- 2.2 Non-Deterministic Finite Automata 24
  - Equivalence of NFAs and DFAs 28
  - NFAs and Complements 33
  - Exercises 36
- 2.3 Minimal Automata 37
  - Exercises 44
- 2.4 Machines with Output 44
  - Computer Arithmetic 48
  - Lexical Analysis 51
  - Minimal Mealy Machines 55
  - Exercises 58
  - Chapter Summary 59

3. Regular Expressions and Grammars **60**
- Where Are We? 60

Chapter Objectives . . . 60
3.1 Regular Expressions . . . 61
Exercises . . . 64
3.2 Equivalence of Regular Expressions and Regular Languages . . . 64
From Regular Expression to NFA . . . 65
From NFA to Regular Expression . . . 68
Exercises . . . 72
3.3 Regular Grammars . . . 73
Left-Linear Grammars . . . 77
Exercises . . . 81
Chapter Summary . . . 83

4. Properties of Regular Languages . . . 84
Where Are We? . . . 84
Chapter Objectives . . . 84
4.1 Closure Properties . . . 85
Computing Set Operations . . . 88
Exercises . . . 92
4.2 Decision Algorithms . . . 93
Exercises . . . 100
4.3 Infinite Regular Languages and a "Pumping Theorem" . . . 101
Exercises . . . 110
Chapter Summary . . . 110

II Context-Free Languages . . . 111

5. Pushdown Automata . . . 112
Where Are We? . . . 112
Chapter Objectives . . . 112
5.1 Adding a Stack to Finite Automata . . . 113
Exercises . . . 122
5.2 Pushdown Automata and Determinism . . . 123
Exercise . . . 128
Chapter Summary . . . 128

6. Context-Free Grammars . . . 129
Where Are We? . . . 129
Chapter Objectives . . . 129
6.1 Context-Free Grammars and Derivations . . . 131
Simplifying Grammars . . . 136
Exercises . . . 139
6.2 Derivation Trees and Ambiguous Grammars . . . 141

Operator Precedence 144
Operator Associativity 145
Expression Trees 147
Exercises 150
6.3 Equivalence of PDAs and CFGs 151
From CFG to PDA 151
From PDA to CFG (Special Case) 153
From PDA to CFG (General Case) 155
Exercises 168
Chapter Summary 169

7. Properties of Context-Free Languages 170
Where Are We? 170
Chapter Objectives 170
7.1 Chomsky Normal Form 171
Removing Lambda 171
Removing Unit Productions 175
Chomsky Normal Form Rules 178
Exercises 184
7.2 Closure Properties 185
Closure Properties of DCFLs 189
Exercises 191
7.3 Decision Algorithms 191
Is a CFL Empty or Infinite? 198
Exercises 202
7.4 Infinite CFLs and Another Pumping Theorem 203
Exercises 211
Chapter Summary 211

III Recursively Enumerable Languages 212

8. Turing Machines 213
Where Are We? 213
Chapter Objectives 213
8.1 Prelude 214
Post Machines 215
Exercise 217
8.2 The Standard Turing Machine 217
Subroutines 226
Halting 229
Exercises 231
8.3 Variations on Turing Machines 231

The Universal Turing Machine . . . 234
Non-Deterministic TM = Deterministic TM . . . 237
Programming Exercise . . . 240
Chapter Summary . . . 240

9. The Landscape of Formal Languages . . . 241
Where Are We? . . . 241
Chapter Objectives . . . 241
9.1 Recursively Enumerable Languages . . . 242
A Non-Recursive, RE Language . . . 243
Context-Sensitive Languages . . . 243
Properties of Recursively Enumerable Languages . . . 243
Exercises . . . 245
9.2 Unrestricted Grammars . . . 245
Context-Sensitive Grammars . . . 251
Equivalence of Unrestricted Grammars and Turing Machines . . . 252
Exercises . . . 258
9.3 The Chomsky Hierarchy . . . 259
Countable Sets . . . 260
Uncountable Sets . . . 261
Chapter Summary . . . 262

10. Computability . . . 263
Chapter Objectives . . . 263
10.1 The Halting Problem . . . 264
10.2 Reductions and Undecidability . . . 268
Exercises . . . 272
Chapter Summary . . . 273

Glossary . . . 274

Bibliography . . . 278

Index . . . 279

Foreword

Computers are everywhere these days—from our phones, to our cars, to appliances in our homes. Indeed, computers, and the programs that run on them, influence every aspect of our lives. It's difficult to imagine what the world was like without them—the entire field of computing is less than one hundred years old. The advances in computing are truly mind-boggling.

Yet none of this would have been possible without a sound theoretical foundation. Alan Turing, Alonzo Church, John von Neumann, and others developed the mathematical formalisms that make our modern programming languages possible. All modern programming languages, from popular Python to the venerable C programming language, have their roots therein.

The theory of computation doesn't have to be hard to understand; it has an elegance and simplicity that makes it approachable. In this way, it is much like Einstein's theory of special relativity, embodied by the famous equation, $E = mc^2$. But in both cases, the simplicity of the expression of the theory veils the years of toil required to derive the theory. Fortunately, one can learn the theory without all the hard work of initial derivation. We truly stand on the shoulders of giants.

Many computer science students find this topic inscrutable, however. Here is where this book comes in: it presents the theory in a logical, gentle manner, with applications, making it much easier to understand. As I read a draft of the book, I actually became excited about the presentation of the material. At long last, there is a book that lays out the theory in a way that it should be. And the results speak for themselves. When Professor Chuck Allison began using drafts of this book in his classes, student performance on assignments and exams went up significantly.

Some students may wonder why they should even bother learning the theory of computation. "After all," they point out, "I'm just going to write application code for websites and mobile apps. I don't need all this theory stuff." That's a fair question. I freely admit that in my twenty years of industry programming, I never wrote a Turing machine. But it was a bit surprising how many times I found myself writing little programming languages and programs to interpret and process them. Foundational computing theory gives you the tools that enable you to do things like that. In addition, most of my industry work was in communication systems, where the software consists of extremely complex finite-state machines, which this book covers in some depth.

As you approach this book, you will notice that it starts very simply. Each concept is liberally illustrated with examples and visualizations. As it progresses, the examples gradually become more complex. When you reach the end, you will likely be surprised at the complexity of the examples you will be able to understand. And that is the essence of learning: through study and practice, you can accomplish things you never thought possible. Enjoy the journey.

– *Neil Harrison*, Computer Science Department Chair, Utah Valley University

Preface

> *I am personally convinced that any science progresses as much by the writing of better textbooks as by the generation of new knowledge, because good textbooks are what allows the next generation to learn the older stuff quickly and well so we can move on to interesting new things instead of having to painstakingly decipher the discoveries of our forebears.*
>
> *– Jaap Weel*

I wrote this book to make the fundamentals of the theory of computation more accessible to the current generation of undergraduate computer science students. Over the course of twenty years of teaching this subject at Utah Valley University (UVU) in Orem, Utah, I have noticed that computer science undergraduates are not as mathematically sophisticated as in the early days of computer science education, when CS students mostly came from mathematics and the sciences. Yet the demand for CS graduates has increased dramatically, with enrollments at UVU and elsewhere following suit. Furthermore, the areas where computing applies to daily life have broadened to the point that society can't seem to do without the convenience that computerized devices afford. CS graduates can make a noticeable contribution to the world of computing work without as much mastery of formal mathematics as was required in times past.

Yet it is crucial that a practicing software engineer understand the nature and limits of computation; the computing practitioner must know what software can and can't do. In this book, I attempt to illuminate the fundamentals of computing for as large an audience as possible. The key results of the *theory of computation* are covered with numerous examples, and sometimes even with working Python programs, while at the same time achieving a measure of rigor with only a minimum of mathematical formalism. Formal proofs appear infrequently, but constructive arguments abound. I use recursive definitions, but use mathematical induction only twice. While mathematical notation appears as needed, I use visual representations profusely to illustrate concepts.

I assume that readers have basic programming knowledge as one would encounter in typical CS1 and CS2 courses, and familiarity with introductory discrete mathematics, including sets, relations, functions, modular arithmetic, the notion of a logarithm, first-order predicate logic, and the structure of simple, directed graphs.

The emphasis is on the structure, properties, and application of formal languages, along with an introductory exposure to computability. Complexity theory is not covered, as it is the topic of a separate undergraduate course on algorithms at UVU. Parsing is only touched on since that topic is covered exhaustively in our compiler course required of CS majors.

I use a uniform approach to transition graphs for the different types of finite-state machines so students feel at home when new features are added. For example, for pushdown automata, I use the

same graph notation as for finite automata, only adding stack operations, and only pop one symbol at a time. In addition, when stack start symbols are used, I explicitly push them in the graph, minimizing changes to what students are already accustomed to with finite automata. Furthermore, I require both final/accepting states *and* an empty stack simultaneously for PDA acceptance. This avoids needless discussion of how to convert acceptance by empty stack to and from acceptance by accepting state only, and makes it easier to show that deterministic pushdown automata are closed under complement, as well as making converting PDAs to context-free grammars less exceptional.

Several programming examples appear, and a small number of modest programming projects appear in the exercises. I use Python 3. Python is among the most readable of programming languages; it is often referred to as "executable pseudocode." It is in widespread use in many computing domains and is universally available. An effective innovation is the use of Python programs to teach reductions of one unsolvable problem to another in a compact, introductory chapter on computability.

I wish to acknowledge the helpful comments of reviewers Chad Ackley, Matt Bailey, Tyler Barney, Eric Belliston, Kevin Black, Curtis Boland, Alan Chavez, Josh Chevrier, Thomas Christiansen, Cameron Clay, Ethan Clegg, Brennen Davis, Daniel Dayley, Michael Elliott, Ryan Ferguson, Landon Francis, Phillip Goettman, Joshua Gray, Drake Harper, Quinn Heap, Jace Henwood, Gregory Hodgson, Chad Holt, Preston Hrubes, Trent Hudgens, Jared Jacobsen, Joel Johnson, Josh Jones, Tyson Jones, Jenny Karlsson, Hunter Keating, Ethan Kikuchi, Jared Leishman, Ryan Lever, Caleb MacDonald, Natalie Madsen, Nathan Madsen, Wesley Mangrum, Clifton Mayhew, Juan Medrano, Michael Mickelson, James Miles, LuAnne Mitchell, Kelly Nicholes, Dakota Nielsen, Eric Nielson, Kris Olson, Timothy Marc Owens, Kailee Parkinson, Cody Powell, Jake Prestwich, Zachary Price, Andrew Ripley, Devin Rowley, Drew Royster, Gail Sanders, Vani Satyanarayan, Blaine Sorenson, Isaac Stark, Benjamin Thornhill, Alexander Vaughn, and Jeremy Warren, all UVU alumni. A special thanks to Dr. Keith B. Olson, my long-time colleague and fellow mathematician, for his edits, insights, and encouragement. I also drew inspiration from Dr. Peter J. Downey, who taught me this subject decades ago in CS 473 at the University of Arizona.

I am indebted to Doug Hamilton for the cover art and to Alexa Schulte for assistance in the cover design.

I hope CS students will find this book helpful in understanding what computers can and cannot do, in gaining insights into crafting programs for problems that involve various states or text processing, and in comprehending more fully the nature of computation.

- Chuck Allison, 2016–2022

1. Introduction

There is nothing so practical as a good theory.
– Kurt Lewin

“Charlecote Park Formal Gardens” by Tony Hisgett, licensed under CC BY 2.0

Computer science predates computers. The theoretical foundations of computation were laid long before the first computer was built[1]. What we now call the *theory of computation* was developed in the mid-twentieth century by researchers seeking a systematic method of solving all mathematical problems[2]. Their efforts led to the design of digital computers, and equally importantly, revealed what automatic computation *cannot* do. But first, let’s be clear on exactly what we mean by *computation.*

A *computation* is the execution of a self-contained *procedure* that processes *input*, unaided by outside intervention once it starts, and, if all goes well, eventually halts, yielding *output* that corresponds to the input given. For example, running a compiler on source code as input and receiving an executable file as output is a computation. A computation that always halts is called an *algorithm.* (Note: We will see computations that do not always halt.) The common metaphor for such a procedure is a “black box”, as depicted in Figure 1–1.

Figure 1–1: Computation as a Black Box

[1] Babbage’s Analytical engine of 1837 is considered the first manifestation of a design with the computational power of general-purpose computers. The idea of an algorithm, which is the essence of computer science, is centuries old, and many automated mechanical processes got their start during the first Industrial Revolution. Here we are mainly interested in ideas more abstract, as in software.

[2] It is interesting that their research discovered that no such method exists!

For our purposes, a computation invokes a procedure that represents some underlying mathematical model—a computing mechanism that we call an *abstract machine*[3]. A computation requires a symbolic notation, or *language*, to express it. A *formal language* is a *set of strings* with some inherent structure.

In computer programming, we need to agree on which strings of symbols are valid for describing programs and data. Source code is just a string of symbols fed into a compiler or interpreter and needs to conform to the syntax of the programming language used. Ultimately, input, output, and procedures appear as streams of symbols from some *alphabet*. For many programming languages, the alphabet is the set of Unicode characters. For a computer, it is the two-symbol alphabet $\{0, 1\}$. The fact that we think of the symbols 0 and 1 as numbers is immaterial. We are not concerned in this book about the notion of data types; we just process strings, usually one symbol at a time. With this simple approach, one can grasp all that is possible by automatic computation. Types are abstractions built upon this elementary foundation.

1.1 Formal Languages

A *formal language* is a set of strings formed with symbols from some finite alphabet. In studying formal languages, we are interested in the many ways symbols may be validly combined to form valid strings for a given context. We talk about *tokens* (or *words*), *sentences* (or *phrases*, which are sequences of tokens), and language *structure* (how strings may be properly arranged). We do not concern ourselves here with the semantics of the words—only the structure of the strings.

A **formal language** is a *set of strings* formed with symbols from some *finite alphabet*.

To illustrate, consider the simple alphabet[4] $\Sigma = \{a, b\}$. Strings such as *bab* and *abbaab* are among the infinite number of strings that can be formed with these two symbols. We can also choose *none* of the symbols, resulting in the *empty string* (which has length 0), often represented by the string literal "" in programming languages, but which we denote in this book by the lowercase Greek letter, *lambda*: λ. The set of all possible strings using this alphabet is the infinite set depicted below.

$$\{\lambda, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb, aaaa, ...\}$$

Observe the natural way to enumerate this set, which we call **canonical order**. Strings are grouped according to *length*, the groups are arranged in increasing length, and the strings within each group are *alphabetized* (i.e., they appear in lexicographical order based on the *alphabet* used (*a* comes before *b* etc.).

[3] A theoretical model of a computing system. For example, the model for a simple calculator is the basic operations of arithmetic. The abstract machines we cover in this book are various types of finite automata (explained in section 1.2).

[4] It is conventional to use the capital Greek letter Σ (Sigma) as an identifier for the current alphabet. We will also use the Greek letters λ (lambda), δ (delta) and Γ (uppercase gamma) in this book, as well as ϕ (phi) for the empty set.

This set is formed by considering all possible ways of concatenating any number of *a*'s and *b*'s together. The set of all possible *concatenations* of elements of an arbitrary set of symbols, *S*, is called the *Kleene star*[5] of that set, and is denoted by S^*. Concatenation of strings is a fundamental operation on formal languages. Since a formal language is a set of strings over some alphabet, a language is therefore a subset of its alphabet's Kleene star.

A formal language over the alphabet Σ is a *subset* of Σ^*.

Suppose the variable x represents the string bab and y stands for $abbaab$. The table below uses these variables to illustrate common operations on strings in a formal language.

Table 1–1: Fundamental Language Operations

Operation	Name	Example
$\|x\|$	Length	$\|x\| = 3$
xy	Concatenation	$xy = bababbaab$
x^n	Repetition	$x^3 = babbabbab$, $x^0 = \lambda$
x^*	Kleene Star	$x^* = \{\lambda, bab, babbab, \ldots\}$
x^R	Reversal	$y^R = baabba$

Since languages are sets, the usual set operations such as union, intersection, difference, complement, etc., also apply.

Example 1–1

Suppose we want to form $\{bab, abbaab\}^*$, which is the Kleene star of the strings denoted by x and y above. As always, lambda is the first element of a Kleene star. This is followed by all possible concatenations of the original strings, in canonical order:

$$\{\lambda, bab, abbaab, babbab, abbaabbab, bababbaab, babbabbab, \ldots\}$$

The string bab precedes $abbaab$ in S^* because of its *length*, not because it appears first in the original set (sets, generally, are unordered, but a canonical ordering is by nature an *ordered set*). We use the *alphabet* to determine the lexicographical *ordering* within each length-group.

---

[5]Usually pronounced "KLAY-nee." Also called "Kleene closure" or "star closure". Note that $aa^* = a^*a$ for any symbol *a*. See https://en.wikipedia.org/wiki/Stephen_Cole_Kleene. A Kleene star allows concatenating zero elements of a set, so λ is always present. A shorthand for "one or more concatenations" uses the plus sign as an exponent. In the case of x in Table 1–1, $x^+ = xx^* = x^*x = \{bab, babbab, babbabbab, \ldots\}$.

Example 1–2

Consider the language, L_e, consisting of all even-length strings over the alphabet $\Sigma = \{a, b\}$:

$$L_e = \{\lambda, aa, ab, ba, bb, aaaa, aaab, aaba, aabb, abaa, \cdots\}$$

Another way to describe this language is the Kleene star of the strings from the set $S = \{aa, ab, ba, bb\}$. In other words, $L_e = S^*$. It is obvious that any string in S^* has even length. Take a moment to verify that any even-length string of *a*'s and *b*'s comes from repeated concatenations of elements of *S*.

It is possible to *generate* all the strings in a formal language by using a set of "substitution rules" (aka "rewrite rules") known as a **formal grammar**. A formal grammar consists of substitution rules that stand for substrings[6] that form part of a complete string in a language. A grammar for the language L_e is:

$$E \to aO \mid bO \mid \lambda$$
$$O \to aE \mid bE$$

Since the variable *E* appears first, it is the *start variable*. The vertical bar is an alternation symbol that separates the various ways to substitute for the variable on the left-hand side, so *E* has three substitution rules ($E \to aO, E \to bO, E \to \lambda$) and *O* has two ($O \to aE, O \to bE$). To generate a string, begin by choosing a right-hand side to replace the start variable, *E*, then continue substituting for variables at will, finally ending by choosing a rule containing no variables (in this case, the rule $E \to \lambda$). We can, for example, use this grammar to derive the string *baaa* as follows[7]:

$$E \Rightarrow bO \Rightarrow baE \Rightarrow baaO \Rightarrow baaaE \Rightarrow baaa\lambda \Rightarrow baaa$$

Since lambda is the empty string, it disappears in concatenation. When all we have left are symbols of the alphabet, we are done generating a string.

Another grammar for L_e comes from using the elements of the set *S* from Example 1–1, $\{aa, ab, ba, bb\}$:

$$S \to aaS \mid abS \mid baS \mid bbS \mid \lambda$$

Using this grammar, we can derive the string *baaa* as follows:

$$S \Rightarrow baS \Rightarrow baaaS \Rightarrow baaa$$

---

[6]These intermediate substrings are analogous to phrases of a sentence, so the grammars we study are also known as *phrase-structure grammars*.

[7]In this book we use the single arrow, $\to$, when defining grammar rules, and the double arrow, $\Rightarrow$, when deriving a string using those rules.

Key Terms

computation • abstract machine • algorithm • formal language • alphabet • lambda (λ) • Kleene star • canonical order • formal grammar

1.2 Finite State Machines

We design our computations to only consider one symbol at a time, making decisions based only on the current "state" and the current input symbol. For the language L_e seen earlier, we need only know whether or not the number of symbols consumed at any point is even. Since integers are either even or odd, we have only two states (evenness vs. oddness) to consider. The **transition graph** in Figure 1–2 depicts a 2-state, abstract machine that will solve our problem; we have only to observe whether or not an input string causes the machine to halt in the "even" state, *E*.

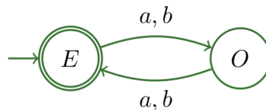


Figure 1–2: A transition graph for a machine that accepts L_e

E is the start state, indicated by an incoming arrow from "out of nowhere." The edges between states *E* and *O* indicate that whenever an *a* or *b* is read, the machine switches state. We combine both *a* and *b* on the same edge instead of having separate edges for each symbol to keep the diagram simple. The double circle for state *E* indicates that it is an *accepting*, or *final* state, meaning that if processing an entire input string ends in that state, then that string is accepted as part of the language. This machine processes the input string *abba* as follows:

Table 1–2: Processing the string *abba*

State	Remaining Input
E	$abba$
O	bba
E	ba
O	a
E	(accept)

Having terminated in state *E*, the string is accepted. A straightforward Python implementation of this machine follows.

Simulating the machine in Figure 1–2

```
def evenab(s):
    state = 'E'      # Start state
    for c in s:      # Process 1 letter at a time
        match state:
            case 'E':
                state = 'O'
            case 'O':
                state = 'E'
    return 'accepted' if state == 'E' else 'rejected'
```

This machine accepts the empty string because the start state is also an accepting state. This makes sense in this case, since zero is an even number.

Whether a string is accepted by a finite state machine depends on ending in an accepting state when the last symbol of the string has been read.

Example 1–3

The following machine accepts all and only those strings over the alphabet $\Sigma = \{0, 1\}$ that end with the symbol `1`.

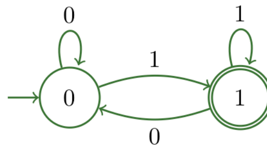


Figure 1–3: Accepts strings ending in 1

Every time a 1 is read, the machine moves to the accepting state. Otherwise it moves to state 0.

Finite state machines, also known as finite **automata** (singular: **automaton**), can give more varied output than just *accept* or *reject*, as illustrated in the next example.

Example 1–4

Removing comments from source code provides an instructive example of an automaton that emits non-trivial output. Suppose a hypothetical language allows comments delimited by a single dollar-sign symbol at each end of a comment's text. An automaton that ignores such comments, but lets uncommented text through as output, appears in Figure 1–4 below.

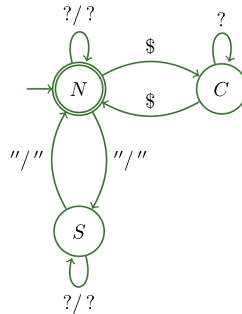


Figure 1-4: State machine to remove $-comments

First, notice that some transition edges have a slash that separates an input symbol from an output symbol; edges without a slash emit no output. (Note: The edges between states *N* and *S* have a double-quote symbol as both input and output; the slash is *not* being quoted.) We use a question mark as a special "wildcard" identifier to represent "the current symbol being processed other than the characters implicitly handled by that state"—in this case, other than a dollar sign or double-quote. The start state, *N*, indicates that we are in "normal" mode, meaning that we are outside of a comment or a string, so we just echo the input symbol to the output stream. When we read a dollar sign outside of a quoted string, we transition to state *C*, indicating that we are now in a comment. We output nothing until after we return to the start state, having read the comment's closing $. The machine operates similarly for quotes and the state *S*, except that everything is printed (quotes included). It is not possible to be in comment mode and string mode simultaneously.

Suppose the string `My $fine $fellow, you owe "$1.50"` is fed into this machine. Execution would then proceed as follows (dashes are used to make spaces visible):

State	Remaining Input	Accumulated Output
N	My-$fine-$fellow,-you-owe-"$1.50"	
N	y-$fine-$fellow,-you-owe-"$1.50"	M
N	-$fine-$fellow,-you-owe-"$1.50"	My
N	$fine-$fellow,-you-owe-"$1.50"	My-
C	fine-$fellow,-you-owe-"$1.50"	My-
C	ine-$fellow,-you-owe-"$1.50"	My-
C	ne-$fellow,-you-owe-"$1.50"	My-
C	e-$fellow,-you-owe-"$1.50"	My-
C	-$fellow,-you-owe-"$1.50"	My-
C	$fellow,-you-owe-"$1.50"	My-
N	fellow,-you-owe-"$1.50"	My-
N	ellow,-you-owe-"$1.50"	My-f
N	llow,-you-owe-"$1.50"	My-fe
N	low,-you-owe-"$1.50"	My-fel
N	ow,-you-owe-"$1.50"	My-fell
N	w,-you-owe-"$1.50"	My-fello
N	,-you-owe-"$1.50"	My-fellow
N	-you-owe-"$1.50"	My-fellow,

State	Remaining Input	Accumulated Output
N	you-owe-“$1.50”	My-fellow,-
N	ou-owe-“$1.50”	My-fellow,-y
N	u-owe-“$1.50”	My-fellow,-yo
N	-owe-“$1.50”	My-fellow,-you
N	owe-“$1.50”	My-fellow,-you-
N	we-“$1.50”	My-fellow,-you-o
N	e-“$1.50”	My-fellow,-you-ow
N	-“$1.50”	My-fellow,-you-owe
N	“$1.50”	My-fellow,-you-owe-
S	$1.50”	My-fellow,-you-owe-“
S	1.50”	My-fellow,-you-owe-“$
S	.50”	My-fellow,-you-owe-“$1
S	50”	My-fellow,-you-owe-“$1.
S	0”	My-fellow,-you-owe-“$1.5
S	”	My-fellow,-you-owe-“$1.50
N		My-fellow,-you-owe-“$1.50”

The operation of this machine is also implemented in the following Python code.

Simulates the machine in Figure 1–4

```python
def dollar(inputstr):
    output = ""
    state = 'N'
    for c in inputstr:
        match state:
            case 'N':
                if c == '$':
                    state = 'C'
                else:
                    output += c
                    if c == '"':
                        state = 'S'
            case 'C':
                if c == '$':
                    state = 'N'
            case 'S':
                output += c
                if c == '"':
                    state = 'N'
    if state != 'N':
        print('Unbalanced delimiters!')
    return output

print(dollar('My $fine $fellow, you owe "$1.50"'))
```

When executed, this program prints `My fellow, you owe "$1.50"`.

Finite automata apply in many real-life situations. Automatic doors in buildings, vending machines, etc., all have some notion of state and of changing state depending on input stimuli over time.

Example 1–5

The machine in Figure 1–5 below illustrates how to score a tennis game[8]. Each time a player wins a volley, the score progresses through the following point-level sequence: *love* (0), 15, 30, 40, *game*. When the score is tied 40–40, it is called a *deuce*. To win the game from deuce, a player must win two volleys in a row. Winning the first volley from deuce gives the player the "advantage", but that player must win the very next volley to win the game. Otherwise play returns to deuce. This could go on for some time (observe the cycles between the Deuce state and the advantage states (*Ad-*) below).

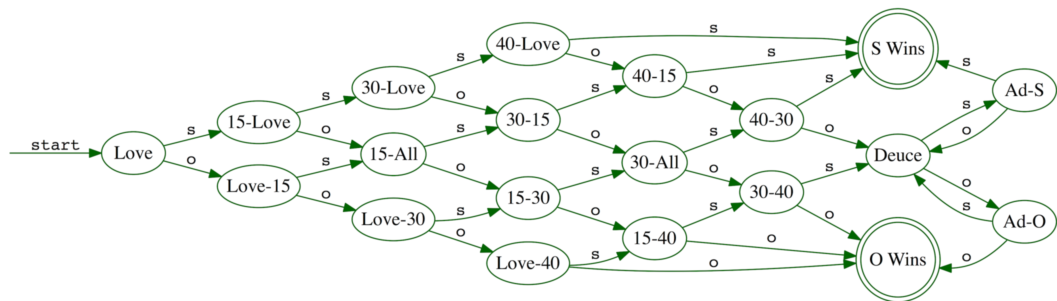


Figure 1–5: Scoring a Game of Tennis

Beginning with the player initially serving, we follow an *s*-edge if the server wins the point or an *o*-edge if the opponent does. There are no out-edges from the states where a player wins, since play ends at that point.

Example 1-6

The diagram in Figure 1-6 models how vending machines from years past (before they had computer chips) kept track of the total amount of money entered. For simplicity, we assume that only nickels, dimes and quarters are accepted and that items cost no more than 50 cents (ignoring any amounts greater than that). The states represent the cumulative total entered at any given point in time.

[8] Diagram based on material from Jeffrey D. Ullman. Used with permission.

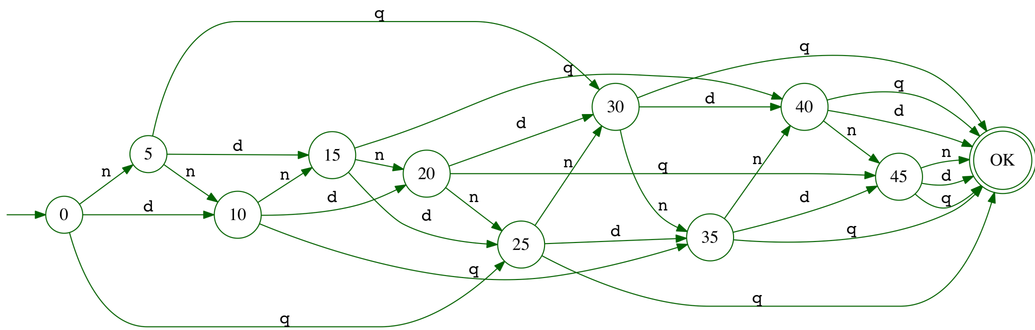


Figure 1–6: Modeling a Vending Machine

Example 1–7

As a final example[9], consider the following schematic of an automatic door.

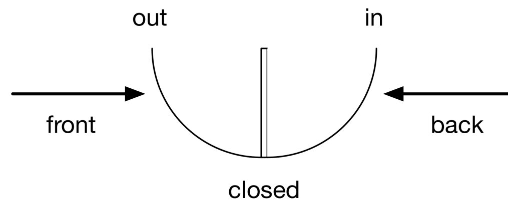


Figure 1–7: A sketch of a 2-way automatic door

This door swings both ways. There are two input stimulus sensors: one when a person walks in from the front (outside), and another when a person exits from the back (inside). If the control mechanism polls for input at small, regular time intervals, there are four possible input configurations over time: **front**, **back**, **both** front and back, and **neither** front nor back. The door can either be opened-*out*, opened-*in*, or *closed*, so these are the three states in our model. The "natural state" is for the door to be closed, so we make that the initial state. We are not testing input strings here, so we have no accepting state. Figure 1-8 below shows an automaton that guarantees that customers can come and go without getting hit by the door. When no one is present within sensor range, the **neither** input signal closes the door.

[9]Example from Sipser, Introduction to the Theory of Computation, 3rd Ed., Cengage, 2013, p. 32.

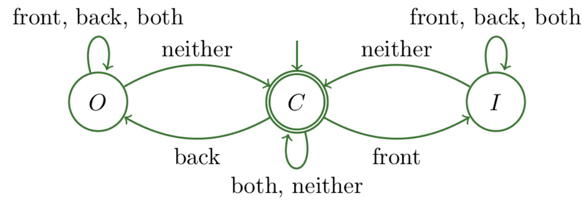


Figure 1–8: Modeling a 2-Way automatic door

Key Terms

state • transition • finite state machine • automaton/automata • transition graph • start/initial state • final/accepting state

---

Exercises

1. List in canonical order the first 11 elements of the subset of $\{0,1\}^*$ that are of *odd length*.
2. List the first 11 elements of $S = \{aba, ba\}^*$ in canonical order. (Note: the alphabet here is $\{a, b\}$, but the set S is being operated upon by the Kleene star.)
3. Draw a finite automaton that accepts all the strings of *odd length* over $\Sigma = \{a, b\}$.
4. Write a formal grammar that generates all strings of *odd length* over $\Sigma = \{a, b\}$.

Chapter Summary

In essence, computation is the meaningful manipulation of symbols. In this book, we relate computations to formal languages, since a computation needs a language to describe its input and output, as well as its operations. Languages can be generated by grammars and recognized by finite-state machines, also called *automata*. The theory of computation comprises three major classes of formal languages along with their associated machines, as shown in the following table.

Language Class	Recognized By	Generated By
Regular	Finite Automata	Regular Grammar Regular Expression
Context-Free	Pushdown Automata	Context-Free Grammar
Recursively Enumerable	Turing Machine	Unrestricted Grammar

As you have seen in this introductory chapter, computation theory is quite practical; it has direct application in digital design, programming languages, compilers—indeed in numerous contexts where an automated process exhibits various states over time. It is no wonder computing has become part of the landscape of our lives. As architect Nicholas Negroponte[10] has said, "Computing is not about computers anymore. It is about living."

Our chief interest is to understand what can be computed in principle, independent of current technology. In Part 3 of this book we will see that the Turing machine is a complete and universal model for automatic computation.

Computation is the meaningful manipulation of *symbols*. Our chief interest is to understand what can be computed *in principle*.

[10]*Being Digital*, 1995.

I Regular Languages

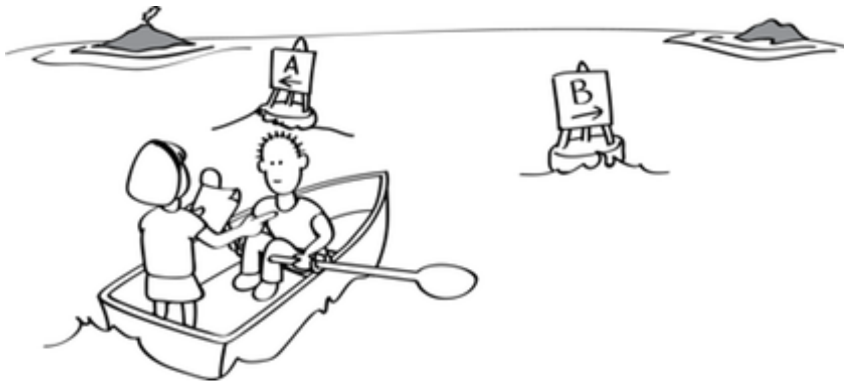


Diagram from csunplugged.com

2. Finite Automata

Vernon Conway: Why is it so difficult for you to accept my orders if you're just a machine?
Blue Robot: Just a machine? That's like saying that you are just an ape.

– *Automata movie (2014)*

"Vintage Pop Machine Innards" by Richard Eriksson, licensed under CC-BY 2.0

Where Are We?

Language Class	Recognized By	Generated By
Regular	**Finite Automata**	Regular Grammar Regular Expression
Context-Free	Pushdown Automata	Context-Free Grammar
Recursively Enumerable	Turing Machine	Unrestricted Grammar

Chapter Objectives

- *Design finite automata to solve simple computational problems*
- *Convert non-deterministic automata to deterministic ones*
- *Minimize the number of states in a deterministic finite automaton*
- *Use finite automata to model computations that produce output strings*

In this chapter, we look at the class of languages that finite automata accept as well as the computations that finite automata can perform. Finite automata are especially useful in text processing and digital circuit design.

2.1 Deterministic Finite Automata

The automata in Chapter 1 are all *deterministic* in that there is no ambiguity about which edge to follow from each state. Deterministic finite automata are also easy to program; simply have a case for each state, and inside each case, choose the target state corresponding to the current input symbol. In the next section, we will see how non-deterministic finite automata can also be useful.

Definition 2.1

A **deterministic finite automaton** (DFA) is a finite state machine consisting of the following:

- Q, a set of **states**, one of which is the *initial* (or *start*) state, and a subset of which are *final* (or *accepting*) states.
- Σ, an **alphabet** of valid input symbols.
- $\delta : Q \times \Sigma \rightarrow Q$, a **function** which, given a state and input symbol, determines the next state of the machine.

Definition 2.1 is merely a mathematical reformulation of what we already understood a DFA to be from Chapter 1. The *transition function*[1], δ, represents the transitions/edges in the machine. The subset of Q comprising the final states can be empty in the case of machines that produce an output string instead of a yes-or-no result. There must be *exactly one* transition out of every state for *each symbol* in the alphabet of the DFA.

There must be *exactly one* transition out of *every* state for *each* symbol in the alphabet of the DFA.

Example 2–1

To illustrate the application of the formal definition of a DFA, consider the DFA in Figure 2–1 below, which accepts all strings over the alphabet $\Sigma = \{a, b\}$ that end with the symbol *b*.

[1] It is customary to use the Greek letter, δ (delta), since it is commonly used in mathematics to denote change, another name for "transition".

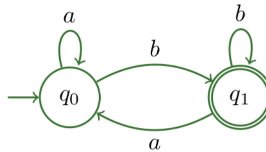


Figure 2–1: DFA accepting strings ending with the symbol *b*

Every time we read a *b*, for all we know it could be the last symbol in the input string, so we move to the accepting state q_1; otherwise we move to state q_0. The formal definition for this DFA is therefore:

- $Q = \{q_0, q_1\}$, where q_0 is the initial state and $\{q_1\}$ is the set of final states
- $\Sigma = \{a, b\}$
- $\delta = \{((q_0, a), q_0), ((q_0, b), q_1), ((q_1, a), q_0), ((q_1, b), q_1)\}$

Yet a third way to depict a DFA is with a *transition table*, as follows:

State	***a***	***b***
$\boldsymbol{q_0}$	q_0	q_1
$+\boldsymbol{q_1}$	q_0	q_1

In a transition table, the initial state appears first and a plus sign indicates each final state.

Since tables are a common data structure in programming languages, they offer yet another approach to implement DFAs in code. The program below uses Python's dictionary data type to simulate the transitions in the table above. The dictionary named `transitions` maps each `<state,symbol>` input pair to a machine state.

```
def ends_with_b(s):
    transitions = {('q0','a'):'q0', ('q0','b'):'q1',
                   ('q1','a'):'q0', ('q1','b'):'q1'}
    state = 'q0'
    for c in s:
        state = transitions[(state,c)]
    return 'accepted' if state == 'q1' else 'rejected'

print(ends_with_b('aabab')) # accepted
print(ends_with_b('aaba'))  # rejected
```

This program is more concise than using `if` and `else` statements to check states directly in code as we did in Chapter 1.

Example 2–2

Now consider how to construct a DFA that accepts all strings over the alphabet $\Sigma = \{a, b\}$ that end with the substring ab. In cases like this it is often helpful to begin by drawing a “partial machine” that accepts only the required substring:

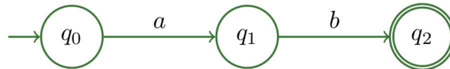


Figure 2–2: Partial DFA accepting strings ending with ab

If we are in state q_0 and read a *b*, we should stay in q_0, since we need an *a* preceding the final *b*. It is only when we read an *a* there that we might have encountered the second-to-the-last symbol in a valid string. State q_1 represents having just read an *a*, so we'll stay there when reading consecutive *a*'s, hence the loops on states q_0 and q_1 in Figure 2–3 below. If we read an *a* in state q_2, it could be the final *a* before the final *b*, so we move back to state q_1. If we encounter a *b* in state q_2, we need to start over by moving to q_0.

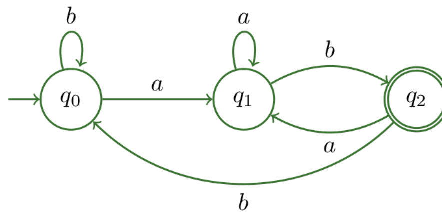


Figure 2–3: Complete DFA accepting strings ending with ab

It is now clear that state q_0 represents having read a b that was *not* preceded by an a.

Example 2–3

To design a DFA that accepts all strings that contain *aba* as a substring, first construct a partial machine that accepts only the string *aba*. (See Figure 2–4 below.)

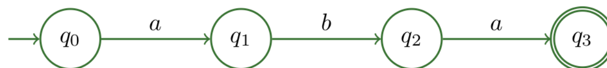


Figure 2–4: Accepts only aba

You can now complete the DFA by filling in the missing edges (see Figure 2–5).

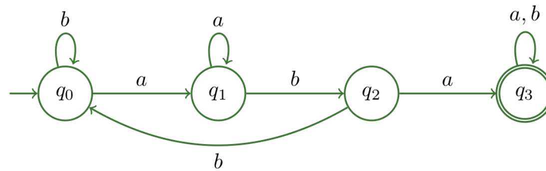


Figure 2–5: Accepts strings containing aba

Example 2–4

Now consider how to define a DFA that accepts the *complement* of the language of Figure 2–5 above (i.e., strings that do *not* contain the substring aba). Once state q_3 is reached, we know that an aba has occurred, so if a string never takes us there, it is in the complement of the language. In general, the complement of language consists of strings that never end in a final state in the original DFA, or, equivalently, that end only in *non-final* states. Therefore, to construct a DFA for the complement of a language, simply **invert the acceptability of each state** in the original DFA. The machine in Figure 2-6 accepts the complement of the language in the previous example.

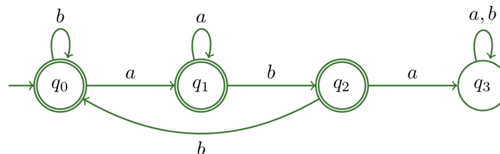


Figure 2–6: Accepts strings not containing aba

There is no going back to an accepting state once a string reaches state q_3. Such a state is called a **dead state** or a **jail state**. Jail states are common in languages that are complements of other languages—in other words, when acceptable strings are defined by properties they *don't* satisfy.

To recognize the *complement* of a language, *invert the acceptability* of each state in a DFA that accepts the original language.

Example 2–5

Another machine that needs a jail state is found in the language of all strings that begin with an *a* and end with a *b*. If the first symbol is a *b*, then there is no hope of the machine accepting it. (See Figure 2–7).

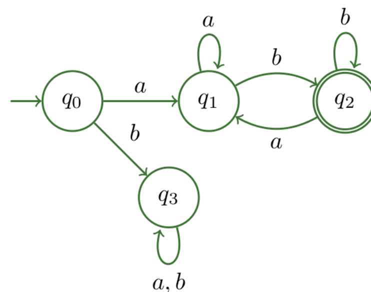


Figure 2–7: A DFA accepting strings starting with a and ending with b

---

Example 2–6

How would we construct a DFA that accepts strings of zeroes and ones where the next-to-last symbol is a 1? Such a string must end with either 10 or 11. Since we never know when the next-to-last input occurs, we need to track those two substrings whenever they occur. See Figure 2–8.

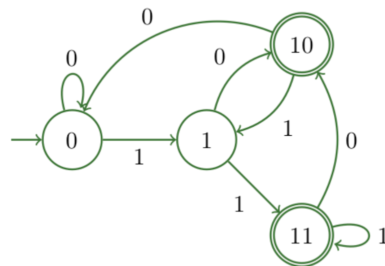


Figure 2–8: Accepts strings with a 1 in the next-to-last position

The state labels reflect the substring just processed that led to each state. Observe how the out-edges from the accepting states behave. The last symbol read reaching those states becomes the next-to-last symbol on the subsequent move.

Example 2–7

Now consider how to recognize strings over $\{0, 1\}$ that end with a 1 in the *third-to-last* position. Reading a 1 starts us on a potential accepting path. From there we will read a 0 or 1, at which point we will have read one of 10 or 11. An acceptable string must be of at least length 3 and end with 100, 101, 110, or 111.

These possibilities are represented by accepting states in the following machine:

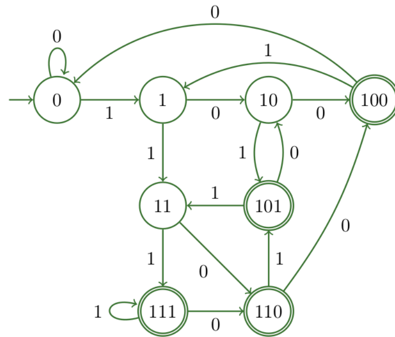


Figure 2–9: A DFA that accepts strings with a 1 in the third-to-last position

Where this machine moves from its accepting states depends on the four possibilities of where the 1's occur in the last two symbols read, which this DFA tracks nicely.

Example 2–8

Consider a machine that reads the bits of a binary number, one at a time, left-to-right, to determine what the modulus (i.e., remainder) of the input number is when divided by 3. We have neither arithmetic instructions nor auxiliary memory—all we have are states—so we must find a way to keep track of the modulus as we go. We need three states, therefore, corresponding to the remainders 0, 1, and 2, with 0 as the initial state (i.e., the number is considered to be zero before any input is consumed). See Figure 2–10.

Figure 2–10: States representing possible remainders mod 3

How do we track the remainder as we read a single bit-symbol at a time? In the simplest instance, we can assume that our current number is 0 to start with. Appending a 0 still gives us $00 = 0$, so $\delta(0, 0) = 0$. Appending a 1 gives the number $01 = 1$, so $\delta(0, 1) = 1$. If instead the current number is 1, the machine is in state-1 (since $1 \equiv 1 \mod 3$), then appending a 0 transforms the number into $10_2 = 2 \equiv 2 \mod 3$, so $\delta(1, 0) = 2$. Similarly, appending a 1 gives $11_2 = 3 \equiv 0 \mod 3$, so $\delta(1, 1) = 0$. If the current number is $2 = 10_2$, appending a 0 yields $100_2 = 4 \equiv 1 \mod 3$, while appending a 1 results in $101_2 = 5 \equiv 2 \mod 3$, so $\delta(2, 0) = 1$ and $\delta(2, 1) = 2$. The following DFA records all of these results.

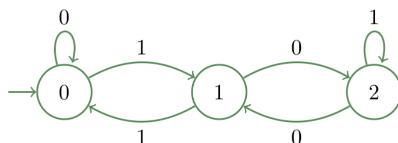


Figure 2–11: DFA whose states identify the residue mod 3 of a binary number

Having established how the machine behaves for the numbers 0, 1, and 2, we now use this information as a base case to show by induction that the machine is correct for any number. The induction hypothesis is that when we are in state-0 that the number is congruent to 0 mod 3, and analogously for states 1 and 2.

To understand how the value of the input number changes as we append the next 0 or 1, consider how to recognize the number 5, say, which appears as 101 in binary. Reading the bits left-to-right, we construct this number a symbol at a time as follows:

1. Start with our number, n, say, initialized to 0, and then read the first 1-bit, “appending” it to our current n. Appending a 1 to a bit string is numerically equivalent to shifting the bits one position to the left (which is the same as multiplying the number by 2), and then adding the 1, giving the number $2 \cdot 0 + 1 = 1$.
2. Next, we read and append the 0-bit, so we shift left (i.e., multiply the running result by 2) and add zero, giving $2 \cdot 1 + 0 = 2 = 10_2$.
3. Finally, we append the last 1-bit to obtain $2 \cdot 2 + 1 = 5 = 101_2$.

To summarize: every time we append a bit, b, we update our number, n, to be $2n + b$, where b is 0 or 1.

We are now able to track the remainders for any bit string. Consider what happens when we are currently in state-0. By the inductive hypothesis, we know that in this state the number must be a multiple of 3, because its remainder is 0, therefore, $n = 3k$ for some k. If we append a 0-bit, then the updated number is $n = 2(3k) + 0 = 6k \equiv 0 \mod 3$, meaning that the number is still a multiple of 3, so we remain in state-0. If instead we append a 1-bit, the number becomes $2(3k) + 1 \equiv 1 \mod 3$, so we move to state-1, as shown in Figure 2–12.

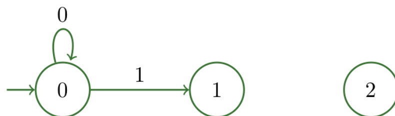


Figure 2–12: Adding edges for state-0

If we are in state-1, then $n \equiv 1 \mod 3 \Rightarrow n = 3k + 1$, for some k, so the results for appending a 0 are $2(3k + 1) + 0 = 6k + 2 = 3(2k) + 2 \equiv 2 \mod 3$, and when appending a 1 we get $2(3k+1)+1 = 6k+3 = 3(2k+1) \equiv 0 \mod 3$. Figure 2–13 updates our working DFA accordingly.

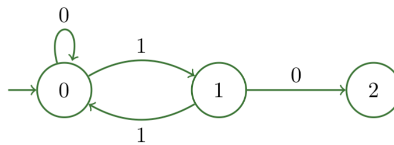


Figure 2–13: Adding edges for state-1

In state-2 we know that $n = 3k+2$, so the results become $2(3k+2)+0 = 6k+4 = 3(2k+1)+1 \equiv 1 \mod 3$ for appending a 0, and appending a 1 gives $2(3k+2)+1 = 6k+5 = 3(2k+1)+2 \equiv 2 \mod 3$. This is consistent with the DFA we first saw in Figure 2–11.

All the languages we have seen so far are examples of what we call *regular languages*, which are simply those languages that are accepted by some DFA.

Definition 2.2

A **regular language** is a formal language for which there exists a deterministic finite automaton that accepts all and only those strings in the language.

Key Terms

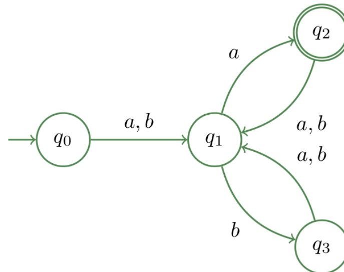
determinism • DFA • transition function • jail state • complement • regular language

Exercises

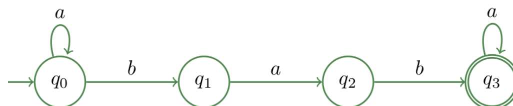
1. Draw a transition graph for the following DFA.

	a	b
q_0	q_0	q_1
q_1	q_0	q_2
$+q_2$	q_1	q_3
q_3	q_2	q_4
q_4	q_3	q_4

2. Draw transition graphs for DFAs that accept the following languages:
   a. Strings over $\Sigma = \{0, 1\}$ that begin with 1 and end with 0.
   b. Strings over $\Sigma = \{a, b\}$ where the substring aa occurs at least twice.
   c. Strings over $\Sigma = \{a, b, c\}$ where every b is followed immediately by at least one c. Any string of a's and c's is accepted if there are no b's at all in the string.
3. Describe in English the language accepted by the following DFA.



4. The following "incomplete" DFA accepts all strings matching the pattern a^*baba^*, but there are edges missing. Complete the DFA by adding a jail state and the missing edges.



5. Draw a DFA for the language of all strings over $\Sigma = \{a, b\}$ that do *not* have two consecutive b's.
6. Define the transition function, δ, for the DFA in the previous problem.
7. Draw a DFA for the language of all strings over $\Sigma = \{a, b\}$ where no two adjacent symbols are the same. For example, bab and $ababa$ are in the language, but aa and abb are not.
8. Draw a DFA for the language of all strings over $\Sigma = \{a, b\}$ that have an even number of a's and an even number of b's. (*Hint*: You only need four states.)
9. Following Example 2–8, draw a DFA that accepts all bit strings that represent numbers congruent to 3 mod 4.
10. Draw a DFA that accepts all bit strings that represent numbers congruent to 3 mod 5

Programming Exercise

1. Implement the DFA in Figure 2–9. Be sure to reflect states and transitions in your code. Test it with two strings—one that is accepted and one that is rejected.

2.2 Non-Deterministic Finite Automata

The DFA in Figure 2–9 took a lot of effort to design. It is possible, however, to more succinctly express the idea of “ends with a 1 in the third-to-last position” than a DFA allows. With non-determinism, we can get right to the point of expressing exactly what we want, as shown in the *non-deterministic* finite automaton (NFA) in Figure 2–14 below.

Example 2–9

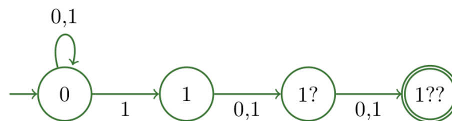


Figure 2–14: An NFA that accepts the language of Figure 2–9

Two things stand out in the transition graph above. First, there are *two choices* of transitions from the initial state for an input of the symbol 1; we may either stay in state-0 or move to state-1. This is what makes this machine non-deterministic. Second, there are *no out-edges* exiting the accepting state. This is because we want to end there. For a string to end in the accepting state, we must choose the edge from state-0 to state-1 at just the right moment. If such a choice can be made, we say that the NFA accepts the string. An accepting path for the input string 10110 is the sequence of states $0, 0, 0, 1, 1?, 1??$. Clearly, only strings with a 1 in the correct position can end in final state. The important thing here is how much easier it is to conceive and express an NFA for this language than a DFA.

This may seem a strange approach to designing finite automata, and programming such a machine may appear to require some sort of backtracking procedure, in case we don’t make the right choice (because we don’t know ahead of time when an input symbol is the last one in the string). Alas, we can’t “back up” in an input string; all decisions must be made based only on the current state and input symbol. Fortunately, it is possible to convert any NFA to a DFA by keeping track of all possible moves as we go. We will show this procedure shortly, but first, let’s see some more NFAs.

Example 2–10

The NFA in Figure 2–15 accepts strings of length two or greater that begin and end with the same symbol.

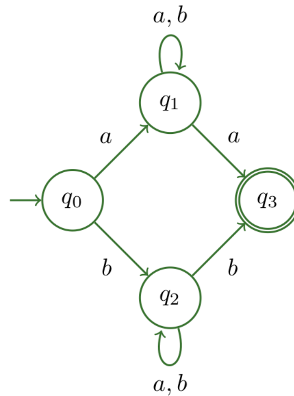


Figure 2–15: NFA accepting strings that begin and end with the same symbol

The non-determinism occurs on the middle states where there are two choices for a and b, respectively. The path to accept the string $bbab$ is q_0, q_2, q_2, q_2, q_3.

It is often easier to use a NFA rather than a DFA to design a finite automaton for a regular language.

Example 2–11

The NFA in Figure 2–16 accepts strings that either contain the substring aa or the substring bb:

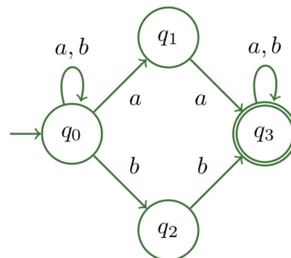


Figure 2–16: NFA that accepts strings containing *aa* or *bb*

It doesn't matter if the string contains both aa and bb. Either path will do in that case. There are multiple accepting paths for the string $baaababba$ for example:

$$q_0, q_0, q_0, q_1, q_3, q_3, q_3, q_3, q_3, q_3$$
$$q_0, q_0, q_1, q_3, q_3, q_3, q_3, q_3, q_3, q_3$$
$$q_0, q_0, q_0, q_0, q_0, q_0, q_0, q_2, q_3, q_3$$

to name only three. (The initial q_0 is where we start before reading input.)

Example 2–12

NFAs have one more interesting feature: they can move from one state to another "on a whim"—in other words, without consuming any input at all. We indicate this by a *lambda transition*. See if you can guess what language the NFA in Figure 2–17 accepts before reading further.

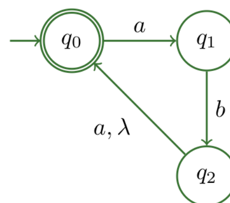


Figure 2–17: What language does this NFA accept?

A lambda transition is basically a "free ride"; we can take it whenever we enter its "from-state" without consuming any further input. That means that for the NFA above, we have a cycle starting and ending with state q_0, where on each round trip we must encounter the substring ab, *optionally* followed by another a. Strings accepted by this NFA include ab,aba,$ababa$,$abababa$, and $abababaab$. We can also represent this language by the expression $\{ab, aba\}^*$, using the Kleene star operator introduced in Chapter 1. In other words, we choose either ab or aba at will until we decide to stop, or we choose nothing at all and obtain the empty string. Both $ababa$ and $abaaba$ can be accepted by the path $q_0, q_1, q_2, q_0, q_1, q_2$, depending on whether or not we consume a when moving from q_2 to q_0.

Example 2–13

Now suppose our alphabet is $\Sigma = \{a, b, c\}$ and we want to accept strings that match the pattern $a^*ac^*b^*$. That is, we need one or more a's followed by zero or more c's followed by zero or more b's. The NFA in Figure 2–18 recognizes this language.

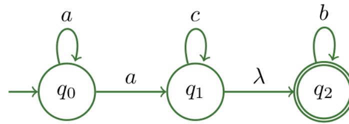


Figure 2–18: NFA accepting $a^*ac^*b^*$

Observe how we represent a Kleene star of a single symbol as a *self-loop* on a state. Also, we need a single edge that is *not* a loop for the required a, because self-loops on states are *optional* moves. The lambda transition above forces the b's to follow the c's while keeping both optional.

We are now able to give a formal definition of a non-deterministic finite automaton.

Definition 2.3

A **non-deterministic finite automaton** (NFA) is a finite state machine consisting of the following:

- Q, a set of **states**, one of which is the initial (or start) state, and a subset of which are final (or accepting) states.
- Σ, an **alphabet** of valid input symbols.
- $\delta : Q \times (\Sigma \cup \lambda) \to 2^Q$, a **function** which, given a state and an input symbol (which could be λ), determines the next possible **set of states** of the machine to move to (i.e., a *choice* of states).

The differences between DFAs and NFAs occur in the definition of δ, namely:

1. λ is a valid "input" (i.e., we can move without consuming any input).
2. More than one state may be reached by the same input symbol.
3. It is possible that no transitions exist at all for a particular input (an *implicit jail*).

The notation 2^Q, sometimes written $\mathcal{P}(Q)$, represents the **powerset**[2] of the set of states, Q. The key feature of NFAs is that there may be zero or more out-edges for any input symbol exiting any state, hence the output is a subset of Q. Also, a DFA is just a special case of a NFA, since machines are *not required* to have any lambda transitions or multiple out-edges from any state for the same symbol. NFAs just give us more convenience in designing automata for regular languages.

[2]The powerset of a set is the set of all possible subsets of that set. If $|Q|$ represents the cardinality of the set Q, then the cardinality of the power set is $2^{|Q|}$, hence the exponential notation for powersets. For example, $Q = \{q_0, q_1\} \Rightarrow 2^Q = \{\phi, \{q_0\}, \{q_1\}, \{q_0, q_1\}\}$, and $|2^Q| = 2^{|Q|} = 2^2 = 4$.

When moving from one state to another in a NFA, remember to take into account **lambda transitions** emanating from the newly entered state.

Equivalence of NFAs and DFAs

We now show how to convert the NFA in Figure 2–15 to a DFA. The key idea is that for each state we have a (possibly empty) set of states to track for every input symbol. The tree in Figure 2–19 illustrates how things can progress as we begin with the start state of the original NFA, q_0, and track all possible moves while reading the first three symbols of any input string. (Going three levels deep suffices in this case to observe all the possible states for this language.) Multiple transitions leaving a state with the same symbol appear in horizontally adjacent boxes below.

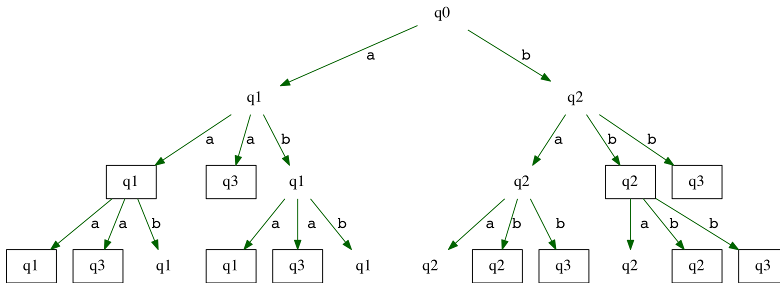


Figure 2–19: Tracking multiple out-edges simultaneously

The symbol a can reach either q_1 or q_3 from state q_1. We will combine these two states into a single, "composite" state, $\{q_1, q_3\}$. The same logic applies to states q_2 and q_3 when leaving state q_2 with a b, giving us the composite state $\{q_2, q_3\}$. See Figure 2–20.

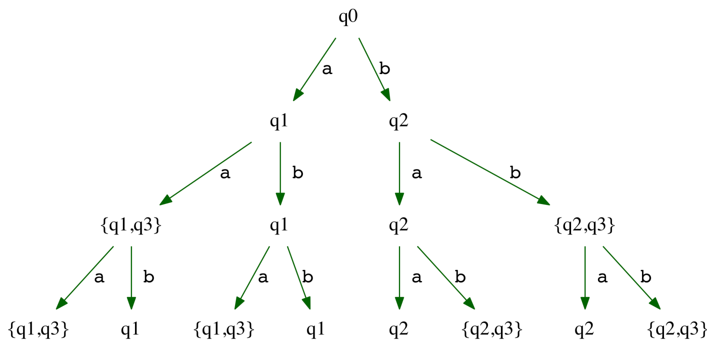


Figure 2–20: Combining like transitions into composite states

This tree suggests that five possible states suffice to process all possible movements: $q_0, q_1, q_2, \{q_1, q_3\}$, and $\{q_2, q_3\}$, giving the following equivalent DFA.

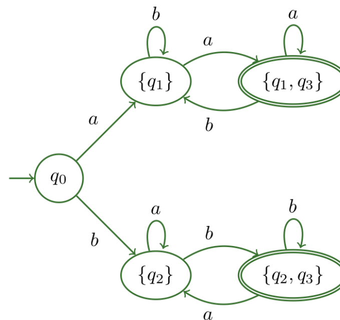


Figure 2–21: DFA for beginning and ending with the same symbol

Whenever a composite state contains any accepting state from the original NFA, that state becomes an accepting state in the resulting DFA. Thus, both states containing q_3 in the figure above are accepting states.

It is usually easier to track multiple moves in a NFA with a transition table instead of a tree. The following is a transition table for the NFA in Figure 2–15.

State	a	b
q_0	q_1	q_2
q_1	$\{q_1, q_3\}$	q_1
q_2	q_2	$\{q_2, q_3\}$
$+q_3$	ϕ	ϕ

Table 2–2: The NFA in Figure 2-15 as a Transition Table

The sets representing the possible output states from q_1 with an a and from q_2 with a b. There are no valid moves from state q_3, indicated by the empty set.

To begin the conversion process to a DFA, we populate a *new* transition table, listing only the start state at first. We refer to Table 2-2 above to determine where the NFA takes us from state q_0 with each symbol of the alphabet, as shown in Table 2–3.

State	a	b
q_0	$\{q_1\}$	$\{q_2\}$

Table 2–3: Starting the NFA-to-DFA conversion process

The NFA reached states q_1 and q_2, so we add rows for these to see where they in turn take us with the possible input symbols:

State	a	b
q_0	$\{q_1\}$	$\{q_2\}$
$\{q_1\}$		
$\{q_2\}$		

Table 2–4: Adding the two newly reached states

As we track where we can move from these states, we update our working table:

State	a	b
q_0	$\{q_1\}$	$\{q_2\}$
$\{q_1\}$	$\{q_1, q_3\}$	$\{q_1\}$
$\{q_2\}$	$\{q_2\}$	$\{q_2, q_3\}$

Table 2–5: Tracking the outputs from states q_1 and q_2

We have obtained two new "states," $\{q_1, q_3\}$ and $\{q_2, q_3\}$, which we process in Table 2–6 below.

State	a	b
q_0	$\{q_1\}$	$\{q_2\}$
$\{q_1\}$	$\{q_1, q_3\}$	$\{q_1\}$
$\{q_2\}$	$\{q_2\}$	$\{q_2, q_3\}$
$+\{q_1, q_3\}$	$\{q_1, q_3\}$	$\{q_1\}$
$+\{q_2, q_3\}$	$\{q_2\}$	$\{q_2, q_3\}$

Table 2–6: The final DFA

No new states have appeared in the body of our table at this point, so we are done. Having tracked all possible movements explicitly, there is no ambiguity of moves using the new set of (possibly composite) states, so the result is *deterministic*, and is equivalent to the transition diagram in Figure 2–21. The technique we have just illustrated is known as *subset construction*[3] (because our destination "states" can represent subsets of the original set of states).

Example 2–14

Let's now convert the NFA in Figure 2–18 to a DFA, referring to its transition table below.

State	a	b	c
q_0	$\{q_0, q_1, q_2\}$	ϕ	ϕ
q_1	ϕ	ϕ	$\{q_1, q_2\}$
$+q_2$	ϕ	$\{q_2\}$	ϕ

Table 2–7: Transition table for the NFA in Figure 2–18

As explained earlier, the output transitions for each state yield a subset of $\{q_0, q_1, q_2\}$. Because of the lambda transition, there is a free ride to q_2 whenever q_1 is reached. The set of states immediately reachable upon entering a state is called its **lambda closure**, which includes the state itself along with

[3] It is also known as the Rabin-Scott *powerset construction*, named after the authors of a 1959 paper.

all states reachable from that state by one or more lambda-transitions (so the lambda closure of q_1 in Figure 2–19 is the set $\{q_1, q_2\}$). Whenever you enter a state, you automatically reach the other states in its lambda closure without consuming a symbol. For this example therefore, q_1, **will never appear** as an output state inside the body of a working transition table **without** q_2.

Definition 2.4

The **lambda closure** of a state is the set containing the state itself along with all other states reachable from that state via one or more consecutive lambda transitions (consuming no input).

As before, we begin the conversion to a DFA by constructing a transition table with only the start state of the original NFA[4], and see where it takes us with each input symbol. (See Table 2–8.)

State	*a*	*b*	*c*
q_0	$\{q_0, q_1, q_2\}$	ϕ	ϕ

Table 2–8: First step in the NFA-to-DFA conversion

As expected, with an a the machine can reach q_0, q_1 and q_2 (the latter because of the lambda edge from q_1 to q_2) from state q_0, so we track these three states together as a single, composite state. We will add this to our running list of states. The state ϕ in this working table represents the fact that there are no edges for the corresponding symbols and states in the original machine, and will become a jail state in the resulting DFA. We won't bother to add a row for a jail state since we know that jails just loop on every symbol.

To fill in the second row, we must track the behavior of all three states inside the composite state simultaneously. From there, an a again reaches the composite state $\{q_0, q_1, q_2\}$. With a c as input, however, state q_1 moves to itself, and, because of the free ride, to q_2 also. (See Table 2–9)

State	*a*	*b*	*c*
q_0	$\{q_0, q_1, q_2\}$	ϕ	ϕ
$\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_2\}$	$\{q_1, q_2\}$

Table 2–9: Processing the new composite state $\{q0, q1, q2\}$

We have reached two new states, so we will add two more rows to track them. See Table 2–10.

[4]Because the start state may have outgoing lambda transitions, the start state for the DFA may be a *set* of states. That is not the case here, but the first step in this construction is always to use the lambda closure of the NFA's start state as the start state of its DFA.

State	a	b	c
q_0	$\{q_0, q_1, q_2\}$	ϕ	ϕ
$+\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_2\}$	$\{q_1, q_2\}$
$+\{q_2\}$	ϕ	$\{q_2\}$	ϕ
$+\{q_1, q_2\}$	ϕ	$\{q_2\}$	$\{q_1, q_2\}$

Table 2–10: Complete Transition Table for the DFA.

No new states have appeared, so we finish by indicating the accepting states. The transition diagram for the complete DFA appears in Figure 2–22. Note the jail state.

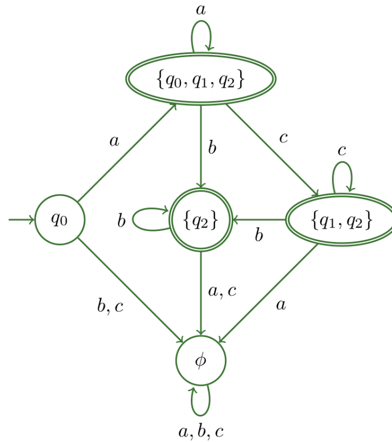


Figure 2–22: The complete DFA diagram after the conversion.

Initial states can have lambda transitions, so remember to make the initial state of the new DFA the lambda closure of the initial state of the NFA when converting an NFA to a DFA.

The following steps summarize the construction of a DFA from a NFA:

1. Initiate a working transition table for the resulting DFA with the lambda closure of the start state of the NFA as the start state of the DFA.
2. For each unprocessed (possibly composite) state, *S*, in the list of states in the left column:
   - For each state, *s*, in *S*:
   - For each symbol, *c*, in the alphabet:
   - For each state, *q*, reachable from *s* by *c*:
   - Add the states of the lambda closure of *q* to the cell in row *S* and column *c*.

3. For each new (possibly composite) state thus obtained, add it to the list of states to process in the left column; go to step 2 unless no unprocessed (possibly composite) state remains in the left column.
4. All resulting states containing any accepting state from the original NFA are accepting states in the resulting DFA.

Since any NFA has an equivalent DFA accepting the same language, both types of automata characterize the class of regular languages.

NFAs and Complements

Recall that the complement of a regular language can be recognized by a machine obtained by inverting the acceptability of each state in a DFA that recognizes the language. The same strategy does not apply, however, to finding the complement of a regular language using a NFA. Why not?

One reason is because NFAs often omit possible moves for input strings, since they represent only what is acceptable in the language. When complementing a language, we are interested in what is not acceptable in the original language, so starting with a NFA won't work.

Example 2–15

Consider the language $\{a, bab\}^*$, which the following NFA accepts.

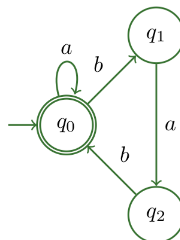


Figure 2–23: NFA for $(a, bab)^*$

Were we to attempt to complement this machine by only reversing the acceptability of the states above, we would get the NFA in Figure 2-24:

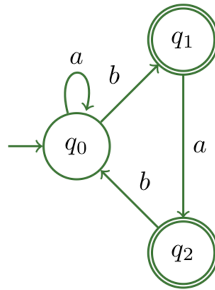


Figure 2–24: Inverting the acceptability of states in Figure 2–23

This NFA does not accept the string *bb*, which is certainly in the complement of $\{a, bab\}^*$. To recognize the complement, we must start with a **DFA** that accepts the language, which in this case requires only adding a jail state for the missing moves. (See Figure 2–25.)

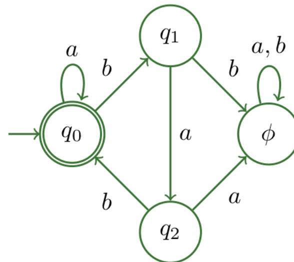


Figure 2–25: DFA for $(a, bab)^*$

We can now invert each state's acceptability to get a DFA for the complement of the language. You can see that the DFA in Figure 2–26 accepts the string *bb*.

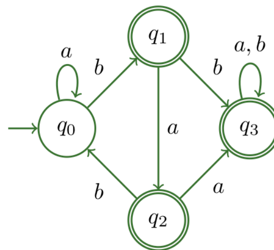


Figure 2–26: DFA for complement of $(a, bab)^*$

Example 2–16

Even if a NFA does not omit any possible moves, inverting its state acceptability can still fail to give the correct complement. Consider the NFA in Figure 2–27, which recognizes the language aa^*b^*.

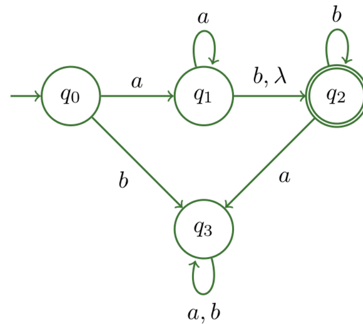


Figure 2–27: NFA accepting aa^*b^*

This NFA is not missing any out-edges, but it has a lambda edge from state q_1 to state q_2. It accepts the singleton string a, as it should. Now look at the NFA that results from inverting the acceptability of each state in Figure 2–28.

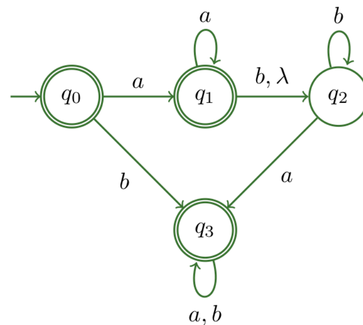


Figure 2–28: Inverting Figure 2–27

This NFA also accepts the string a, so it is not the complement after all.

To find the complement of a regular language, start with a **DFA**.

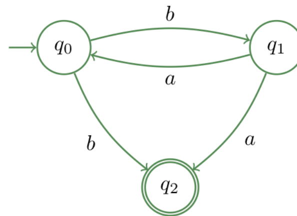
Key Terms

non-determinism • lambda transition • lambda closure • composite state • subset construction

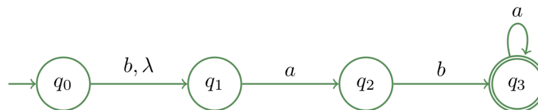
Exercises

1. List all possible paths in the automaton in Figure 2–16 (Example 2–11) that accept the string $abbbaa^*$.
2. Draw a NFA for the language of all strings over $\Sigma = \{a, b\}$ that begin and end with the same symbol, (similar to example 2–10), but without the restriction that the string is at least length 2—that is, the strings λ, a, b should also be accepted.

   The following NFA pertains to problems 3–5.



3. What language does this NFA represent?
4. Convert this NFA to a DFA using the tabular method shown in this section.
5. Draw the transition graph for the DFA from the previous problem.
6. Draw a NFA that accepts all strings over $\Sigma = \{a, b\}$ that either end in *ab* or contain the substring *bb*.
7. Convert the NFA in the previous exercise to a DFA. Draw the transition diagram.
8. Draw a NFA that accepts all strings over $\Sigma = \{a, b\}$ that have both *aa* and *bb* as substrings.
9. Convert the following NFA to a DFA. Draw the transition diagram.



10. Draw a NFA that accepts the language ab^*, (i.e., the language $\{a, ab, abb, abbb, ...\}$). Then find its complement and draw its transition graph.

2.3 Minimal Automata

DFAs are useful in recognizing text patterns and they are easily implemented in code. We haven't considered, however, whether an automaton is "efficient." Our measure of efficiency is the *number of states*, since that determines the number of possible transitions that exist.

Example 2–17

Examine the following DFA for any redundant states.

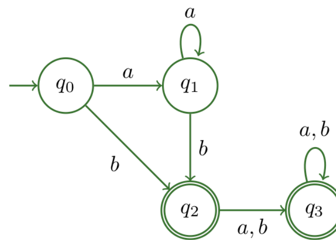


Figure 2–29: Is this DFA "efficient"?

Once we reach state q_2, the machine remains in an accepting state regardless of subsequent input. States q_2 and q_3 can therefore collapse into a single, accepting state, as shown in the next figure.

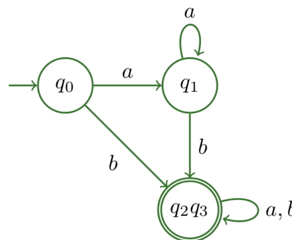


Figure 2–30: Combining states q_2 and q_3

Now think about what language this machine accepts. If a string starts with a *b*, it is accepted. If it starts with an *a*, it is not accepted until a *b* is read. In other words, this machine accepts the language of strings that have at least one *b*. We can therefore combine states q_0 and q_1 to obtain the minimal DFA in Figure 2–31.

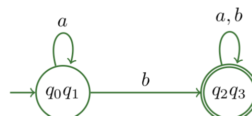


Figure 2–31: A minimal DFA for the language accepted in Figure 2–29

We have discovered that states q_0 and q_1 play the same role in the original DFA. States q_2 and q_3 are likewise indistinguishable in their function, and we have reduced the number of states from 4 to 2.

Example 2–18

Which states are indistinguishable in the following DFA?

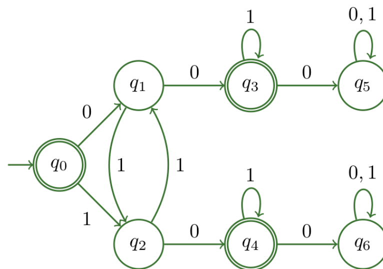


Figure 2–32: Another inefficient DFA

First, note that there are two jails: states q_5 and q_6. There is never a need for more than one jail state, so we combine those. A little thought reveals that q_3 and q_4 behave identically, and q_1 and q_2 as well. The minimal DFA appears in Figure 2–33 below.

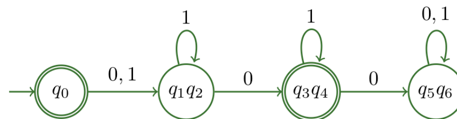


Figure 2–33: Minimal DFA for Figure 2–32

Not all redundancies in DFAs are so easily recognized. We need an **algorithm** to determine when states are indistinguishable in the role they play in a state machine. Our approach is to examine all possible pairings of states for "distinguishability". When we are finished, we can combine into a single state those states that have not been distinguished from each other.

We know that final states clearly play a different role than non-final states, since they give different "output" (accept vs. reject), so we first mark all final/non-final pairs of states as *distinguishable*. It is convenient to use a tabular approach to track the process. We will use the DFA in Figure 2–29 for illustration. Examine the structure of the following table.

	q_1	q_2	q_3
q_0		✔	✔
q_1		✔	✔
q_2			

Table 2–11: Tracking distinguishable states in Figure 2–29

The six unshaded cells in the upper triangle in the body of the table correspond to the six possible pairings of states in the DFA: (q_0, q_1), (q_0, q_2), (q_0, q_3), (q_1, q_2), (q_1, q_3) and (q_2, q_3). We have placed check marks in the cells that pair a final state with a non-final state, since we know that they are distinguishable from the get-go. This is the initial configuration for the minimization process.

We now visit the two remaining unchecked cells to see if we can distinguish them or not. We define distinguishability of states recursively as follows.

Definition 2.5

Two DFA states, q_1 and q_2, are **distinguishable** if:

- **exactly one** of the states is an *accepting state*, or
- The states $\delta(q_1, c)$ and $\delta(q_2, c)$ are *distinguishable* for *at least one symbol* $c \in \Sigma$

In other words, two states are distinguishable if they have different acceptability or if they move with the same symbol to states which have been previously found to be distinguishable (so the minimization algorithm will be iterative). Distinguishable states transition to semantically different subsequent states with the *same input symbol*.

We can now check (q_0, q_1) and (q_2, q_3) for distinguishability. (We already know from Example 2–17 that neither pair is distinguishable, but let's illustrate the algorithm, which we will formalize shortly.) For (q_0, q_1), we have, referring to Figure 2-29:

$$\delta(q_0, a) = q_1, \delta(q_1, a) = q_1$$
$$\delta(q_0, b) = q_2, \delta(q_1, b) = q_2$$

The output states for each input symbol state are identical, so they are certainly indistinguishable. Looking at (q_2, q_3) we find the following:

$$\delta(q_2, a) = q_3, \delta(q_3, a) = q_3$$
$$\delta(q_3, b) = q_3, \delta(q_3, b) = q_3$$

Both inputs move to q_3, so these states can also be combined, as we surmised when we arrived at the minimal DFA in Figure 2–31.

Example 2–19

Now let's use the process to minimize the DFA in Figure 2–32. We first initialize the table, marking pairs of final vs. non-final states distinguishable.

	q_1	q_2	q_3	q_4	q_5	q_6
q_0	✔	✔			✔	✔
q_1			✔	✔		
q_2			✔	✔		
q_3					✔	✔
q_4					✔	✔
q_5						

Table 2–12: Initial table for minimizing Figure 2–32

We have nine pairs corresponding to the empty cells above to test for distinguishability:

(q_0, q_3):
$\delta(q_0, 0) = q_1,\ \delta(q_3, 0) = q_5$
$\delta(q_0, 1) = q_2,\ \delta(q_3, 1) = q_3$ ✔
(q_0, q_4):
$\delta(q_0, 0) = q_1,\ \delta(q_4, 0) = q_6$
$\delta(q_0, 1) = q_2,\ \delta(q_4, 1) = q_4$ ✔
(q_1, q_2):
$\delta(q_1, 0) = q_3,\ \delta(q_2, 0) = q_4$
$\delta(q_1, 1) = q_2,\ \delta(q_2, 1) = q_1$
(q_1, q_5):
$\delta(q_1, 0) = q_3,\ \delta(q_5, 0) = q_5$ ✔
(q_1, q_6):
$\delta(q_1, 0) = q_3,\ \delta(q_6, 0) = q_6$ ✔
(q_2, q_5):
$\delta(q_2, 0) = q_4,\ \delta(q_5, 0) = q_5$ ✔
(q_2, q_6):
$\delta(q_2, 0) = q_4,\ \delta(q_6, 0) = q_6$ ✔
(q_3, q_4):
$\delta(q_3, 0) = q_5,\ \delta(q_4, 0) = q_6$
$\delta(q_3, 1) = q_3,\ \delta(q_4, 1) = q_4$
(q_5, q_6):
$\delta(q_5, 0) = q_5,\ \delta(q_6, 0) = q_6$
$\delta(q_5, 1) = q_5,\ \delta(q_6, 1) = q_6$

When we find moves that distinguish two states, we don't have to check any remaining symbols for that pair (as with the last four check marks above). During this round, we found six new pairs of distinguishable states, so we update our table as follows.

	q_1	q_2	q_3	q_4	q_5	q_6
q_0	✔	✔	✔	✔	✔	✔
q_1			✔	✔	✔	✔
q_2			✔	✔	✔	✔
q_3					✔	✔
q_4					✔	✔
q_5						

Table 2-13: Round 2 of checking for distinguishable pairs

Since we found new check marks, we must process the remaining empty cells again, in case the newfound distinguishable pairs lead to yet other ones. Doing so, however, we find that no new changes occur, so we now combine (q_1, q_2), (q_3, q_4), and (q_5, q_6) to obtain Figure 2–33.

We can now describe the minimization algorithm as follows:

1. For a machine with n states, form a $(n-1) \times (n-1)$ triangular table, listing the first $n-1$ states down the left side as row headers, and the last $n-1$ states horizontally as column headers.
2. Initialize the table by placing a check mark in those cells where **exactly one** of the corresponding states in the pair is a final state.
3. For each remaining empty cell: if the two corresponding states move to distinguishable states on the **same symbol** of the alphabet, place a check mark in that cell.
4. If a new check mark was entered anywhere in step 3, repeat step 3.
5. Combine the states that correspond to any remaining empty cells.

Indistinguishability (but not distinguishability!) is transitive: if the pair (x, y) is indistinguishable, as well as (y, z), then the pair (x, z) is also an indistinguishable pair, and all three states can be combined into one state, $\{x, y, z\}$.

Example 2–20

It was easy to see which states could be combined in the previous two examples. The DFA in the next figure isn't so easy to minimize by cursory inspection.

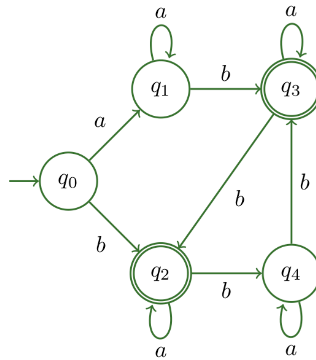


Figure 2–34: Another DFA to minimize

The initial setup for the minimization process follows in Table 2-14.

	q_1	q_2	q_3	q_4
q_0		✔	✔	
q_1		✔	✔	
q_2				✔
q_3				✔

Table 2-14: Initial table for minimizing the DFA in Figure 2–34

We now process each empty cell:

(q_0, q_1):
$$\delta(q_0, a) = q_1,\ \delta(q_1, a) = q_1$$
$$\delta(q_0, b) = q_2,\ \delta(q_1, b) = q_3$$
(q_0, q_4):
$$\delta(q_0, a) = q_1,\ \delta(q_4, a) = q_4$$
$$\delta(q_0, b) = q_2,\ \delta(q_4, b) = q_3$$
(q_1, q_4):
$$\delta(q_1, a) = q_1,\ \delta(q_4, a) = q_4$$
$$\delta(q_1, b) = q_3,\ \delta(q_4, b) = q_3$$
(q_2, q_3):
$$\delta(q_2, a) = q_2,\ \delta(q_3, a) = q_3$$
$$\delta(q_2, b) = q_4,\ \delta(q_3, b) = q_2 \quad ✔$$

On this iteration, we see that states q_2 and q_3 are distinguishable, so we update the table in preparation for another iteration.

	q_1	q_2	q_3	q_4
q_0		✔	✔	
q_1		✔	✔	
q_2			✔	✔
q_3				✔

Table 2-15: Distinguishing states q_2 and q_3

Now we test for other distinguishable pairs based on this new information:

(q_0, q_1):

$\delta(q_0, a) = q_1,\ \delta(q_1, a) = q_1$

$\delta(q_0, b) = q_2,\ \delta(q_1, b) = q_3$ ✔

(q_0, q_4):

$\delta(q_0, a) = q_1,\ \delta(q_4, a) = q_4$

$\delta(q_0, b) = q_2,\ \delta(q_4, b) = q_3$ ✔

(q_1, q_4):

$\delta(q_1, a) = q_1,\ \delta(q_4, a) = q_4$

$\delta(q_1, b) = q_3,\ \delta(q_4, b) = q_3$

This leaves only one indistinguishable pair, (q_1, q_4). A quick check verifies that these states remain indistinguishable, so we combine them in the final result in Figure 2-35.

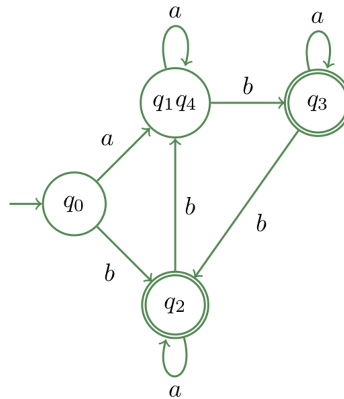


Figure 2–35: Minimal DFA for Figure 2–34

Key Terms

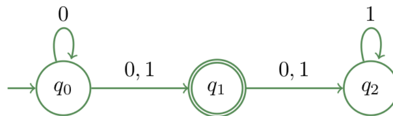
minimal DFA ● distinguishable states ● transitive

Exercises

1. Show that the DFA in Figure 2–9 is minimal.
2. Minimize the DFA defined by the transition function below. The accepting states are $\{q_3, q_4, q_8, q_9\}$.

$$\delta(q_0, a) = q_1, \delta(q_0, b) = q_9$$
$$\delta(q_1, a) = q_8, \delta(q_1, b) = q_2$$
$$\delta(q_2, a) = q_3, \delta(q_2, b) = q_2$$
$$\delta(q_3, a) = q_2, \delta(q_3, b) = q_4$$
$$\delta(q_4, a) = q_5, \delta(q_4, b) = q_8$$
$$\delta(q_5, a) = q_4, \delta(q_5, b) = q_5$$
$$\delta(q_6, a) = q_7, \delta(q_6, b) = q_5$$
$$\delta(q_7, a) = q_6, \delta(q_7, b) = q_5$$
$$\delta(q_8, a) = q_1, \delta(q_8, b) = q_3$$
$$\delta(q_9, a) = q_7, \delta(q_9, b) = q_8$$

3. Find a minimal DFA for the following NFA.



2.4 Machines with Output

Example 1–4 showed a finite automaton that echoed all its input except for the text of dollar-delimited comments. The generic term for such an output-producing machine is *finite-state transducer*,

borrowing the name from electronics (a transducer converts energy from one form to another). Our transducers convert input text to output text. The machine in Figure 2–36 below emits output as it moves from one state to another, using a slash on transitions to separate the input from the output. Such a transducer is also called a Mealy machine[5].

Example 2–1

The following Mealy machine emits a 1 whenever the input symbol changes, and a 0 otherwise.

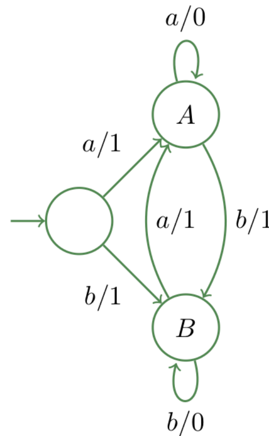


Figure 2–36: A Mealy machine that tracks changes in input

The first symbol always emits a 1, since it follows nothing (which is considered a "change"). Given the input string *aababba* the output is 1011101. The states are aptly named to reflect the most recent symbol read.

Example 2–22

Mealy machines can act as *recognizers* for specific substrings by printing a 1 whenever they encounter them. This has an advantage over a DFA recognizer because it indicates the position of the last symbol of multiple, matching substrings. The machine in Figure 2-37 recognizes the substrings aa and *bb*.

[5]Named after George H. Mealy who introduced them to describe sequential circuits. Another type of machine, named after Edward F. Moore, emits output when a state is entered instead of during a transition. Both types of machines solve the same set of problems; one may be preferable over the other in different situations. We will use only Mealy machines.

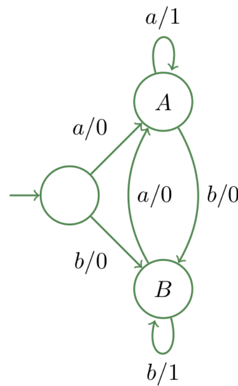


Figure 2-37: Recognizes substrings *aa* or *bb*

The output for the input string $aaabbabbba$ is 0110100110. This machine catches overlapping substrings, hence the initial substring aaa above prints 011.

Example 2–23

A parity bit is a bit that is added at the end of a bit string to indicate whether the number of 1-bits in the original string is even or odd. An *even parity bit* of 0 will be appended if the number of original 1-bits is even; otherwise the even parity bit is 1. Using an even parity bit is a rudimentary error-checking process that always produces an even number of 1-bits in a transmitted string. The machine in the following figure appends an even parity bit to its input.

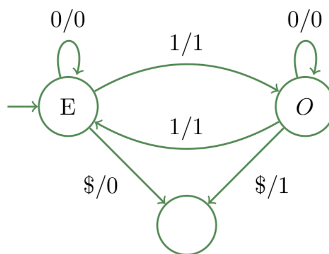


Figure 2–38: An even-parity checker

It is necessary to know when the string ends, so we use a dollar sign to mark the end of the input string.

Example 2–24

The following diagram from Cohen[6], represents a sequential circuit.

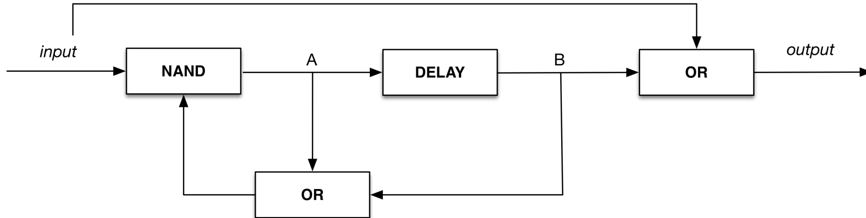


Figure 2–39: A simple sequential circuit

The input bit feeds into both the first NAND and the final OR. At each clock pulse the circuit advances, so the new B becomes what the old A was, the new A becomes the NAND of the input with the OR of the old A and old B, and the OR of the old B and the input is the output for that pulse. The internal state is the pair of values in A and B at the same moment in time, namely, one of $\{(0,0),(0,1),(1,0),(1,1)\}$. We can trace the possible configurations in a transition table. We begin with the start state, and see where it leads (not all possible combinations may appear in some circuits). (See Table 2–17).

State	0	1
q_0: $(\mathbf{0},\mathbf{0})$		

Table 2–17: Start state of the sequential circuit

Now examine what happens in state q_0 (i.e., where $A = B = 0$) and the input is 0. The new B is the old A, which is 0. The new A is $\neg(0 \wedge (0 \vee 0)) = \neg 0 = 1$. The output is $0 \vee 0 = 0$. If the input is 1, then A becomes $\neg(1 \wedge (0 \vee 0))$, which is 1, B is still 0, but the output is 1 this time. The table below updates its first row with these results, using $q_1 = (1,0)$.

State	0	1
q_0: $(\mathbf{0},\mathbf{0})$	(1,0)/0	(1,0)/1
q_1: $(\mathbf{1},\mathbf{0})$		

Table 2–18: First row of Table 2–17 completed

We now seek where $q_1(A = 1, B = 0)$ leads. If the input is 0, the new A becomes $\neg(0 \wedge (1 \vee 0)) = \neg 0 = 1$, as before, and, as always, B becomes the old A, so we go to a new state, $(1,1)$. If the input is 1, the new A is $\neg(1 \wedge (1 \vee 0)) = 0$, and $B = 1$, leading to a new state, $(0,1)$:

[6] Cohen, Daniel, *Introduction to Computer Theory*, Second Edition, Wiley, 1997, p. 162.

State	0	1
$q_0: (0, 0)$	(1,0)/0	(1,0)/1
$q_1: (1, 0)$	(1,1)/0	(0,1)/1
$q_2: (0, 1)$		
$q_3: (1, 1)$		

Table 2–19: Second row completed

If you complete the table, you should get a Mealy machine equivalent to the following diagram.

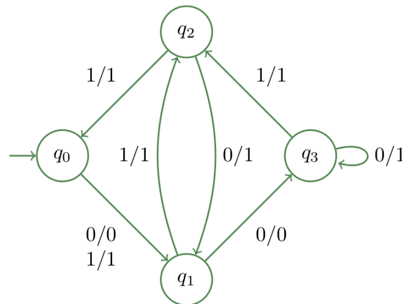


Figure 2–40: Mealy machine for the circuit in Figure 2-40

Computer Arithmetic

Mealy machines naturally model low-level computer arithmetic algorithms. A one's-complement machine, for example, is modeled by the following, trivial machine.

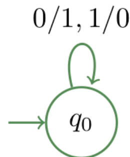


Figure 2–41: A one's-complement machine

Example 2–25

Adding two binary numbers is also "Mealy-able," but requires reading corresponding bits simultaneously, right-to-left. The machine will therefore take *pairs* of bits as input, in reverse order. Recall how to add binary numbers by hand:

```
 1101
 0110
10011
```

The Mealy machine below considers each vertical bit-pair, right-to-left, mimicking what we do manually. We assume it emits the result in right-to-left order.

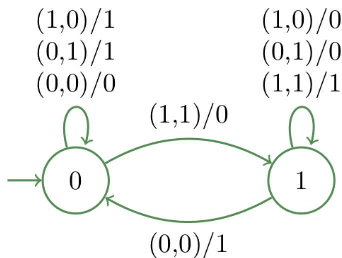


Figure 2–42: Adding two binary numbers

The one thing missing is the leftmost carry bit (only 0011 is printed for the sum above instead of 10011). To solve this problem, we assume that each state emits the bit in its label, which is the carry, when the machine halts there. The code below implements the carry.

```
def binadd(x,y):
    assert len(x) == len(y)
    state = '0'
    pairs = list(zip(x,y))      # combine strings pairwise
    result = ''
    for pair in reversed(pairs):
        if state == '0':
            if pair in [('1','0'), ('0','1')]:
                result += '1'
            elif pair==('0','0'):
                result += '0'
            elif pair==('1','1'):
                result += '0'
                state = '1'
            else:
                assert False, 'invalid input'
        else:   # state == '1'
            if pair in [('1','0'), ('0','1')]:
                result += '0'
            elif pair==('1','1'):
                result += '1'
            elif pair==('0','0'):
                result += '1'
```

```
                state = '0'
            else:
                assert False, 'invalid input'
    if state == '1':
        result += '1'            # append carry
    return result[::-1]          # reverse result

print(binadd('1101','0110'))    # 10011
```

Example 2–26

Subtracting two binary numbers follows a similar pattern, except the states track what is *borrowed* instead of what is carried:

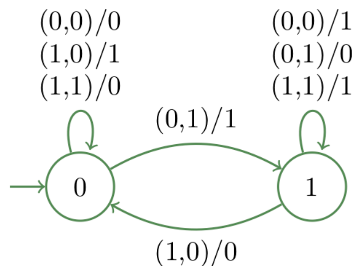


Figure 2–43: Subtracting binary numbers

Using the same input as in the previous example, we get 1101-0110=0111, which in decimal is 13-6=7. Subtracting the other way, we get 0110-1101=1001. The latter result is the two's-complement of 0111, and represents -7 decimal, as expected. Therefore, the bit in the label in each state is the *borrow* (or equivalently, the *sign-bit*; 1 for negative, 0 otherwise).

Example 2–27

As a final example of computer arithmetic, we design a Mealy machine that rounds its binary input down to nearest even number. All this requires is to make the last bit zero if it isn't already. But we need to know when the last symbol has been read, so we need an *end-of-string symbol*; we'll use the dollar sign again. Furthermore, we need to have a *one-symbol delay* in emitting output; since we don't print the dollar sign, we always print the previous symbol consumed after reading the current character. See Figure 2–44 below.

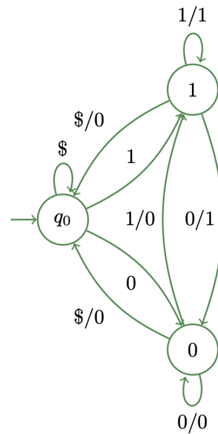


Figure 2–44: Rounds to the nearest even number

The outgoing edges from q_0 emit nothing. Instead, we use the other two states to track the previous symbol read. When the string has been exhausted, we emit a 0 no matter what. The output for 101$ is 100, as expected.

Lexical Analysis

Mealy-style machines are useful in separating input into meaningful parts. In the next example, we consider how to design a machine that will ignore C-style comments from its input, while outputting everything else.

Example 2–28

C-style comments are supported by many popular programming languages and are delimited with a "/*" at the beginning and a "*/" at the end. Since the delimiter is a two-symbol sequence, we need to detect when a slash and asterisk occur *together*. Reading a slash may or may not introduce a comment. What if a slash is followed by another slash? The second slash may or may not belong to a comment, but the first slash must pass through as output regardless. The following diagram is a start on the design for a Mealy Machine to solve the problem.

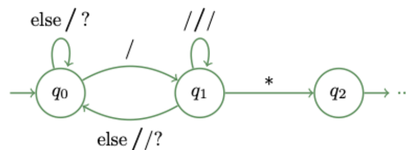


Figure 2–45: Partial machine for removing C-style comments

Technically, Mealy machines have single symbols for input and always have an output symbol on each edge, so that the input and output have the same number of symbols (this is important in electronics). We are obviously taking some liberties with notation here. First, we use the input "else" to cover all cases where the other edges leaving a state do not apply. We also use the question mark as a wildcard symbol representing "the current symbol being processed," as we did in Chapter 1. Finally, we omit the separating slash when a transition produces no output, as in the edges from q_0 to q_1 and q_1 to q_2 above. (Some authors use lambda after a slash to indicate no output.)

Starting in state q_0, we merely echo the input until we encounter a slash, in which case we move to state q_1. At this point, we don't know if that slash actually begins a comment. If an asterisk does not follow immediately, we must echo the *previous* slash, so we are *one symbol behind* in emitting output in state q_2. We can also encounter multiple slashes in a row (e.g., if the code contains `x //* this is the divisor: */y`.) Even though it may not be syntactically correct for some languages, it is also possible that a language uses `//` as a valid operator (like Python 3). If a language uses C-style comments *and* has such an operator, as in `x//y`, then when we reach the `y`, we must output both a slash and the `y`. This explains why the edge going back to state q_0 from q_1 prints two symbols.

Once we get an asterisk in state q_1, we are inside a comment so we do not print the preceding slash and we move to state q_2. Similar logic applies when exiting a comment, which you can complete as an exercise.

Example 2–29

One of the first steps in compiling source code, in addition to removing comments, is to recognize the "words" in the code, that is, "tokens" such as keywords, variables, constants, and operators. A token consists not only of its text but also its type, or *class*. For example, in the expression `x+1`, the compiler needs to know that `x` is a variable, `+` is an operator, and `1` is a literal/constant. A machine can extract a stream of tokens and pass the result on to the next phase of compilation.

To illustrate, we will use a simple language from Louden[7], TINY, which contains only the following tokens:

Tokens in the TINY Language

Description	Class	Representation
Identifiers	ID	Consist of only alphabetic symbols
Keywords	KWD	Special identifiers: **if then else repeat until read write end**
Numbers	NUM	Base-10 integers, leading zeroes okay
Operators	OPR	`+ - * / < > ( ) =` (equality)
Semi-colon	SEMI	;
Assignment	ASGN	:=

[7]Louden, Kenneth C., Compiler Construction: Principles and Practice, PWS Publishing, 1997. Used with permission.

Comments are delimited by single, opening and closing curly braces. Here is sample TINY program:

A sample TINY program

```
{ A TINY Program
    CS 3240
}
    read x;
    if 0 < x then
        fact := 1;
        repeat
            fact := {embedded comment}fact * x;
            x := x - 1
        until x=0;
        write fact
    end
    {}
```

A lexical analysis of this program could be depicted as follows:

Classifying the tokens in the TINY program

```
KWD: read
ID: x
SEMI: ;
KWD: if
NUM: 0
OPR: <
ID: x
KWD: then
ID: fact
ASSIGN: :=
NUM: 1
SEMI: ;
KWD: repeat
ID: fact
ASSIGN: :=
ID: fact
OPR: *
ID: x
SEMI: ;
ID: x
ASSIGN: :=
ID: x
OPR: -
```

```
NUM: 1
KWD: until
ID: x
OPR: =
NUM: 0
SEMI: ;
KWD: write
ID: fact
KWD: end
```

The machine in Figure 2–46 below, doesn't precisely follow the definition of a Mealy machine, but it does indicate how to produce the required output above. It collects the symbols of a token and then outputs its text and token class when it returns back to the start state.

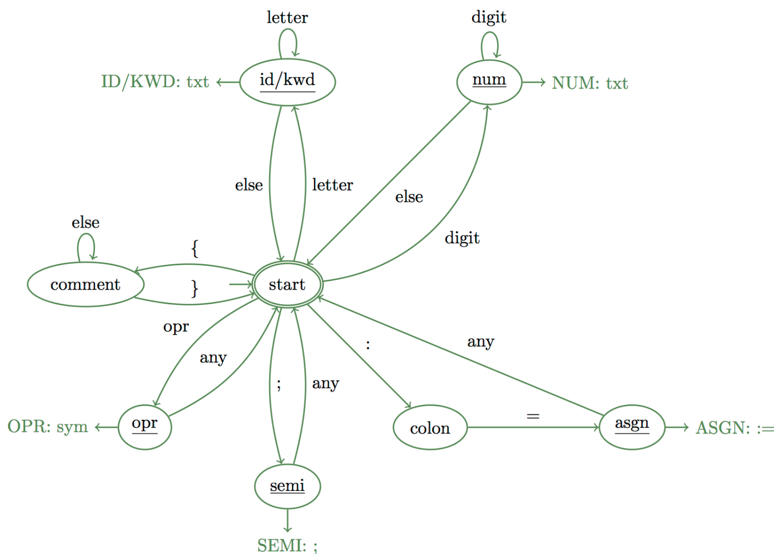


Figure 2–46: Machine that accepts TINY tokens

Note the free arrows exiting the states representing the token classes. We use these to indicate that when leaving that state, the input it has collected for its token will be emitted, along with the token class identifier. The tokens **id** and **kwd** are combined. This suggests an implementation that searches a table of keywords on the identifier to determine whether its class is ID or KWD. Because of space limitations, this diagram is missing a loop on the start state ignoring whitespace. Finally, since tokens are often delimited by non-space symbols, those symbols must be pushed back onto the input stream in some cases when returning to the start state. The start state is accepting to indicate that the input was syntactically valid. A string that does not end in the start state contains a syntax error.

Minimal Mealy Machines

We can minimize the number of states in a Mealy machine just as we did for DFAs. The machine below, which prints a 1 for each b and a 1 for every other a, starting with the first, has one more state than it needs.

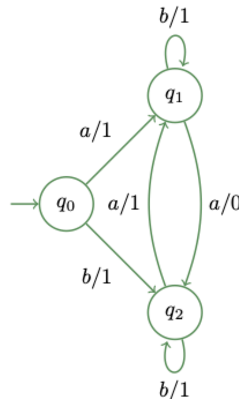


Figure 2–47: An "inefficient" Mealy Machine

Since we want the first a to emit a 1, and then alternate from there, we can eliminate state q_0 and make q_2 the initial state:

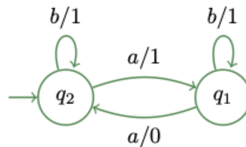


Figure 2–48: A minimal version of Figure 2–47

No move is needed for b since it emits a 1 regardless of state.

The process to minimize a Mealy machine is like the DFA minimization process in section 2.3, except instead of acceptability[8], it is the *output* that initially distinguishes one state from another. Observe the output of the different states below.

$$\delta(q_0, a) = 1, \delta(q_0, b) = 1$$
$$\delta(q_1, a) = 0, \delta(q_1, b) = 1$$
$$\delta(q_2, a) = 1, \delta(q_2, b) = 1$$

[8]Indeed, acceptability is the "output" of a state in a DFA, should the input terminate there. So the first step for both DFAs and Mealy machines is in essence to distinguish states by their "output".

The outputs for q_0 and q_2 are identical. In this simple case, we already know we can combine these states, but in general we must wait until all distinguishable pairs of states are found before merging indistinguishable states. The rest of the algorithm is identical to the state-minimization algorithm in the previous section.

To *initialize* the process of minimizing a Mealy machine, compare the *output* from different states.

Example 2–30

To illustrate the algorithm of minimizing a Mealy machine, inspect Figure 2–49 below.

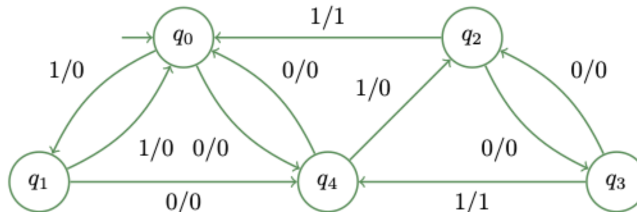


Figure 2–49: A Mealy machine to minimize

As before, we determine which states can be distinguished by first examining their output symbols:

$$\delta(q_0, 0) = 0, \delta(q_0, 1) = 0$$
$$\delta(q_1, 0) = 0, \delta(q_1, 1) = 0$$
$$\delta(q_2, 0) = 0, \delta(q_2, 1) = 1$$
$$\delta(q_3, 0) = 0, \delta(q_3, 1) = 1$$
$$\delta(q_4, 0) = 0, \delta(q_4, 1) = 0$$

States q_0, q_1, and q_4 each give an output of 0 for both inputs, and the other two states yield 0 for a and 1 for b. Thus, each state in $\{q_0, q_1, q_4\}$ is distinguishable from those in $\{q_2, q_3\}$, giving the initial table below.

	q_1	q_2	q_3	q_4
q_0		✔	✔	
q_1		✔	✔	
q_2				✔
q_3				✔

We now follow the state minimization algorithm from section 2.3:

(q_0, q_1):

$\delta(q_0, 0) = q_4, \quad \delta(q_1, 0) = q_4$

$\delta(q_0, 1) = q_1, \quad \delta(q_1, 1) = q_0$

(q_0, q_4):

$\delta(q_0, 0) = q_4, \quad \delta(q_4, 0) = q_0$

$\delta(q_0, 1) = q_1, \quad \delta(q_4, 1) = q_2$ ✔

(q_1, q_4):

$\delta(q_1, 0) = q_4, \quad \delta(q_4, 0) = q_0$ ✔

(q_2, q_3):

$\delta(q_2, 0) = q_3, \quad \delta(q_3, 0) = q_2$

$\delta(q_2, 1) = q_0, \quad \delta(q_3, 1) = q_4$ ✔

We now add check marks to distinguish (q_0, q_4) and (q_1, q_4) in preparation for another iteration:

	q_1	q_2	q_3	q_4
q_0		✔	✔	✔
q_1		✔	✔	✔
q_2			✔	✔
q_3				✔

On the next iteration (not shown) q_0 and q_1 remain indistinguishable and can therefore be combined. See Figure 2–50.

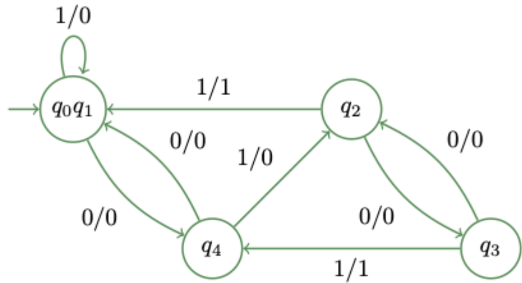


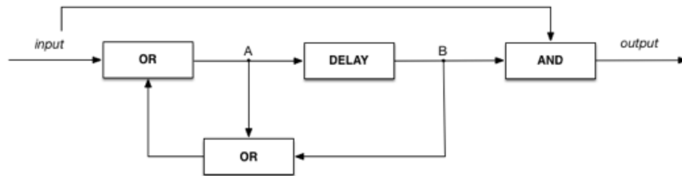
Figure 2–50: Minimal version of Figure 2-49

Key Terms

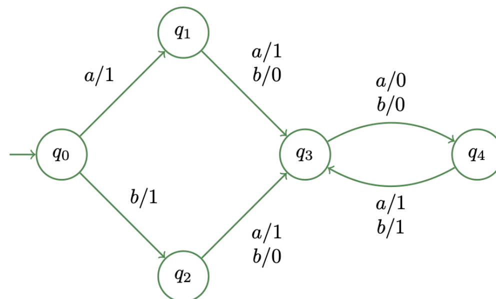
Mealy machine (aka "finite state transducer") ● delayed output

Exercises

1. Draw a Mealy machine that reads ones and zeroes and prints a one every time the substring 1001 is recognized. For example, the input 01001001 yields 00001001.
2. Draw a Mealy machine that performs a circular shift-left operation on strings of a's and b's with an end-of-string marker, so that $abaab$\$ yields the string $baaba$.
3. Draw a Mealy machine that rounds an even binary number up to the nearest odd binary number. This also needs an end marker (\$).
4. Draw a Mealy machine that reads bits and prints an even-parity bit after every four bits. You may assume that the input length is a multiple of four. As an example, the input string 10110110 yields 1011**1**0110**0**.
5. Find a Mealy machine for the circuit depicted below.



6. Use the procedure described at the end of this section to minimize the following Mealy machine.



7. Create a machine based on Figure 2–45 to also ignore //-style comments (which ignore everything through the end of the current line) in addition to traditional C-style comments.

Programming Exercises

1. Write a program that implements the comment-ignoring DFA from Exercise 7 above.
2. Implement the machine in Figure 2–46. Use the sample TINY program in this section as input to obtain the expected output.
3. Implement the machine in Exercise 4 above.

Chapter Summary

Deterministic finite automata are useful because they describe so many common scenarios where changes in state occur. They are also straightforward to implement, but can at times be challenging to design correctly. Non-deterministic finite automata are often easier to design than DFAs, but are more difficult to implement. Fortunately, NFAs have an associated DFA that performs the same function, obtained by subset construction. Both types of automata characterize the regular languages. In addition, for every regular language, there is a unique DFA with a minimal number of states.

Finite automata also model processes that convert input into meaningful output, such as filtering out certain text or counting the number of occurrences of substrings. These so-called Mealy machines also have a unique, minimal configuration.

Glossary

algorithm
A TM/procedure that always halts, yielding output

automaton
a finite-state machine

breadth-first traversal
a graph traversal algorithm that visits all nodes at the same distance from a given node before preceding to the next level

Chomsky Hierarchy
a classification of all types of formal languages

Chomsky Normal Form (CNF)
a standard form for context-free grammars where each rule generates either a single terminal or exactly two non-terminals

Church-Turing Thesis
the conjecture that any computation can be expressed as a Turing Machine

closure (of operators)
the property that the result of any operations on elements from a set remains in that set

computable
a function or language that can be processed by a Turing machine that always halts

computation
a procedure that processes input and yields output (if it halts); the meaningful manipulation of symbols

computational grammar
an unrestricted grammar that computes a function given an input string delimited by variables

context-free
describes a language recognized by a pushdown automaton or generated by a context-free grammar (the latter has only a single variable on the left-hand sides of its rules)

context-sensitive language/grammar
a language decided by a Linear Bounded Automaton, or generated by a non-contracting, unrestricted grammar

countable
describes a set whose elements can be arranged in a one-to-one correspondence with (a subset of) the natural numbers

decidable
a language or function for which there is a TM that always halts and accepts/computes the language/function

depth-first traversal
: a graph traversal algorithm that visits nodes in complete, left-most paths before preceding to other paths

derivation
: the process of generating a string from a grammar

determinism
: describes a process that operates the same way, without variation or ambiguity, every time the same input is given

DFA deterministic finite automaton; traverses the same path every time the same input is given

DPDA
: deterministic pushdown automaton

dynamic programming
: a procedure that works in stages, storing the results from each stage, so that subsequent stages may use results from previous stages

encoding
: a symbolic representation of set elements

formal language
: a set of strings from from symbols of a given finite alphabet

grammar
: a set of substitution rules for describing/generating strings in a language

halt state
: a state in an automaton that has no moves (out-edges); equivalently, it never appears in the domain of a transition function

instantaneous description (ID)
: of a Turing machine; a representation of the current state, the position of the read/write head, and the current contents of the used portion of the tape

lambda closure
: of a state in an NFA; the state itself along with the states reachable from that state by one or more lambda transitions

Kleene star
: zero or more concatenations of symbols of an alphabet or strings of a set

LBA Linear Bounded Automaton, a Turing machine that only needs a constant factor times the number of input symbols for working tape space

non-contracting grammar
: a formal grammar where the length of subsequent sentential forms in a derivation never decrease

non-deterministic finite automaton (NFA)
: a finite-state machine that allows zero or more transitions on a state for the same input symbol, as well as lambda transitions

non-terminal
: another name for a variable in a formal grammar; not a symbol in the alphabet in use

normalized PDA
: a PDA where every lambda-pop transaction is accompanied by equivalent non-lambda-pop transitions popping each symbol of the stack alphabet

nullable
: a variable in a formal grammar that can yield lambda in one or more substitutions

operator associativity
: describes how adjacent operators of the same precedence are ordered (right vs. left associative, achieved with right or left recursion in a CFG)

operator precedence
: describes the order of evaluation for adjacent operators (governed by where the associated rule appears in a CFG)

parse/derivation tree
: a graphical representation of a derivation using a context-free grammar

partial function
: a function defined only on a subset of its domain

post machine (queue machine)
: a finite automaton that uses a queue for auxiliary storage

product table
: a tabular representation of the simultaneous execution of two automata

pushdown automaton (PDA)
: a finite automaton that uses a stack for auxiliary storage

recognizable
: describes a recursively enumerable language or partial function that has an associated Turing machine that halts for only a proper subset of its domain

recursive language
: a formal language decided by some Turing machine (always halts)

recursive variable
: a variable in a formal grammar that appears both on the left and right-hand sides of the same rule

recursively enumerable language (RE)
: a formal language that has an associated Turing machine that halts on strings in the language, but may not halt on strings not in the language

reduction
: of one problem/computation to another. Used to show that a problem/computation is solvable/-computable or not in terms of another problem/computation

regular
: describes a language represented by a regular expression or grammar, or that is decided by a finite automaton

regular intersection/union
: the intersection/union of a context-free language with a regular language

subset construction
: the process of tracking all possible output states together as a composite state when converting a NFA to a DFA

terminal
: a symbol in the alphabet of a formal language

total function
: a function defined on all input in its domain

transition
: an action in an automaton, represented by an edge in a transition graph, an entry in a transition table, or by a transition function

transition graph
: a directed graph that represents how states change when processing input symbols from an alphabet

transition table
: a tabular representation of an automaton, showing how input symbols map to states (and sometimes to auxiliary storage)

Turing machine
: a finite-state machine that uses a two-way, read/write, infinite tape for auxiliary storage

uncountable
: describes a set that is not countable

undecidable
: a language or computation for which there is no algorithm/TM to decide it (may hang on some inputs)

unit production
: a grammar rule of the form $V_1 \to V_2$, where V_1 and V_2 are single variables

unrestricted grammar
: a formal grammar that allows strings on the left-hand side of a rule instead of just single variables (enables contextual replacement)

useless variable
: a variable in a formal grammar that either is not reachable from the start variable, or that never yields a terminal string

Bibliography

1. Bernhardt, C., *Turing's Vision: The Birth of Computer Science*, MIT Press, 2016.
2. Brookshear, J., *Theory of Computation: Formal Languages, Automata, and Complexity*, Benjamin/Cummings, 1989.
3. Cohen, D., *Introduction to Computer Theory*, Second Edition, Wiley, 1997.
4. Enderton, H., *Computability Theory: An Introduction to Recursion Theory*, Academic Press, 2011.
5. Feynman, R., *Feynman Lectures on Computation*, Perseus, 1996.
6. Hopcroft, J., and Ullman, J., *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, 1979.
7. Kinber, E., and Smith, C., *Theory of Computing: A Gentle Introduction*, Prentice-Hall, 2001.
8. Lewis, H., and Papadimitriou, C., *Elements of the Theory of Computation*, Prentice-Hall, 1981.
9. Linz, P., *An Introduction to Formal Languages and Automata*, Sixth Edition, Jones & Bartlett, 2017.
10. Louden, K., *Compiler Construction: Principles and Practice*, PWS, 1997.
11. Martin, J., *Introduction to Languages and the Theory of Computation*, Fourth Edition, McGraw-Hill, 2011.
12. Minsky, M., *Computation: Finite and Infinite Machines*, Prentice-Hall, 1967.
13. Nagpal, C., *Formal Languages and Automata Theory*, Oxford University Press, 2011.
14. Parkes, A., *A Concise Introduction to Languages and Machines*, Springer-Verlag, 2008.
15. Rich, E., *Automata, Computability, and Complexity: Theory and Applications*, Prentice-Hall, 2008.
16. Scott, M., *Programming Language Pragmatics*, Fourth Edition, Morgan Kaufmann, 2016.
17. Shallit, J., *A Second Course in Formal Languages and Automata Theory*, Cambridge University Press, 2009.
18. Sipser, M., *Introduction to the Theory of Computation*, Third Edition, Cengage Learning, 2013.
19. Sudkamp, T., *Languages and Machines: An Introduction to the Theory of Computer Science*, Addison-Wesley, 1997.
20. Webber, A., *formal language: A Practical Introduction*, Franklin-Beedle, 2008.

Index

accepting state, 5
algorithm, 1
ambiguous CFG, 148
automata (automaton), 6
  deterministic, 16
  minimal, 38
  non-deterministic, 25, 28
  regular, 15
breadth-first traversal, 246
canonical order, 2
character class, 64
Chomsky Hierarchy, 267
Chomsky Normal Form (CNF), 176
Church-Turing Thesis, 238
closure
  lambda, 32
closure properties
  and non-regular languages, 110
  of context-free languages, 193
  of deterministic context-free languages, 193
  of recursively enumerable languages, 251
  of regular languages, 88
complement, 19
  of deterministic PDA, 126
  of NFA, 34
composite state, 29
computation, 1, 12
computational grammar, 259
concatenation, 3
  of CFGs, 138
  of DFAs, 67
context-free grammar (CFG), 135
  ambiguous, 148
  from PDA, 159
  CNF rules, 183
  removing lambda, 176
  removing unit productions, 180
  to PDA, 155
context-free language (CFG)
  pumping theorem for, 213
context-sensitive language, 251
contrapositive, 105
countable sets, 268
CYK algorithm, 196
depth-first traversal 98
derivation
  and CFGs, 135
  tree, 145
DFA, 16
  complement of, 19
  minimal, 38
dynamic programing, 196
end-of-string symbol, 51
enumerable sets, 268
expression trees, 151
final state, 5
finite-state machine, 5
  applications of, 9–11
formal grammar, 4
formal language, 2
function
  transition, 16
generalized transition graph (GTG) 71
grammar(s), 4
  computational, 259
  context-free, 135
  context-sensitive, 259
  formal, 4
  left-linear, 78
  regular, 74
  right-linear, 75
  simplifying, 140
  unrestricted, 253
graph
  generalized transition, 71
  reachability, 142
  transition, 5
halt state 225, 226
Halting problem 272
jail state, 19
Kleene star, 3
Instantaneous description (ID) of TMs, 227
lambda, 2
  closure, 32
  removing from CFG, 176
  transition, 27
lambda-pop transition, 157
language(s), 1
  closure properties of context-free, 189
  closure properties of recursively enumerable, 251
  closure properties of regular, 88
  context-free, 115
  formal, 2
  recursive 238, 250
  recursively enumerable 238, 250
  regular, 23
left-linear grammar, 78
lexical analysis 52–56
linear bounded automaton (LBA), 251
machine
  abstract 1
  finite-state, 5
  Mealy, 46
  Turing, 12

Mealy Machine, 46
minimal, 56–58
with delay, 51
membership problem, 275
minimal DFA, 38
modulus, 21
NFA 25, 28
complement of, 34–36
from regular expression, 66
to DFA, 29–34
to regular expression, 69
non-terminal (variable), 134
left-recursion of, 149
non-terminating, 141
nullable, 176
right-recursion of, 150
unreachable, 140
operator
associativity, 149
precedence, 148
parity bit, 47
parse tree, 145
parsing, 196
pigeonhole principle 100, 103, 212
Post (queue) machine 223
postfix notation, 152
powerset 28, 269
prefix notation, 152
product table
context-free with regular languages, 191
regular languages, 89
Pumping Theorem
for context-free languages, 213
for regular languages, 104
pushdown automata (PDA), 115
deterministic 119, 126–130
from CFG, 155
non-deterministic, 112
normalized, 162
to CFG, 159
transition table, 123
with two stacks, 222
reachability graph of CFG, 142
recognizer
Mealy machine, 46
Turing machine, 238
recursive language, 238, 250
properties, 251
recursively enumerable language, 238, 250
properties, 251
reductions (of TMs), 276
regular expression, 62
algebra, 64
complement of, 71
from NFA, 69
to NFA, 66
regular grammar, 76
regular intersection, 190
regular union, 190
regular language, 23
closure properties of, 88
pumping theorem for, 104
Rice's Theorem, 279
right-linear grammar, 75
sentential form, 134
sequential circuit, 48
stack
alphabet, 118
operations, 116
variable lifetime, 160
state(s)
accepting, 5
composite, 29
distinguishable, 40
final, 5
jail (dead), 19
structured programming, 238
subset construction, 31
subroutines, 234
syntax tree, 145
table
product, 89
transition, 17
tape alphabet, 234
tokenization, 53–55
transition, 5
lambda, 27
transition function, 16
transition table, 17
for PDA, 123
Turing machine, 12, 225, 234
and functions, 230
and "hanging", 237
and stay option, 241
and subroutines, 234
and unrestricted grammars, 260
encoding, 242–245
multi-tape, 241
multi-track, 239
non-deterministic, 245
universal, 242
uncountable sets, 269
undecidability, 276
unit production, 176, 180
removing, 180
unrestricted grammar, 253
and Turing machines, 260
variable (see non-terminal)