THEORY OF COMPUTING

AN OPEN INTRODUCTION

Taylor J. Smith

THEORY OF COMPUTING

AN OPEN INTRODUCTION α

Taylor J. Smith *St. Francis Xavier University*

Copyright © Taylor J. Smith 2024
https://taylorjsmith.xyz

This work is licensed under a Creative Commons
Attribution-ShareAlike 4.0 International License.
For license details, visit
https://creativecommons.org/licenses/by-sa/4.0/.

Typeset in Computer Modern using LaTeX
Figures produced using Ti*k*Z

α pre-publication edition, September 2024

CONTENTS

PREFACE

When you think about the theory of computation, what comes to mind? In fact, what *is* the "theory" of computation? Computers are real, tangible machines that humans built, so surely we should know all about how they work. However, all the day-to-day work we do with our computers belies the reality of computation itself and all of its intricacies.

At some point, we've all found ourselves in a position where we wanted to do something with our computer, but the hardware just wasn't up to the task. Imagine, then, if we were to remove all of the physical components of a computer—its finite memory, its finite storage space, its limited processing power. In doing so, we would end up with an ideal computer, a machine with no limitations, one that can solve any problem we throw at it... or so it would seem.

In this book, we will build our way up from a simple machine that can answer nothing more than yes/no questions all the way to our ideal computer. In the process, we will learn about the fundamental limits of computation itself. What is a computer truly capable of? What makes some problems harder for a computer to solve than others? What kinds of problems can a computer solve at all? What kind of problems cannot ever be fully solved by a computer, no matter how many resources we throw at it? We will investigate all of these questions, and much more.

ABOUT THIS BOOK

For students. The chances are high that, if you're reading this book, you're enrolled in a course titled "Theory of Computing" or something similar. And the chances are just as high that you need to take that course as a part of your degree program. (If you're taking the course for fun, then you're truly a student after my own heart.) In my career, I have noticed that the theory of computing has a reputation among students for being dry and difficult, largely because these students are made to take a course with no coding, devoid of flashy tech, and even worse: packed with mathematics.

But I believe this reputation is undeserved. Yes, theory isn't flashy, and yes, theory is mathematical. But theory is also beautiful in that it reveals a side of computer science that no other course comes close to touching.

Before we embark on our journey, consider abandoning any preconceived notions you may have gleaned from other students. While it may take a little time to build up our vocabulary and notation, the results we will study in this book are truly deep and enlightening. The theory of computing is a unique subject in that it touches literally every other area of computer science in some way, and you're invited to find and explore the many connections between the material we learn in this book and the material from the other areas of computer science that interest you.

The material in this book does not require any more prerequisite knowledge on your part than a familiarity (and comfort) with discrete mathematics and introductory computer science—namely, fundamental data structures and programming constructs. If you feel you need to brush up on your mathematical knowledge, I have included some review material in Appendix A.

I have tried to write this book in a way that stimulates your curiosity and encourages you to come back to it again and again, even if you only ever end up taking one course in the theory of computing. As a first-timer, you or your class might read the first few chapters to become acquainted with abstract models of computation. Then, later, you might focus on the following chapters where we progress from computation to complexity theory. Further reads may have you adventure into the later chapters that explore specialized topics. No matter how you approach the material, whether it be in the classroom or on your own time, I hope you return to these pages and learn something new on every read.


In certain parts of this book, you will encounter paragraphs marked with a "dangerous bend" sign. Inspired by Nicolas Bourbaki and Donald Knuth, who each used the symbol in their works, I include this warning sign at any place where I feel the material might be more difficult to grasp on one's first reading. Just like driving on a winding road, take it slow and easy, and exercise caution with these paragraphs!

Additionally, as you progress through the book, you may notice that I have included comprehensive chapter notes with pointers to the literature at the end of each chapter. I have also interspersed citations at appropriate spots throughout the text itself as the need arises. One can argue that computer science is unique in how forward-looking it is as a field: the state of the art changes monthly, if not weekly. But this comes with a downside in that computer scientists rarely stop to evaluate and appreciate the history of our subject. Thus, if any particular topic in this book grabs your interest, I strongly encourage you to track down copies of the papers and books I have cited, either through your library or online. There's no better way

to appreciate the ideas in this book than to read the actual words of the people who discovered them.

For instructors. This book emerged from the lecture notes I wrote for my undergraduate and graduate-level courses on the theory of computing. My lecture notes were, in turn, influenced by what I studied as an undergraduate and graduate student. The material I chose to include in this book was guided not only by what students *ought* to learn, but also by what I *wanted* to learn as a student myself. Thus, while you'll find all of the core material within these pages, you'll also find quite a bit of enriched content for those looking to take that extra step into this exciting subject.

The approach that I take to teaching the theory of computing is influenced (perhaps heavily) by my research focus as an automata theorist. I believe the best way to introduce undergraduate students to this material is by building up from a simple and accessible model of computation: the finite automaton. I am aware of other approaches to teaching this material that begin with alternative models such as Boolean circuits, but I don't agree with this pedagogy: although circuits as a model prevail in both the research literature and in hardware, I find them too technical and fiddly to teach as a *first* model. They may be suitable for electrical engineering students, but we're teaching future computer scientists who should be used to abstraction.

By starting with a model that can do nothing more than read input and either accept or reject, students can easily grasp fundamental properties and draw connections between the model and more familiar real-world tools, such as regular expressions. From these foundations, I demonstrate to students how augmenting the finite automaton—first with a stack and then with a tape—produces progressively stronger models of computation. Here, I follow an analogous progression in drawing out the fundamental properties of both the pushdown automaton and the Turing machine.

With the introduction of the Turing machine, the focus of the book shifts from the models of computation themselves to what the models are capable of computing. I introduce students both to decision problems and to the idea of "programming" a Turing machine in the form of describing the steps performed by a Turing machine in order to decide a problem. My undergraduate course culminates in a discussion on the limits of computation and undecidable problems.

(The material in the present edition of the book stops here. Everything discussed in the following paragraphs will appear in future editions of the book. See the following section, "About This Edition", for more details.)

My graduate course picks up our study more or less where my undergraduate course leaves off; indeed, the only prerequisite knowledge I assume of my graduate students is that they know what Turing machines are and how they perform computations. Here, the book shifts focus once more,

from discussing *whether* problems can be solved to discussing *how efficiently* problems can be solved. The majority of the material I cover at the graduate level focuses on complexity theory: we discuss foundational results in complexity before focusing on time and space complexity classes, hardness, completeness, and complements. (I also cover basic complexity theory notions such as P, NP, and NP-completeness in my undergraduate-level algorithm analysis course, but not in my undergraduate-level theory of computing course.)

One aspect of my graduate-level course that I particularly enjoy is that, after covering the major results in complexity theory, I get to veer off into specific areas and special topics. These topics form the final focus of the book, often covering newer results and more obscure material that students might not otherwise learn during their studies. Many of the topics I discuss in my graduate-level course were inspired by presentations given in past offerings of the course, where students independently study some advanced aspect of the theory of computing and deliver a mini-lecture in the final weeks of the term. If you teach a similar course, I encourage you to try the same activity: with any luck, you'll gain as much inspiration as I do.

I have tried, where possible, to align the material in this book with the illustrative learning outcomes presented in the ACM/IEEE-CS/AIII Computer Science Curricula 2023, available online at `https://www.acm.org/education/curricula-recommendations`. In particular, this book most closely follows the AL-Models and AL-Complexity knowledge units. The particular sections of this book that satisfy the learning outcomes are outlined in Table 1. Note, however, that some learning outcomes—particularly those that are more appropriate for a course on algorithm analysis—do not fall within the scope of this book, and may be better served by other books such as that by Erickson [2019].

ABOUT THIS EDITION

You may have noticed that I refer to the present book as the "α pre-publication edition". This is to highlight the fact that this book is still under active development and revision. In time, I will release the β, γ, δ, etc. ... pre-publication editions, each of which will contain more and (hopefully) better material, until I feel the book is in a good-enough state to receive the designation of "first edition".

Occasionally, you will encounter paragraphs marked with an "under construction" sign. These signs mark areas where I have promised to write future material, but haven't yet done so.

I have already started to lay the groundwork for certain future sections, which I have marked in the book with a diamond ($\diamond$) symbol. Beyond

Table 1
ALIGNMENT BETWEEN THIS BOOK AND CS CURRICULA 2023

Learning outcome	Relevant section of book
AL-Models	
CS Core 1	Sections 1.2, 2.2, 3.1
CS Core 2	Sections 1.2, 2.2, 3.1
CS Core 3	Section 1.1
CS Core 4	Sections 1.1, 1.2
CS Core 5	Sections 1.1, 1.2, 2.1, 2.2, 3.1
CS Core 6	Section 3.5
CS Core 7	Section 5.2
CS Core 8	Sections 4.3, 4.4, 5.2, 5.3, 5.5
CS Core 9	Section 3.6
CS Core 10	*Not covered*
KA Core 11	Sections 1.2, 1.3, 1.4, 2.2, 2.3, 2.4
KA Core 12	Sections 1.5, 2.5
KA Core 13	Sections 4.3, 5.2
KA Core 14	Sections 5.1, 5.4
KA Core 15	Sections 1.3, 2.3, 3.2
KA Core 16	Section 5.7
KA Core 17	Sections 5.2, 5.3, 5.5
KA Core 18	*Not covered*
KA Core 19	*Not covered*
AL-Complexity	
CS Core 1	*Forthcoming*
CS Core 2	*Forthcoming*
CS Core 3	*Not covered*
CS Core 4	*Not covered*
CS Core 5	*Forthcoming*
CS Core 6	*Not covered*
CS Core 7	*Not covered*
CS Core 8	*Forthcoming*
CS Core 9	*Not covered*
CS Core 10	*Forthcoming*
CS Core 11	*Forthcoming*
CS Core 12	*Forthcoming*
CS Core 13	*Forthcoming*
KA Core 14	*Not covered*
KA Core 15	*Not covered*
KA Core 16	*Forthcoming*
KA Core 17	*Forthcoming*
KA Core 18	*Forthcoming*
KA Core 19	*Forthcoming*
KA Core 20	*Forthcoming*

individual sections, my long-term plan is to include the following chapters—for which I have material written—into future pre-publication editions of this book:

6 Foundations of Complexity
Discussing basic complexity classes, speedup and compression, the gap theorem, constructible functions, the time and space hierarchy theorems, and Savitch's theorem.

7 Time Complexity
Discussing basic time complexity classes, including P, NP, E, NE, EXP, and NEXP.

8 Space Complexity
Discussing basic space complexity classes, including L, NL, PSPACE, NPSPACE, EXPSPACE, and NEXPSPACE.

9 Hardness, Completeness, and Complements
Discussing polynomial-time many-one reductions, NP-completeness, PSPACE-completeness, logarithmic-space many-one reductions, NL-completeness, complement complexity classes, and the Immerman–Szelepcsényi theorem.

10 Probabilistic Computation
Discussing probabilistic Turing machines, Las Vegas and Monte Carlo algorithms, and randomized complexity classes.

11 Provers, Verifiers, and Interactivity
Discussing proof systems, Arthur and Merlin, interactive protocols, zero-knowledge proofs, and probabilistically checkable proofs.

My longer-term (and very optimistic) plan will see the following not-yet-written chapters added to the book at some point in the future:

Relativization

Circuit Complexity

Quantum Computation

. . . and more?

I realize that the present book, unlike most textbooks, does not yet contain exercises, and this is another omission that I intend to rectify in future editions. I have a variety of questions collected from assignments I have given to my students, and in time I will include an appropriate selection of these questions at the ends of each chapter.

Lastly, one feature I will eventually add to this book is an index, though this will likely be added only after I have written up all of the content outlined previously.

ON BEING OPEN ACCESS

At some point in the now-far past, possibly after I returned home from the university bookstore with an armful of textbooks and course readings, I came across the following quote that stuck with me:

> *Information wants to be free.*
> — STEWART BRAND

In the following years as I progressed through my higher education, in spite of the growing influence of the internet, I noticed a trend away from the spirit of this quote: more expensive textbooks, course materials locked away behind learning management systems, the commodification of education. This trend has instilled within me the strong belief that high-quality educational materials should be made available for free, in perpetuity, to everybody. Moreover, and perhaps influenced by the impacts both the free software and open source movements had on me during my upbringing in computer science, I believe that "free" in this context should mean not just without cost (*gratis*) but also with minimal restriction (*libre*).

Authors are often asked about their motivation for writing a book: my motivation is to make information free. To this end, I have published this book under a Creative Commons BY-SA license to encourage everyone to use this material in the spirit of David Wiley's five Rs of openness:

- **Retain**: download, print, and own this book for as long as you like.
- **Reuse**: use part or all of the material in this book however you want.
- **Revise**: add to or modify the material for use in your own classroom, create audio and video materials based on the material, translate the book, and make it even better and more accessible.
- **Remix**: take the material from this book and combine it with open educational resources from other courses and institutions.
- **Redistribute**: share this book freely with your friends, colleagues, and strangers.

The terms of this book's particular license are available online at `https://creativecommons.org/licenses/by-sa/4.0/`.

Since this book is CC BY-SA licensed, the only stipulations on these five Rs are that anything you create that is based on this book must include an attribution to the original book and must likewise be licensed openly. If

you use or adapt the material from this book, please include the following attribution:

> Taylor J. Smith. *Theory of Computing: An Open Introduction.* Self-published open educational resource, α pre-publication edition, 2024. taylorjsmith.xyz/tocopen/.

Additionally, even though it isn't a part of the license conditions, please feel free to get in touch with me if you use or adapt this material. I would love to hear about what you do with it!

FURTHER READING

As the author of the present book, I may be slightly biased in believing that you should read this instead of any others. But for readers who are entirely new to this field of study, I can recommend a handful of great alternative undergraduate-level treatments of this subject: Hopcroft and Ullman [1979] wrote the very first book on theoretical computer science I read, Rich [2008] wrote the book I learned from as an undergraduate student, and Sipser [2013] wrote the book I previously used to teach undergraduate students. There are, of course, dozens of other books on the subject, with one favourite of mine being that by Kozen [1997].

Note that, although there are newer editions of the Hopcroft and Ullman (and now, also Motwani) book, I have very deliberately cited the first edition here. By the authors' own admission, the newer editions of their book are "larger on the outside, but smaller on the inside"—not exactly a selling point—and as a result, they pale in comparison to the original. If you can get your hands on a copy of the first edition, grab it!

Try as I might, I cannot include everything in this book, so readers having some more familiarity with the subject may wish to consult various graduate-level texts to gain exposure to advanced topics not mentioned here. In the past, I have used the book by Arora and Barak [2009] to teach graduate students. Other fine advanced books include those by Kozen [2006] and by Papadimitriou [1994].

ACKNOWLEDGEMENTS

There's nothing quite like writing a book to prompt you to reflect on who drove you to this point to begin with. First and foremost, I must thank my family and friends for their unyielding support and for reminding me that there's more to life than work. When it comes to work, I wouldn't be where I am today were it not for the professors who taught me the very

subject I'm writing about: I thank Lucian Ilie and Helmut Jürgensen of the University of Western Ontario, Gregor Richards and Jeffrey Shallit of the University of Waterloo, and particularly Kai Salomaa of Queen's University, who also served as my indefatigable doctoral supervisor and continues to be my close colleague and confidant. I would also be remiss not to thank Mark Daley of the University of Western Ontario for lending me his copy of Hopcroft and Ullman (first edition, naturally) over the summer between the first and second years of my bachelor's degree, thereby being directly responsible for me becoming a theorist.

I would like to express my gratitude to St. Francis Xavier University for providing me with the means to write this book, and to the many students in CSCI 356 and CSCI 541 who have learned (and hopefully continue to learn) from my materials. While writing this book, I benefitted from research funding support through the Natural Sciences and Engineering Research Council of Canada Discovery Grant RGPIN-2024-04799.

Your feedback. Try as I might, I'm sure that there are mistakes lurking in these pages that snuck past me, even after hundreds of read-throughs and revisions. I'm doubly sure that there are sections of the text where I could've explained something in a different, clearer, or just plain better way. In my pursuit to expunge errors, obliterate omissions, take out typos, and polish prose, I welcome comments from you, the reader. I will also happily consider suggestions for topics you would like to see added to future editions. I can be reached via email at `tjsmith@stfx.ca`.

Antigonish, Canada T. J. S.
September 2024

CHAPTER ONE

REGULAR LANGUAGES

If you frequently use a Unix-based system with a terminal, you may be familiar with utilities such as `grep`, which searches an input text file for lines that match a specified format. For example, on your computer, you can search the dictionary file (`/usr/share/dict/words`) for all words that contain `theory`:

```
$ grep theory /usr/share/dict/words
countertheory
theory
theoryless
theorymonger
```

But, to be fair, doing something like that is a bit overkill when you could just open the file in a text editor and use the Find tool to search for the word "theory". Where `grep` really shines is when you need to search for text matching a *pattern*, like so:

```
$ grep ^u.*ity$ /usr/share/dict/words
ubiquity
ultimity
ultrafilterability
...
usability
utility
utterability
uxoriality
```

In this example, we searched for all words in `/usr/share/dict/words` that began with a `u` and ended with `ity`, such as `university`. The `ubiquity` of this pattern in the English language is evident:

```
$ grep ^u.*ity$ /usr/share/dict/words | wc -l
     235
```

Utilities like `grep` use patterns to perform fast searches in text files, and the sequence of symbols that makes up such a pattern is known as *regex* or, formally, a *regular expression*.

1.1. REGEX AND REGULAR EXPRESSIONS

To define regular expressions, let's think about the types of things we can match. For example, as a base case, we might want to be able to match nothing—this can be represented by a nonsensical regex like `a^`, which attempts to match a symbol `a` that occurs before the start of a line. We might also want to match an empty line (which is distinct from matching nothing!), which can be done with the regex `^$`.

Let's now actually attempt to match something more meaningful. The smallest nonempty thing we can match is a single symbol, which can be matched by a regex consisting of the symbol itself; say, `a`. From this, we can build up more complicated regexes by joining together smaller ones. For instance, we can match two regexes independently by joining them together with a special *union* symbol; say, `(a | b)`, which matches lines that contain either an `a`, or a `b`, or both. We can also combine, or *concatenate*, two regexes by simply putting them together—the regex `ab` matches an `a` immediately followed by a `b`. Lastly, it would be nice to incorporate some kind of *repetition* mechanism to match something never, once, or many times. This can be done using a special "star" symbol such as that in the trivial regex `.*`, which matches zero or more occurrences (represented by the star, `*`) of any symbol (represented by the dot, `.`).

Now, since we're in a mathematically oriented course, we should properly formalize each of these match types. Fortunately, we can bring over notions from mathematics to correspond to each match type. Matching nothing can be denoted by an empty set symbol, $\emptyset$. Likewise, matching an empty line is like matching a set that contains one element which has zero length—let's denote this zero-length element by the symbol ϵ. To denote the union of two regular expressions, we could use $\cup$, but since we're dealing with regular expressions and (strictly speaking) not sets, we'll instead use the symbol $+$. Concatenation is straightforward; we'll simply write the regular expressions side-by-side. Finally, we can keep the star symbol as it is.

Taking this all together, we arrive at a formal definition for regular expressions.

Definition 1.1 (Regular expressions)

Let Σ be an alphabet. The class of regular expressions is defined inductively as follows:

1. $\boldsymbol{r} = \emptyset$ is a regular expression;
2. $\boldsymbol{r} = \epsilon$ is a regular expression;
3. For each $a \in \Sigma$, $\boldsymbol{r} = a$ is a regular expression;
4. For regular expressions $\boldsymbol{r}_1$ and $\boldsymbol{r}_2$, $\boldsymbol{r}_1 + \boldsymbol{r}_2$ is a regular expression;
5. For regular expressions $\boldsymbol{r}_1$ and $\boldsymbol{r}_2$, $\boldsymbol{r}_1\boldsymbol{r}_2$ is a regular expression; and
6. For a regular expression $\boldsymbol{r}$, $\boldsymbol{r}^*$ is a regular expression.

Note that our earlier regex examples used symbols like `^`, `.`, and `$`, when Definition 1.1 didn't define any of those symbols. This is because regexes and regular expressions aren't exactly the same thing. In fact, our definition of a regular expression is the purely theoretic definition, meant simply to give us the bare minimum needed to match simple patterns. It is therefore different from a practical regex implementation, where we can use special symbols to indicate the start or end of a word, match any symbol instead of one specific symbol, and make back-references, among other things. Appropriately, the literature sometimes refers to these practical regex implementations as *extended regular expressions*, and this is what you'll encounter on most computers. In the context of this lecture, though, when we refer to a regular expression, we will be following Definition 1.1.

1.1.1. Words, Languages, and Operations

Another way in which regular expressions stand apart is in the terminology we use to refer to what we're matching. Since we aren't using a terminal to write our theoretical regular expressions, we aren't simply matching lines of text in a text file. Instead, a regular expression corresponds more generally to some collection of things that match a pre-specified pattern.

To describe what we can do with regular expressions (and other models of computation we will see later), we will borrow some terminology from linguistics, which happens to be the field from which much of early theoretical computer science developed!

- The symbols we use, like $\{\mathtt{a}, \mathtt{b}\}$ or $\{\mathtt{0}, \mathtt{1}\}$, come from an *alphabet*. Often, we represent an alphabet by the symbol Σ.
- Sequences of symbols are called *words* or *strings*. For example, consider the English lowercase alphabet $\{\mathtt{a}, \mathtt{b}, \ldots, \mathtt{z}\}$. Some

words we can create with this alphabet are `cat`, `computer`, and `pneumonoultramicroscopicsilicovolcanoconiosis`. We often use lowercase variables like w, x, y, or z to denote words, and we use the symbol ϵ to denote the special zero-length *empty word*.

- Sets of words are called *languages*. Much like with plain sets, we can either list the words in a language explicitly, or we can describe a language in terms of some property or properties of each word therein. For example, over the English lowercase alphabet, the language of words with three consecutive double letters is

  $$\{\texttt{bookkeep}, \texttt{bookkeeper}, \texttt{bookkeepers}, \texttt{bookkeeping}\}.$$

  Also much like sets, languages can be either finite or infinite. We often use uppercase variables like L to denote languages, and we use the symbol $\emptyset$ to denote the special *empty language* containing no words.

Since languages are essentially sets, we can apply operations from set theory directly to languages, and doing so allows us to produce new languages. Indeed, we already saw three operations being applied not to languages, but to regular expressions in Definition 1.1. This trio of operations has their own name: the *regular operations*.

Definition 1.2 (Regular operations)

Let L, L_1, and L_2 be languages. The three regular operations are defined as follows:

- Union: $L_1 \cup L_2 = \{w \mid w \in L_1 \text{ or } w \in L_2\}$;
- Concatenation: $L_1 L_2 = \{wv \mid w \in L_1 \text{ and } v \in L_2\}$; and
- Kleene star: $L^* = \bigcup_{i \geq 0} L^i$, where

$$\begin{aligned} L^0 &= \{\epsilon\}, \\ L^1 &= L, \text{ and} \\ L^i &= \{wv \mid w \in L^{i-1} \text{ and } v \in L\}. \end{aligned}$$

The union operation, naturally, works in exactly the same way for languages as it does for sets. The concatenation operation takes two words and "connects" the end of the first word to the beginning of the second word. Lastly, the Kleene star operation—or, as we previously referred to it,

the star operation—is simply repeated concatenation of all words with all other words in some language.

Note that, since the Kleene star allows us to take zero copies of a word, the empty word ϵ is always included in the resulting language.

Example 1.3

Let $L_1 = \{\texttt{a}, \texttt{b}\}$ and $L_2 = \{\texttt{d}, \texttt{e}\}$. Then

$$\begin{aligned} L_1 \cup L_2 &= \{\texttt{a}, \texttt{b}, \texttt{d}, \texttt{e}\}, \\ L_1 L_2 &= \{\texttt{ad}, \texttt{ae}, \texttt{bd}, \texttt{be}\}, \\ L_1^* &= \{\epsilon, \texttt{a}, \texttt{b}, \texttt{aa}, \texttt{ab}, \texttt{ba}, \texttt{bb}, \texttt{aaa}, \texttt{aab}, \dots\}, \text{ and} \\ L_2^* &= \{\epsilon, \texttt{d}, \texttt{e}, \texttt{dd}, \texttt{de}, \texttt{ed}, \texttt{ee}, \texttt{ddd}, \texttt{dde}, \dots\}. \end{aligned}$$

Empty Words and Empty Languages. The empty word ϵ and the empty language $\emptyset$ interact with operations a little differently than other words and languages.

For the empty word ϵ and any language L, we have that

$$\begin{aligned} L \cup \epsilon = \epsilon \cup L \neq L \text{ in general;} \\ L\epsilon = \epsilon L = L; \text{ and} \\ \epsilon^* = \{\epsilon\}. \end{aligned}$$

For the empty language $\emptyset$ and any language L, we have that

$$\begin{aligned} L \cup \emptyset = \emptyset \cup L = L; \\ L\emptyset = \emptyset L = \emptyset; \text{ and} \\ \emptyset^* = \{\epsilon\}. \end{aligned}$$

Other Operations. We may additionally define some shorthand to make our notation look nicer, though strictly speaking, this notation is not "official". Recall that the star symbol matches zero or more occurrences of whatever it's associated with. If we wanted to match one or more occurrences, we could write $L^+ = LL^* = L^*$, and this is referred to as the "Kleene plus" symbol. Similarly, as we did in Definition 1.2, we can use exponents to denote iterated concatenation; that is, L^i denotes L concatenated with itself i times.

1.1.2. Language of a Regular Expression

To tie everything together thus far, we can observe a direct correspondence between regular expressions and languages. Every regular expression *represents* a language, and we denote the language represented by a regular expression $\boldsymbol{r}$ by $L(\boldsymbol{r})$. Note that each of the six basic regular expressions correspond to their own language. If $\boldsymbol{r} = \emptyset$, then $L(\boldsymbol{r}) = \emptyset$. Likewise, if $\boldsymbol{r} = \epsilon$, then $L(\boldsymbol{r}) = \{\epsilon\}$, and if $\boldsymbol{r} = a$, then $L(\boldsymbol{r}) = \{a\}$. The remaining three regular expressions correspond exactly to the union, concatenation, or repetition of their constituent languages. We will denote the class of languages represented by some regular expression by RE.

Often, figuring out the language represented by a given regular expression is as simple as reading through the regular expression and breaking it into its constituent components.

Example 1.4

Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, and consider the language

$$L_{\text{odda}} = \{w \mid w \text{ contains an odd number of } \mathtt{a}\mathtt{s}\}.$$

This language is represented by the regular expression

$$\boldsymbol{r}_{\text{odda}} = \mathtt{b}^*(\mathtt{a}\mathtt{b}^*\mathtt{a}\mathtt{b}^*)^*\mathtt{a}\mathtt{b}^*.$$

The first component of $\boldsymbol{r}$, $\mathtt{b}^*$, recognizes zero or more leading $\mathtt{b}$s. The middle component, $(\mathtt{a}\mathtt{b}^*\mathtt{a}\mathtt{b}^*)^*$, recognizes zero or more pairs of $\mathtt{a}$s, where each $\mathtt{a}$ is followed by zero or more $\mathtt{b}$s. The last component, $\mathtt{a}\mathtt{b}^*$, recognizes an additional $\mathtt{a}$ to ensure the total number of $\mathtt{a}$s is odd, followed by zero or more $\mathtt{b}$s.

Note that regular expressions need not be unique; to illustrate, the same language in Example 1.4 is recognized by the regular expression $\boldsymbol{r}'_{\text{odda}} = \mathtt{b}^*\mathtt{a}\mathtt{b}^*(\mathtt{a}\mathtt{b}^*\mathtt{a}\mathtt{b}^*)^*$.

Working in reverse, we can take a regular expression and determine the language it represents through a straightforward substitution process.

Example 1.5

Consider the regular expression $\boldsymbol{r} = (\mathtt{a} + \mathtt{b})^*\mathtt{b}$ over the alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}\}$. We can "decompose" the language represented by $\boldsymbol{r}$ in the

following way:

$$\begin{aligned} L(\boldsymbol{r}) &= L((\mathtt{a}+\mathtt{b})^*\mathtt{b}) \\ &= L(\mathtt{a}+\mathtt{b})^* L(\mathtt{b}) && \text{(breaking apart concatenation)} \\ &= (L(\mathtt{a}) \cup L(\mathtt{b}))^* L(\mathtt{b}) && \text{(rewriting as union of languages)} \\ &= (\{\mathtt{a}\} \cup \{\mathtt{b}\})^* \{\mathtt{b}\} && \text{(rewriting as single-symbol languages)} \\ &= \{\mathtt{a}, \mathtt{b}\}^* \{\mathtt{b}\}. && \text{(rewriting as union of symbols)} \end{aligned}$$

Therefore, $\boldsymbol{r}$ represents the language $L = \{w \mid w \text{ ends with } \mathtt{b}\}$.

Operational Order of Precedence. Just like how mathematics has an order of operations, regular expressions abide by their own order of precedence. The star is always applied first, followed by concatenation, and then union. If we want to, we can modify the order in which operations are applied by adding parentheses to a regular expression, and this does not affect the language represented by that regular expression.

Example 1.6

Consider the regular expressions

$$\boldsymbol{r}_1 = \mathtt{0} + \mathtt{1}^*\mathtt{10} + \mathtt{1}^* \text{ and } \boldsymbol{r}_2 = (\mathtt{0}+\mathtt{1})^*\mathtt{1}(\mathtt{0}+\mathtt{1})^*.$$

Clearly, the only visual difference between $\boldsymbol{r}_1$ and $\boldsymbol{r}_2$ is the addition of parentheses. However, the languages represented by $\boldsymbol{r}_1$ and $\boldsymbol{r}_2$ are quite different:

- The expression $\boldsymbol{r}_1$ represents the language containing (i) the word `0`, (ii) all words consisting of at least one `1` with one `0` at the end, and (iii) all words consisting of zero or more `1`s.
- The expression $\boldsymbol{r}_2$ represents the language consisting of all words that contain at least one `1`.

Regular Languages. Going all the way back to Definition 1.2, why did we refer to these three operations as the "regular" operations? As it turns out, taking just these three operations is sufficient to allow us to define the smallest class of languages that is interesting enough to study: the *regular languages*.

Definition 1.7 (Regular languages)

Let Σ be an alphabet. The class of regular languages is defined inductively as follows:

1. The empty language $\emptyset$ is regular.
2. The language $\{\epsilon\}$ containing only the empty word is regular.
3. For each $a \in \Sigma$, the language $\{a\}$ is regular.
4. If L_1 and L_2 are regular, then $L_1 \cup L_2$ is regular.
5. If L_1 and L_2 are regular, then $L_1 L_2$ is regular.
6. If L is regular, then L^* is regular.

Remark. There is a smaller class called the class of *finite languages*. However, it's not too interesting: it consists only of languages with a finite number of words. Introducing the Kleene star allows us to produce infinite-size languages.

If we compare Definitions 1.1 and 1.7, we can start to convince ourselves why we used the word "regular" to refer to both of these concepts. All regular languages have a corresponding regular expression, and all regular expressions correspond to a regular language! The similarities between our two definitions mean that we can actually rewrite our definition of the regular languages to draw a direct connection to regular expressions:

Definition 1.7′ (Regular languages)

If some regular expression $\boldsymbol{r}$ represents a language L, then L is regular.

Now, you might be asking yourself: from where did the word "regular" arise to refer to these expressions and languages? Stephen Kleene introduced the notion of a regular language in the 1950s, but his justification for the terminology was basically that he couldn't come up with any better name:

> *We would welcome any suggestions as to a more descriptive term.*
> — STEPHEN KLEENE

This, in turn, brings to mind another famous quote in computer science:

> *There are only two hard things in Computer Science: cache invalidation and naming things.*
> — PHIL KARLTON

1.2. FINITE AUTOMATA

SOME MIGHT ARGUE that the entire point of studying computer science is to determine exactly what computers are capable of. After all, humans created computers so that we could pass off boring or repetitive work onto a machine and give our brains a break! However, considering a full computer at the very beginning of our studies is kind of like learning to swim by jumping into the deep end of a pool. In order to learn without getting overwhelmed, we will begin by considering a very simple model of computation that gives us just enough power to actually perform an elementary computation.

If you've ever purchased something from a vending machine or waited in your car at a traffic light, then you've interacted with a *finite automaton*. Even the computer you use every day relies on a finite automaton to complete its tasks. Consider the conceptual diagram of how a computer schedules processes, depicted in Figure 1.1. When the computer needs a new process, it creates one and enters the circle labelled "new". The arrow labelled "start" indicates that this is where the life of the process begins. When sufficient memory is available, the computer admits the process by following the arrow to the circle labelled "ready". Then, the computer's scheduler dispatches the process and moves to the circle labelled "running". At any time, the computer can interrupt the process or wait for an event to occur; these arrows take us to other circles in the diagram. Finally, when the process is done, it exits to the circle labelled "finished". This circle being doubled indicates that this is where the life of the process ends.

We can use finite automata to model simple computations that can only be in a single discrete mode at once and that don't require us to remember or store any information. The circles, or *states*, of the finite automaton correspond to the current mode of computation we're in. For example, did we just begin the computation, or are we midway through, or have we almost wrapped up? The arrows, or *transitions*, of the finite automaton take us between modes, depending on the arrow's label. Thus, if we have a running process, then interrupting that process will put us in a different mode than telling the process to wait for some event to occur.

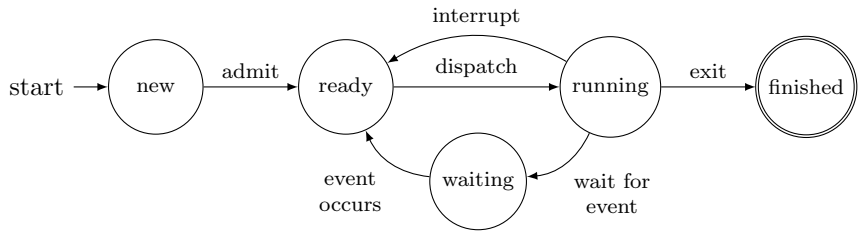


Figure 1.1. How a computer schedules processes.

Formally speaking, a finite automaton is just a 5-tuple.

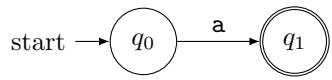
Definition 1.8 (Finite automaton)

A finite automaton is a tuple $(Q, \Sigma, \delta, q_0, F)$, where

- Q is a finite set of *states*;
- Σ is an *alphabet*;
- $\delta\colon Q \times \Sigma \to Q$ is the *transition function*;
- $q_0 \in Q$ is the *initial* or *start state*; and
- $F \subseteq Q$ is the set of *final* or *accepting states*.

> *Remark.* There are occasions in these notes where some grammatical pedantry is warranted, and here is one such occasion. One should always use "finite automat<u>on</u>" to refer to a single instance of such a model; writing "<u>a</u> finite automata<u>a</u>" is never correct.

We're already familiar with alphabets, and we know a little bit about states and transitions from our process scheduling example. The transition function δ is the mathematical formalization of the arrows in our diagram. Given an ordered pair of state and symbol being read, the transition function tells us which state to go to next. For example, if we had a very simple finite automaton like



then the single transition would be represented by the function $\delta(q_0, \mathtt{a}) = q_1$. If a finite automaton has a large number of transitions, then we can represent each transition concisely in a table format rather than writing each transition out individually.

Note that, since we're dealing with a transition *function*, any pair of state and symbol can map to *at most* one state. This condition ensures that we always make the same transition on the same state/symbol pair.

> *Remark.* Note that transition functions don't always have to behave in this way—just those that map to the state set Q. We'll soon see what happens if we don't enforce such a strict condition on our transition function, but for now, our finite automata will operate in this way.

As was the case in our process scheduling example, some states in our very simple finite automaton have some special flair added to them. The state q_0 has an arrow labelled "start" pointing to it, and the state q_1 has

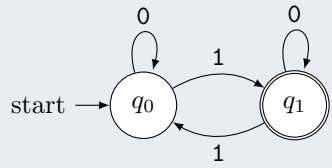
doubled circles. This is how we denote initial and final states in a finite automaton. Initial states have an incoming transition arrow pointing at the state, while final states are double-circled. We typically have just one initial state in a finite automaton, but it's possible to have more than one. On the other hand, we can have as many or as few final states as we want.

Example 1.9

Consider the finite automaton $\mathcal{M}_1 = (Q, \Sigma, \delta, q_0, F)$ where $Q = \{q_0, q_1\}$, $\Sigma = \{\texttt{0}, \texttt{1}\}$, q_0 is the initial state, $F = \{q_1\}$, and δ is defined as follows:

	0	1
q_0	q_0	q_1
q_1	q_1	q_0

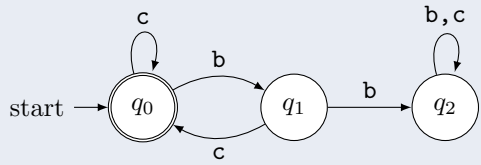
We can draw this finite automaton diagrammatically:



This finite automaton checks whether a binary word has odd parity; that is, whether it contains an odd number of 1s.

Example 1.10

Consider the following diagram of a finite automaton:



This finite automaton checks whether every occurrence of b in an input word is immediately followed by an occurrence of c.

Based on this diagram, we can establish that $Q = \{q_0, q_1, q_2\}$, $\Sigma = \{\texttt{b}, \texttt{c}\}$, q_0 is the initial state, $F = \{q_0\}$, and δ is defined as follows:

	b	c
q_0	q_1	q_0
q_1	q_2	q_0
q_2	q_2	q_2

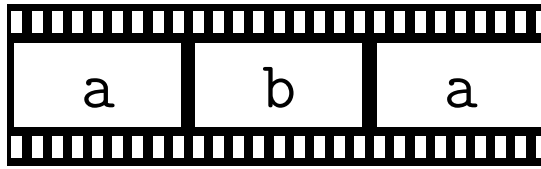


Figure 1.2. A film reel.

1.2.1. Computations and Accepting Computations

Now that we know how to define a finite automaton, what can we do with it? Observe that, in our definition, we took care to specify the alphabet Σ. This alphabet specifies the kinds of *input words* a finite automaton is expecting to receive. Giving an input word to a finite automaton is much like typing `input()` in a Python program or `scanf()` in a C program; it gives the computer something to read and work with.

We can imagine an input word is written on a reel of film, much like in Figure 1.2, where each symbol in the word occupies its own frame. Now, imagine the finite automaton is a film projector, but the rewind button is broken. When we feed the film reel into the projector, the projector can only show one frame at a time. Moreover, since the rewind button is broken, once the projector pulls in the next frame, it can never return to the previous one. This is essentially how a finite automaton processes its input: starting with the first symbol of the input word, the finite automaton reads the symbol, transitions to a state, and then moves to the next symbol.

Once the finite automaton reaches the end of its input word and has no more symbols left to read, it must make a decision. Its decision depends entirely on the state it finds itself in at the moment it reaches the end of the word. If the finite automaton is in a final state when it runs out of symbols, then it *accepts* the word. Otherwise, the finite automaton must be in a non-final state, and it therefore *rejects* the word.

Going one step further, we can precisely define what it means for a finite automaton to accept an input word by introducing the notion of an *accepting computation*. An accepting computation is akin to a set of steps showing us every state a finite automaton enters from the moment it starts reading its input word to the moment it accepts its input. We don't need anything new to define this; we already have all the machinery we need.

Definition 1.11 (Accepting computation of a finite automaton)

Let $\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$ be a finite automaton, and let $w = w_0 w_1 \dots w_{n-1}$ be an input word of length n where $w_0, w_1, \dots w_{n-1} \in \Sigma$. The finite

automaton $\mathcal{M}$ accepts the input word w if there exists a sequence of states $r_0, r_1, \ldots, r_n \in Q$ satisfying the following conditions:

1. $r_0 = q_0$;
2. $\delta(r_i, w_i) = r_{i+1}$ for all $0 \leq i \leq (n-1)$; and
3. $r_n \in F$.

In other words, the computation of a finite automaton must satisfy three conditions in order to be considered an accepting computation: it must start in the initial state, every subsequent state must be reachable by the transition function in one computation step after reading one symbol, and it must end in the final state.

1.2.2. Language of a Finite Automaton

The set of all input words that a finite automaton $\mathcal{M}$ accepts is called the *language* of the finite automaton, denoted $L(\mathcal{M})$, and it's just like any other language: it consists of words over some alphabet Σ. If a finite automaton $\mathcal{M}$ accepts (or *recognizes*) a language A, then $L(\mathcal{M}) = A$. Note that, although a finite automaton can accept possibly many input words, it can only recognize *one* language.

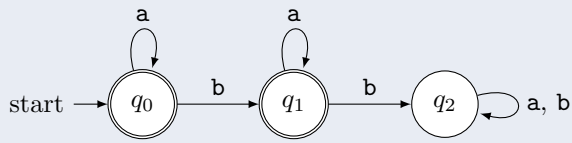
> *Remark.* For clarity's sake, here the word "accept" will be reserved for input words given to a finite automaton, while the word "recognize" will be used to refer to the language of a finite automaton. Both words essentially mean the same thing: the finite automaton has given us a positive answer. Unfortunately, many authors and textbooks use these words interchangeably.

Example 1.12

Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, and consider the language

$$L_{|w|_\mathtt{b} \leq 1} = \{w \mid w \text{ contains at most one occurrence of the symbol } \mathtt{b}\}.$$

This language is recognized by the following automaton:



If the input word w contains zero $\mathtt{b}$s, then the finite automaton will remain in the final state q_0. Likewise, if w contains one $\mathtt{b}$, then the

finite automaton will enter and remain in the final state q_1. Only if w contains two or more bs does the finite automaton enter the state q_2, where it becomes "stuck" and can no longer accept the input word.

Example 1.13

A finite automaton with no final states cannot accept any words, but it is still able to recognize one language: the empty language $\emptyset$. This is because the language of input words accepted by the finite automaton is empty!

As a matter of notation, we will refer to the class of languages recognized by *some* finite automaton by the abbreviation DFA. (What does the D mean? We'll find out in the next section...)

1.2.3. Nondeterminism

Remember how, when we were discussing the transition function earlier, we mandated a condition that any pair of state and symbol must map to *at most* one state? This condition ensured that if we gave the same input word to the same finite automaton, we would end up with the same result. This is known as *deterministic* computation. (And now you know what the D in DFA stands for!)

While determinism isn't inherently a bad thing, it can unfortunately make our job harder if we're trying to construct a finite automaton that recognizes certain "difficult" languages. For example, suppose we wanted to construct a deterministic finite automaton that recognizes the language of words over the alphabet $\Sigma = \{0, 1\}$ where the third-from-last symbol is 0. This finite automaton should accept input words like 011, 10010, and 1010001010011000, but it should reject input words like 110 or 01. Sounds easy to do, right? After all, we really just need to check one symbol: the symbol in the third-from-last position. As it turns out, however, the deterministic finite automaton in question ends up being rather complicated—just take a look at Figure 1.3. Keep in mind also that this deterministic finite automaton *only* works for input words where the third-from-last symbol is 0. If we wanted to, say, check the fourth-from-last symbol, we would need to construct a whole new finite automaton—and as Figure 1.4 illustrates, this finite automaton would have *twice as many* states as our previous one!

So, how do we make our job easier and our finite automata smaller? We get rid of the determinism condition. Specifically, we allow for state/symbol pairs to map to one *or more* states. We can preserve the "function" part of our transition function by mapping each state/symbol pair not to multiple different states individually, but rather to a subset of the state set Q.

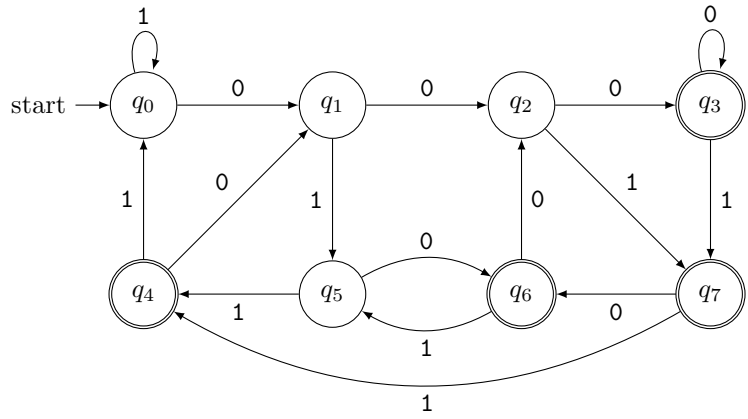


Figure 1.3. A deterministic finite automaton accepting words whose third-from-last symbol is 0.

If we get rid of the determinism condition, then the finite automaton can, in a sense, "guess" which step to take at certain points in the computation. If, in a given state, there is more than one transition out of that state on the same symbol, then the finite automaton has multiple options for which transition it can take. As you might have figured, this property is called *nondeterminism*, and the definition of a nondeterministic finite automaton is nearly identical to our earlier definition of a deterministic finite automaton.

Definition 1.14 (Nondeterministic finite automaton)

A nondeterministic finite automaton is a tuple $(Q, \Sigma, \delta, q_0, F)$, where

- Q is a finite set of states;
- Σ is an alphabet;
- $\delta\colon Q \times \Sigma \to \mathcal{P}(Q)$ is the transition function;
- $q_0 \in Q$ is the initial or start state; and
- $F \subseteq Q$ is the set of final or accepting states.

Following our earlier comment, the only change we had to make is in the transition function: we now map to the power set $\mathcal{P}(Q)$ instead of the state set Q. The element of the power set being mapped to is exactly the subset of states that the nondeterministic finite automaton can transition to from its current state and on its current symbol.

As an illustration of how nondeterminism can simplify the finite automata

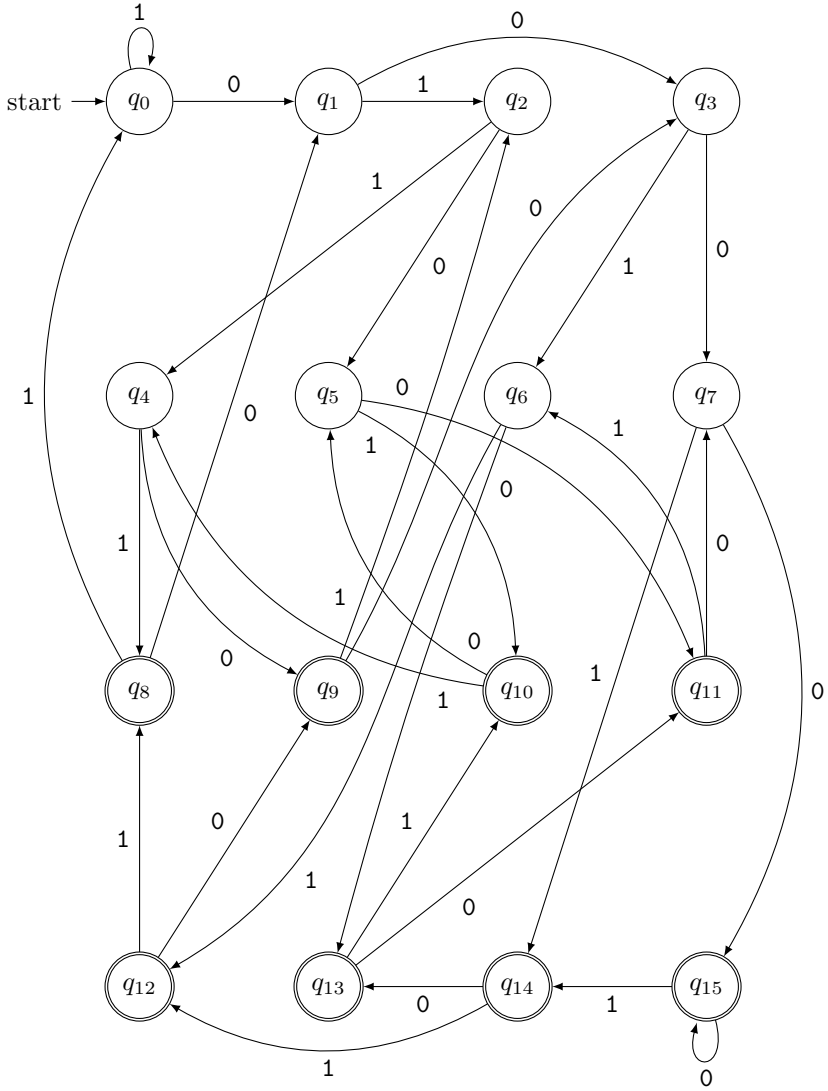


Figure 1.4. A deterministic finite automaton accepting words whose fourth-from-last symbol is 0.

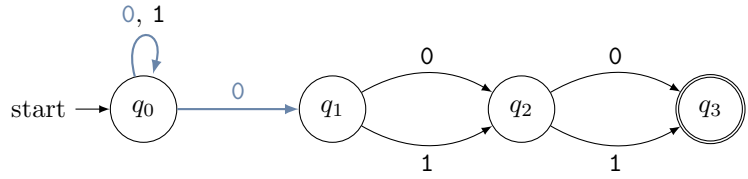


Figure 1.5. A nondeterministic finite automaton accepting words whose third-from-last symbol is 0. Observe the two transitions leaving state q_0, both labelled by the symbol 0.

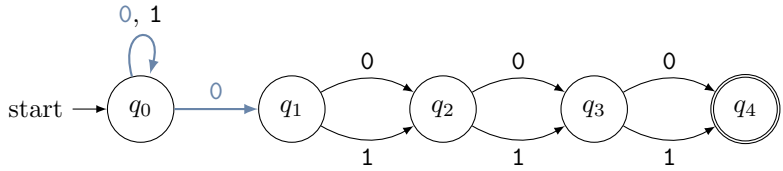


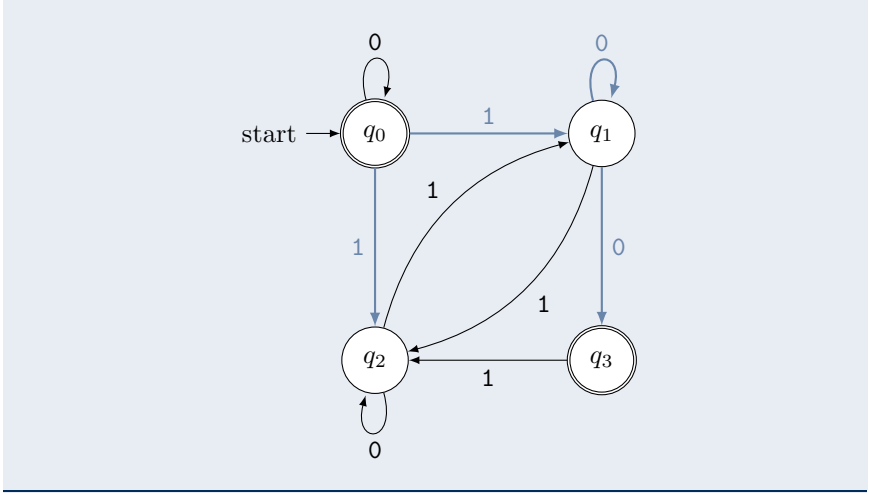
Figure 1.6. A nondeterministic finite automaton accepting words whose fourth-from-last symbol is 0. Contrast the size of this finite automaton to the one depicted in Figure 1.4.

we construct, let's bring back our example of the language of words whose third-from-last symbol is 0. The nondeterministic version of the finite automaton recognizing this language is illustrated in Figure 1.5. Here, the state q_0 is doing double duty: not only is it reading all of the symbols in the input word up to the third-from-last symbol, but it's also checking that the third-from-last symbol is in fact 0. If it is, then we transition from state q_0 to state q_1, and the remaining states simply read the last two symbols, whatever they may be. Likewise, Figure 1.6 depicts a similar construction for the "fourth-from-last" language.

The nondeterminism in these machines is limited to state q_0, where we have two outgoing transitions on the same symbol 0: one transition loops back to the same state q_0, while the other transition takes us to state q_1. We can represent this with the transition function by writing $\delta(q_0, 0) = \{q_0, q_1\}$, and this abides by our definition since $\{q_0, q_1\} \in \mathcal{P}(Q)$.

Example 1.15

The following finite automaton is nondeterministic, because states q_0 and q_1 each have more than one outgoing transition on the same symbol:



A nondeterministic finite automaton accepts an input word in exactly the same way as a deterministic finite automaton: if the finite automaton is in a final state and there are no more symbols of the input word left to read, then the input word is accepted. If not, then the input word is rejected. We will refer to the class of languages recognized by *some* nondeterministic finite automaton by the abbreviation NFA.

The computation of a nondeterministic finite automaton, however, is slightly different than in the deterministic case. Since the finite automaton can take potentially many transitions from one state/symbol pair, at such a point in the computation, the finite automaton "splits up" and runs multiple copies of itself in parallel. If we were to visualize such a computation, we would obtain a diagram that resembles a tree. In fact, such a visualization is called a *computation tree*!

In each branch of the computation tree, the corresponding copy of the finite automaton continues its computation until it either reaches the end of the input word or finds itself with no more transitions to follow, which could happen if the finite automaton reads a symbol in a state with no outgoing transition on that symbol. If there are no transitions to follow, that branch dies while the remaining branches continue with their computations. Similarly, if there are no symbols left to read in the input word and that copy of the finite automaton isn't in a final state, that branch dies.

A computation of a nondeterministic finite automaton is therefore accepting only if there exists at least one branch of the computation where the finite automaton is in a final state after reading every symbol of the input word. Just as we did before, we can formalize the notion of an accepting computation for nondeterministic finite automata; the only change we need to make is in the second condition, to account for the change we made to

render the transition function nondeterministic.

Definition 1.16 (Accepting computation of a nondeterministic finite automaton)

Let $\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$ be a nondeterministic finite automaton, and let $w = w_0 w_1 \dots w_{n-1}$ be an input word of length n where $w_0, w_1, \dots w_{n-1} \in \Sigma$. The finite automaton $\mathcal{M}$ accepts the input word w if there exists a sequence of states $r_0, r_1, \dots, r_n \in Q$ satisfying the following conditions:

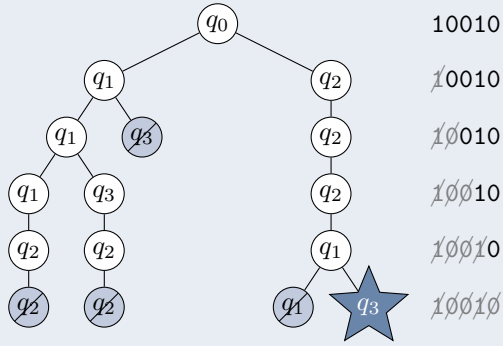
1. $r_0 = q_0$;
2. $r_{i+1} \in \delta(r_i, w_i)$ for all $0 \leq i \leq (n-1)$; and
3. $r_n \in F$.

Observe that, compared to Definition 1.11, we substituted inclusion for strict equality in the second condition—this is because the transition function no longer needs to take us *exactly* to state r_{i+1}. Rather, state r_{i+1} needs only to be in the subset mapped to by the transition function.

Example 1.17

Recall the nondeterministic finite automaton from Example 1.15. Does this automaton accept the input word 10010? Let's check by drawing the computation tree.

Each vertex indicates the current state of the finite automaton at a given point in the computation, and the symbols remaining in the input word at that point are listed on the right. A crossed-out vertex denotes a rejecting computation, while the starred vertex denotes an accepting computation.



Since there exists at least one branch of the computation tree where the

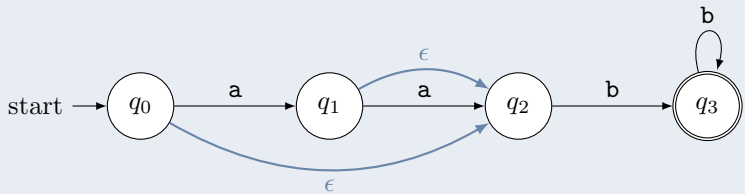
finite automaton is in a final state after reading the entire input word, the finite automaton accepts the word `10010`.

1.2.4. Epsilon Transitions

Going one step further, we can take a nondeterministic finite automaton and modify it so that it can transition not just after reading a symbol, but *whenever it wants*. If a certain special transition called an *epsilon transition* exists between two states q_i and q_j, a finite automaton in state q_i can immediately transition to state q_j without reading the next symbol of the input word. We call such a model a nondeterministic finite automaton *with epsilon transitions*, and the class of languages recognized by this model is denoted by ϵ-NFA.

Example 1.18

The following nondeterministic finite automaton uses epsilon transitions:



This finite automaton accepts all input words starting with zero, one, or two `a`s followed by at least one `b`.

Example 1.19

The nondeterministic finite automaton in Figure 1.7 recognizes the language of signed and unsigned floating-point numbers. Some words in this language include `365.25E+2`, `-10E40`, `+2.5`, and `42E-1`. The epsilon transitions in this finite automaton allow for words to omit the decimal portion of the number, the sign in the exponent, or both.

Note that adding an epsilon transition to a deterministic finite automaton inherently makes it nondeterministic. This is because we've given the finite automaton the option to transition between two states with or without reading a symbol. There cannot exist a "deterministic finite automaton with epsilon transitions".

We won't spend too much time discussing further details of nondeterministic finite automata with epsilon transitions, since the model is almost

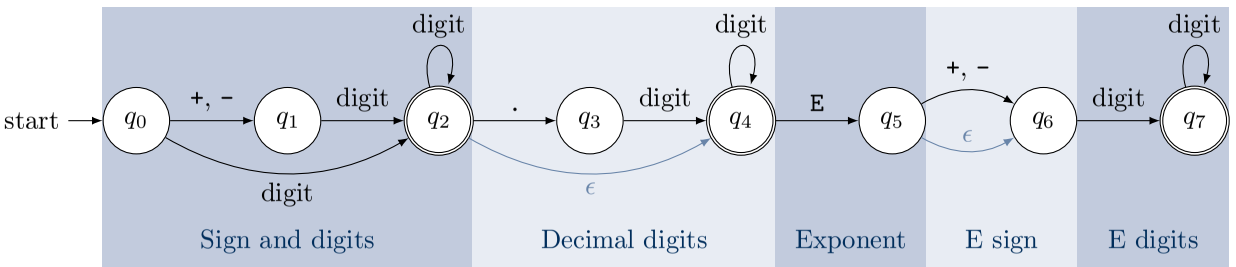


Figure 1.7. A nondeterministic finite automaton with epsilon transitions that recognizes the language of floating-point numbers.

identical to the usual nondeterministic finite automaton model. However, we mention it here because, as we will later see, epsilon transitions can make certain constructions and proofs much easier for us.

1.3. EQUIVALENCE OF MODELS

By now, we've learned about a handful of different models of computation: regular expressions, deterministic finite automata, nondeterministic finite automata, and nondeterministic finite automata with epsilon transitions. Regular expressions give us a textual, language-oriented way of reasoning about regular languages, while finite automata allow us to think graphically in terms of machines. While these two approaches may seem far apart, there actually isn't as much difference between them as one might think.

Let's focus on finite automata for a moment. Going from deterministic to nondeterministic models, we saw that we can construct finite automata that both recognize the same language and are easier to understand—for instance, by virtue of having fewer states or transitions. By introducing epsilon transitions, we learned that we don't even necessarily need to read symbols in order to transition from one state to another.

It seems that this ongoing weakening of conditions keeps giving us models that can "do more". You may be surprised to learn, however, that all of these models of computation are equivalent in terms of the languages they can recognize! No matter what flavour of finite automaton we have, we don't actually gain any additional recognition power.

We will prove this automaton equivalence in two steps. First, we will devise a procedure to convert from a nondeterministic finite automaton with epsilon transitions to one without. Afterward, we will see how to convert from a nondeterministic finite automaton to a deterministic finite automaton.

1.3.1. ϵ-NFA = NFA

In our first procedure, we will use the notion of *epsilon closure* to remove epsilon transitions from a nondeterministic finite automaton. The epsilon closure of a state q is the set of states where there exists some sequence of epsilon transitions from q to that state. Note that the epsilon closure of q always includes q itself.

Theorem 1.20

Given a nondeterministic finite automaton with epsilon transitions $\mathcal{M}$, we can convert it to a nondeterministic finite automaton $\mathcal{M}'$ without

epsilon transitions.

Proof. Let $\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$ be a nondeterministic finite automaton with epsilon transitions. We will construct an equivalent nondeterministic finite automaton $\mathcal{M}' = (Q', \Sigma, \delta', q_0', F')$ without epsilon transitions in the following way:

1. Take Q' to be the original state set Q, and remove all states having only epsilon transitions *to* that state. The initial state is not removed, so take $q_0' = q_0$. All final states in $\mathcal{M}$ remain final states in $\mathcal{M}'$ unless they were removed.

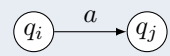
2. Take δ' to be the original transition function δ, but with all epsilon transitions removed. For all states removed in the previous step, also remove all transitions *from* that state.

3. Add new transitions to the transition function δ' as follows:

   - If there exists a "chain" of transitions in $\mathcal{M}$ beginning at a state q_i and ending at a state q_j, where all but the last transition is an epsilon transition and the last transition is on some symbol $a \in \Sigma$,

     then replace this "chain" in $\mathcal{M}'$ with a single transition on a between q_i and q_j.

   - If there exists a "chain" of epsilon transitions in $\mathcal{M}$ beginning at a state q_i and ending at a final state $q_f \in F$,

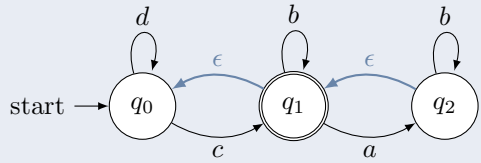
     then remove this "chain" from $\mathcal{M}'$ and make q_i a final state.

In this way, we have constructed a nondeterministic finite automaton without epsilon transitions recognizing the same language as the original finite automaton. ■

Example 1.21

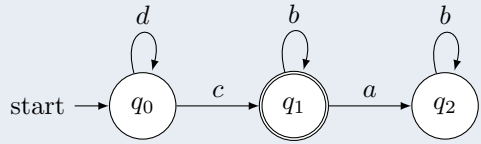
Consider the following nondeterministic finite automaton with epsilon transitions highlighted:



We will use our construction to convert this to a nondeterministic finite automaton without epsilon transitions.

1. First, we take our state set Q' and our initial state q_0'. Since there are no states in this finite automaton having *only* incoming epsilon transitions, we don't need to remove any states.

2. Next, we take our transition function δ' with all epsilon transitions removed. We don't need to remove any other transitions from removed states, since we had no such states in the previous step.

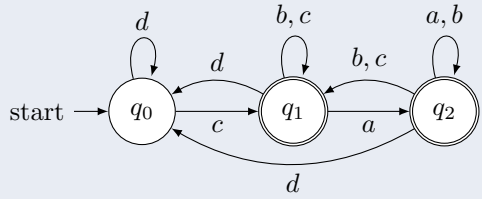


3. Now, we add new transitions to δ' by considering any "chains" in the original finite automaton:

   - For epsilon transition chains ending in a transition on a symbol, we have the following:
     - $q_1 \xrightarrow{\epsilon} q_0 \xrightarrow{d} q_0$ is replaced by $q_1 \xrightarrow{d} q_0$;
     - $q_1 \xrightarrow{\epsilon} q_0 \xrightarrow{c} q_1$ is replaced by $q_1 \xrightarrow{c} q_1$;
     - $q_2 \xrightarrow{\epsilon} q_1 \xrightarrow{b} q_1$ is replaced by $q_2 \xrightarrow{b} q_1$;
     - $q_2 \xrightarrow{\epsilon} q_1 \xrightarrow{a} q_2$ is replaced by $q_2 \xrightarrow{a} q_2$;
     - $q_2 \xrightarrow{\epsilon} q_1 \xrightarrow{\epsilon} q_0 \xrightarrow{d} q_0$ is replaced by $q_2 \xrightarrow{d} q_0$; and
     - $q_2 \xrightarrow{\epsilon} q_1 \xrightarrow{\epsilon} q_0 \xrightarrow{c} q_1$ is replaced by $q_2 \xrightarrow{c} q_1$.

- For epsilon transition chains ending at a final state, we have the following:
  - $q_2 \xrightarrow{\epsilon} q_1$, so state q_2 becomes a final state.

Adding these transitions and final states produces our nondeterministic finite automaton without epsilon transitions:



1.3.2. NFA = DFA

In our next procedure, we will learn how to "simulate" nondeterminism in a deterministic finite automaton. Recall that, in a nondeterministic finite automaton, the transition function maps state/symbol pairs to an element of $\mathcal{P}(Q)$. We can get around the issue of having multiple transitions from one state on the same symbol not by changing our transitions, but by changing our set of states: we simply need to create one state corresponding to each element of $\mathcal{P}(Q)$!

Theorem 1.22

Given a nondeterministic finite automaton $\mathcal{N}$, we can convert it to a deterministic finite automaton $\mathcal{N}'$.

Proof. Let $\mathcal{N} = (Q, \Sigma, \delta, q_0, F)$ be a nondeterministic finite automaton. We assume that $\mathcal{N}$ contains no epsilon transitions; if it does, then use the construction of Theorem 1.20 to remove the epsilon transitions.

We will construct a deterministic finite automaton $\mathcal{N}' = (Q', \Sigma, \delta', q_0', F')$ in the following way:

1. Take $Q' = \mathcal{P}(Q)$; that is, each state of $\mathcal{N}'$ corresponds to a subset of states of $\mathcal{N}$. Note that our deterministic finite automaton may not need to use all of these states; usually, we omit any inaccessible states to make our diagram easier to follow.

2. For each $q' \in Q'$ and $a \in \Sigma$, take

$$\delta'(q', a) = \{q \in Q \mid q \in \delta(s, a) \text{ for some } s \in q'\}.$$

This is perhaps the most difficult step of the construction. Remember that each state q' of $\mathcal{N}'$ corresponds to a *subset* of states of $\mathcal{N}$. Thus, when we read a symbol a in state q' of $\mathcal{N}'$, the transition function δ' takes us to the state in $\mathcal{N}'$ that corresponds to whatever *subset* of states q of $\mathcal{N}$ we could have transitioned to upon reading a in some previous state s of $\mathcal{N}$.

3. Take $q_0' = \{q_0\}$; that is, the initial state of $\mathcal{N}'$ corresponds to the subset containing only the initial state of $\mathcal{N}$.
4. Take $F' = \{q' \in Q' \mid q'$ corresponds to a subset containing at least one final state of $\mathcal{N}\}$. In this way, $\mathcal{N}'$ accepts only if $\mathcal{N}$ would be in a final state at the same point in its computation.

In this way, we have constructed a deterministic finite automaton recognizing the same language as the original finite automaton. ■

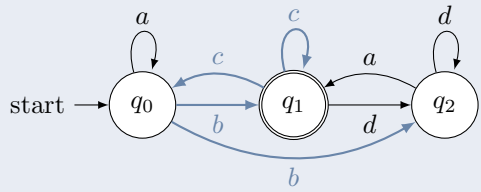
The procedure allowing us to convert from nondeterministic to deterministic finite automata is known as the *subset construction*, because each state of our deterministic finite automaton corresponds to a subset of states from the original nondeterministic finite automaton.

Step 2 of the subset construction procedure is the most involved step. Fortunately, we can obtain the transition function of our deterministic finite automaton $\mathcal{N}'$ using a tabular method via the following steps:

1. Construct a table where the rows are the states of $\mathcal{N}$ and the columns are the symbols of the alphabet Σ.
2. For each state q_i and symbol a, write the set of states mapped to by $\delta(q_i, a)$ in the corresponding row/column entry.
3. After all entries are filled, take all sets of states listed in the table that don't yet have their own row, and create a new row corresponding to that set of states.
4. Repeat steps 2 and 3 until no new rows can be added to the table.

Example 1.23

Consider the following nondeterministic finite automaton, where all nondeterministic transitions from a state are highlighted:



We will use our tabular construction method to obtain the transition function of our desired deterministic finite automaton. Our initial table looks like the following:

	a	b	c	d
q_0				
q_1				
q_2				

We fill in the initial table entries by consulting the transition function of $\mathcal{N}$, where — denotes no transition:

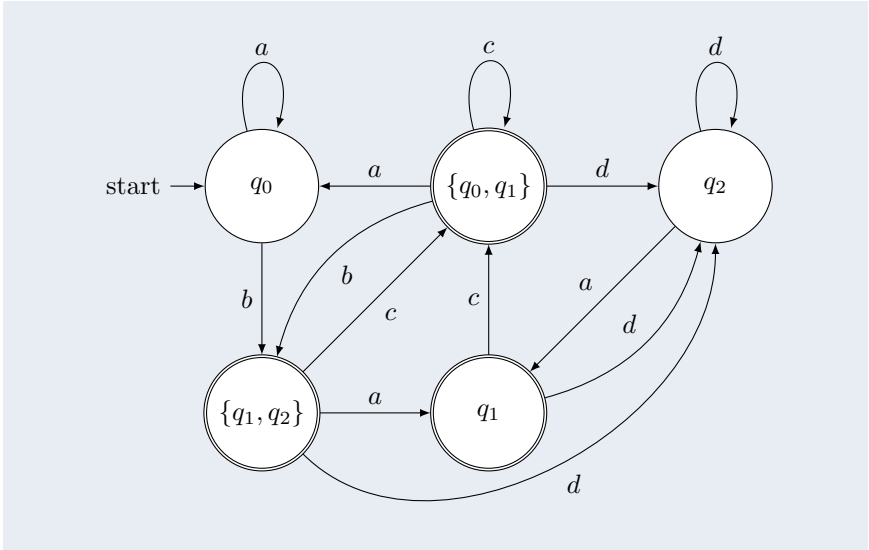
	a	b	c	d
q_0	q_0	$\{q_1, q_2\}$	—	—
q_1	—	—	$\{q_0, q_1\}$	q_2
q_2	q_1	—	—	q_2

Note that there are now two entries in our table without corresponding rows: $\{q_0, q_1\}$ and $\{q_1, q_2\}$. We proceed to add these entries as rows to our table and we fill in the entries for these new rows:

	a	b	c	d
q_0	q_0	$\{q_1, q_2\}$	—	—
q_1	—	—	$\{q_0, q_1\}$	q_2
q_2	q_1	—	—	q_2
$\{q_0, q_1\}$	q_0	$\{q_1, q_2\}$	$\{q_0, q_1\}$	q_2
$\{q_1, q_2\}$	q_1	—	$\{q_0, q_1\}$	q_2

After filling in these new entries, we find that all entries now have corresponding rows, so our table construction is complete. We can now use this table to construct our deterministic finite automaton! Each row of the table corresponds to an accessible state of our deterministic finite automaton, and the table itself specifies our transition function.

Our resultant deterministic finite automaton is the following:



Note that we don't need to come up with procedures for the other directions of conversion: a deterministic finite automaton is a "nondeterministic finite automaton" that doesn't use nondeterminism, and a nondeterministic finite automaton is a "nondeterministic finite automaton with epsilon transitions" that doesn't use any epsilon transitions. Therefore, we can convert in any direction between all three of our finite automaton models, and so we conclude that all of our finite automaton models are equivalent in terms of recognition power.

1.3.3. DFA = RE

Let's now turn back to regular expressions. Since regular expressions are entirely symbol-based, it might be easier for us in some cases to represent a language using a regular expression. In other cases, it might be easier for us to directly construct a finite automaton that recognizes the language. However, is it always the case that, if we can do one, we can also do the other?

We now know that all models of finite automata are equivalent in terms of their recognition power, so all that remains is for us to discover how we can bring regular expressions under this same umbrella. For this last step, we will devise a procedure—actually, two procedures—that allows us to convert a deterministic finite automaton into a regular expression and vice versa.

One direction of our procedure, taking us from finite automaton to regular expression, will systematically eliminate individual states until the automaton is in a simpler standard form. From this standard form, we can

then translate each component of the finite automaton into a component of an equivalent regular expression.

The other direction of our procedure, taking us from regular expression back to finite automaton, will break down a regular expression into its constituent parts and then build up an equivalent finite automaton piece-by-piece.

Theorem 1.24

There exists a deterministic finite automaton $\mathcal{M}$ recognizing a language A if and only if there exists a regular expression $\boldsymbol{r}$ representing the same language A.

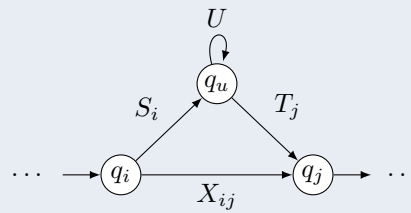
Proof. ($\Rightarrow$): To prove this direction of the statement, we will take a deterministic finite automaton recognizing the language A, and then convert the finite automaton to a regular expression. We will use a *state elimination algorithm* to perform this conversion.

Note that, for this proof only, we will assume that the transitions of our finite automaton can be labelled by regular expressions and not just symbols.

Suppose that we are given a deterministic finite automaton $\mathcal{M}$ such that $L(\mathcal{M}) = A$. Further suppose, without loss of generality, that there exists at most one transition between any two states of $\mathcal{M}$; we can make this assumption since multiple transitions between two states on symbols $a_1, \ldots, a_n$ can be replaced by the single transition on the regular expression $a_1 + \cdots + a_n$.

If $\mathcal{M}$ contains no final state, then $A = \emptyset$ and we are done. Otherwise, if $\mathcal{M}$ contains multiple final states, convert them to non-final states and add epsilon transitions from each former final state to a new single final state. If the initial state is also a final state, make a similar change to the initial state.

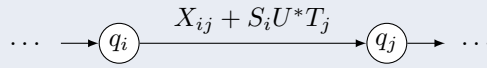
Now, we eliminate all states q_u of $\mathcal{M}$ that are neither initial nor accepting. Suppose that $\mathcal{M}$ contains the following substructure:



In this substructure, all transitions from states $q_i \neq q_u$ to state q_u are labelled by a regular expression S_i; all transitions from state q_u to states

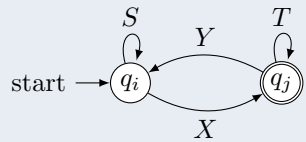
$q_j \neq q_u$ are labelled by a regular expression T_j, and for all such states q_i and q_j the transition between these states is labelled by a regular expression X_{ij}, or $\emptyset$ if no such transition exists. Lastly, any loop from q_u to itself is labelled by a regular expression U, or $\emptyset$ if no loop exists.

We may eliminate state q_u from $\mathcal{M}$ as follows: for each pair of states q_i and q_j, the regular expression X_{ij} on the transition is replaced by $X_{ij} + S_i U^* T_j$.



We then repeat this procedure for all non-initial and non-final states until the only states remaining in the finite automaton are the single initial and final states.

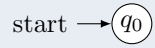
Suppose that, at this stage of the algorithm, our finite automaton is of the following form, where S, T, X, and Y are regular expressions:



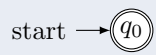
If any of these transitions do not exist, then we simply add them to the finite automaton labelled by $\emptyset$. Then the language recognized by $\mathcal{M}$ is represented by the regular expression $S^* X (T + Y S^* X)^*$.

($\Leftarrow$): To prove this direction of the statement, we will take a regular expression $\boldsymbol{r}$ where $L(\boldsymbol{r}) = A$ and convert it into a nondeterministic finite automaton $\mathcal{M}$ using a construction known as the *McNaughton–Yamada–Thompson algorithm* [McNaughton and Yamada, 1960; Thompson, 1968]. We consider each of the basic regular expressions:

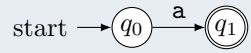
1. If $\boldsymbol{r} = \emptyset$, then $L(\boldsymbol{r}) = \emptyset$ and this language is recognized by the following nondeterministic finite automaton:



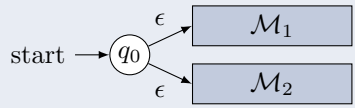
2. If $\boldsymbol{r} = \epsilon$, then $L(\boldsymbol{r}) = \{\epsilon\}$ and this language is recognized by the following nondeterministic finite automaton:



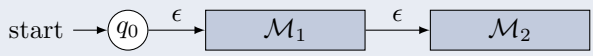
3. If $\boldsymbol{r} = a$ for some $a \in \Sigma$, then $L(\boldsymbol{r}) = \{a\}$ and this language is recognized by the following nondeterministic finite automaton:



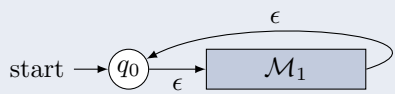
4. If $\boldsymbol{r} = \boldsymbol{r}_1 + \boldsymbol{r}_2$ for some regular expressions $\boldsymbol{r}_1$ and $\boldsymbol{r}_2$, construct two finite automata $\mathcal{M}_1$ and $\mathcal{M}_2$ recognizing $L(\boldsymbol{r}_1)$ and $L(\boldsymbol{r}_2)$, respectively. Then $L(\boldsymbol{r})$ is recognized by the following nondeterministic finite automaton with epsilon transitions:



5. If $\boldsymbol{r} = \boldsymbol{r}_1\boldsymbol{r}_2$ for some regular expressions $\boldsymbol{r}_1$ and $\boldsymbol{r}_2$, construct two finite automata $\mathcal{M}_1$ and $\mathcal{M}_2$ recognizing $L(\boldsymbol{r}_1)$ and $L(\boldsymbol{r}_2)$, respectively. Then $L(\boldsymbol{r})$ is recognized by the following nondeterministic finite automaton with epsilon transitions:



6. If $\boldsymbol{r} = \boldsymbol{r}_1^*$ for some regular expression $\boldsymbol{r}_1$, construct a finite automaton $\mathcal{M}_1$ recognizing $L(\boldsymbol{r}_1)$. Then $L(\boldsymbol{r})$ is recognized by the following nondeterministic finite automaton with epsilon transitions:



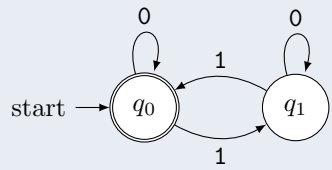
In each case, we can convert the basic regular expression to a nondeterministic finite automaton, and we can then determinize the overall finite automaton using our procedure from Theorem 1.22. We therefore end up with a deterministic finite automaton recognizing the same language represented by the original regular expression. ■

Remark. If the finite automaton constructions in Cases 4, 5, and 6 of the previous proof aren't entirely convincing to you, or feel a bit hand-wavy, worry not. In the following section, we will work through a series of far more precise constructions for each of these automata.

As an illustration of the state elimination algorithm we used in one direction of our proof, let us consider a small example of converting a deterministic finite automaton to a regular expression.

Example 1.25

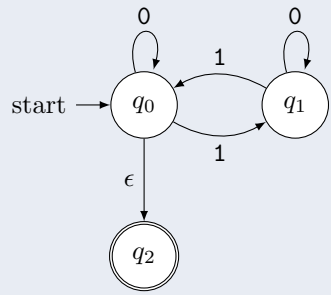
Consider the following deterministic finite automaton $\mathcal{M}$:



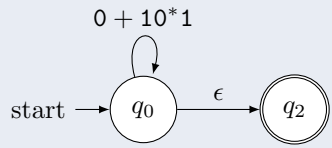
This finite automaton recognizes the language

$$L(\mathcal{M}) = \{w \mid w \text{ contains an even number of } \mathtt{1}\mathtt{s}\}.$$

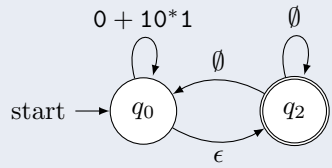
Since the initial state of $\mathcal{M}$ is also a final state, we begin by creating a new final state, converting the initial state to be nonaccepting, and adding an epsilon transition from the initial state to our new final state.



We now use our state elimination algorithm to remove q_1, which is the only state that is neither initial nor accepting. There exists a single transition from q_0 to q_1 and a single transition from q_1 to q_0. Let $S_0 = \mathtt{1}$, $T_0 = \mathtt{1}$, $X_{00} = \mathtt{0}$, and $U = \mathtt{0}$. We can then eliminate the state q_1 by relabelling the loop on q_0 to use the regular expression $X_{00} + S_0U^*T_0 = \mathtt{0} + \mathtt{1}\mathtt{0}^*\mathtt{1}$.



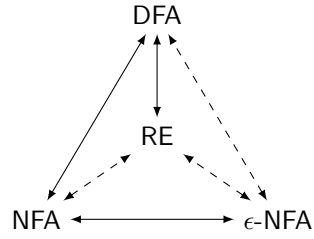
We add the missing transitions to obtain a finite automaton of the form specified in the proof of Theorem 1.24:



Consequently, the regular expression corresponding to this finite automaton is $(\mathtt{0} + \mathtt{10^*1})^*\epsilon(\emptyset + \emptyset(\mathtt{0} + \mathtt{10^*1})^*\epsilon)^*$. Using our rules for operations applied to empty words and empty languages, this regular expression simplifies to $(\mathtt{0} + \mathtt{10^*1})^*$.

1.3.4. Kleene's Theorem

Let's now review all that we've done by drawing a *Scutum Fidei*-esque diagram connecting each of our models of computation:



In our diagram, a solid line indicates that we have a method of directly converting between two models of computation, while a dashed line indicates that we have an indirect method—say, by performing two consecutive conversion steps.

All of our conversions considered apart may seem like nothing more than mechanical procedures or unimportant intermediate steps that we can employ in some larger system. However, taken together as we did in our diagram, these conversions reveal what might reasonably be called the most important theorem in the entire study of regular languages.

Theorem 1.26 (Kleene's theorem)

A language R is regular if it satisfies any of the following equivalent properties:

1. There exists a deterministic finite automaton $\mathcal{M}_{\mathrm{D}}$ such that

$L(\mathcal{M}_{\text{D}}) = R$;

2. There exists a nondeterministic finite automaton $\mathcal{M}_{\text{N}}$ such that $L(\mathcal{M}_{\text{N}}) = R$;
3. There exists a nondeterministic finite automaton with epsilon transitions $\mathcal{M}_{\text{E}}$ such that $L(\mathcal{M}_{\text{E}}) = R$; or
4. There exists a regular expression $\boldsymbol{r}$ such that $L(\boldsymbol{r}) = R$.

Note that we don't need to prove anything here—the proof of Kleene's theorem is baked into the descriptions of each of our conversion procedures!

1.4. CLOSURE PROPERTIES

Another important consideration when we discuss any model of computation is that of *closure properties*, since they allow us to determine whether we can apply certain operations to words or languages while still allowing the same model to accept or recognize the result.

We say that a set S is *closed* under an operation $\circ$ if, given any two elements $a, b \in S$, we have that $a \circ b \in S$ as well. You might be familiar with the notion of closure from elsewhere in mathematics: for example, the set of integers is closed under the operations of addition, subtraction, and multiplication, since for all integers a and b, we know that $a + b$, $a - b$, and $a \times b$ are integers. On the other hand, the set of integers is not closed under the operation of division, since (for example) $1, 2 \in \mathbb{Z}$ but $1/2 \notin \mathbb{Z}$.

We can prove all kinds of closure results for the class of regular languages, but here we will focus only on a few basic operations: the regular operations of union, concatenation, and Kleene star, together with two new set-inspired operations, complement and intersection, which will come in handy later. For each result, we will follow the same general style of proof to establish closure. If the operation $\circ$ is binary, as is the case for union, concatenation, and intersection, then we will take two finite automata $\mathcal{M}$ and $\mathcal{N}$ recognizing languages $L(\mathcal{M})$ and $L(\mathcal{N})$ and directly construct a new finite automaton recognizing the operation language $L(\mathcal{M}) \circ L(\mathcal{N})$. On the other hand, if the operation is unary, as is the case for Kleene star and complement, then we need only take one finite automaton $\mathcal{M}$ recognizing the language $L(\mathcal{M})$ and directly construct a new finite automaton recognizing the operation language $L(\mathcal{M})^{\circ}$.

Union. We begin by considering the union operation. To determine whether some input word belongs to the union of two languages $L(\mathcal{A})$ and $L(\mathcal{B})$, we must check that the word is accepted by either $\mathcal{A}$ or $\mathcal{B}$, or by both. Thus,

we must essentially perform two parallel "subcomputations" for each of these finite automata. This parallelism means we must also incorporate nondeterminism into our computation, since we don't know in advance which of the two finite automata will accept the word.

Theorem 1.27

The class of regular languages is closed under the operation of union.

Proof. Suppose we are given two finite automata, denoted $\mathcal{A} = (Q_{\mathcal{A}}, \Sigma, \delta_{\mathcal{A}}, q_{0_{\mathcal{A}}}, F_{\mathcal{A}})$ and $\mathcal{B} = (Q_{\mathcal{B}}, \Sigma, \delta_{\mathcal{B}}, q_{0_{\mathcal{B}}}, F_{\mathcal{B}})$. We construct a nondeterministic finite automaton with epsilon transitions $\mathcal{C}$ recognizing the language $L(\mathcal{A}) \cup L(\mathcal{B})$ in the following way:

- Take $Q_{\mathcal{C}} = Q_{\mathcal{A}} \cup Q_{\mathcal{B}} \cup \{q_0\}$.
- Take $q_{0_{\mathcal{C}}} = q_0$.
- Take $F_{\mathcal{C}} = F_{\mathcal{A}} \cup F_{\mathcal{B}}$.
- Define $\delta_{\mathcal{C}}$ such that, for all $q \in Q_{\mathcal{C}}$ and for all $a \in \Sigma \cup \{\epsilon\}$,

$$\delta_{\mathcal{C}}(q,a) = \begin{cases} \delta_{\mathcal{A}}(q,a) & \text{if } q \in Q_{\mathcal{A}}; \\ \delta_{\mathcal{B}}(q,a) & \text{if } q \in Q_{\mathcal{B}}; \text{ and} \\ \{q_{0_{\mathcal{A}}}, q_{0_{\mathcal{B}}}\} & \text{if } q = q_0 \text{ and } a = \epsilon. \end{cases}$$

■

The "union" finite automaton $\mathcal{C}$ is depicted in Figure 1.8a. Note that, since we now know that all of our models of finite automata are equivalent to one another (and equivalent to regular expressions), the fact that we used nondeterministic finite automata with epsilon transitions in our proof is irrelevant. We did so mainly because it makes the construction easier on us.

Concatenation. Next, we consider the concatenation operation. To determine whether some input word belongs to the concatenation language $L(\mathcal{A})L(\mathcal{B})$, we again need to perform two "subcomputations" on both finite automata $\mathcal{A}$ and $\mathcal{B}$, but this time in series. The first part of the word should take us to a final state of $\mathcal{A}$, at which point we will jump to $\mathcal{B}$ to read the remaining second part of the word. However, since we don't know where this "jumping point" is within the word, we again need nondeterminism to guess when we have reached a final state of $\mathcal{A}$.

Theorem 1.28

The class of regular languages is closed under the operation of concatenation.

Proof. Suppose we are given two finite automata, denoted $\mathcal{A} = (Q_{\mathcal{A}}, \Sigma, \delta_{\mathcal{A}}, q_{0_{\mathcal{A}}}, F_{\mathcal{A}})$ and $\mathcal{B} = (Q_{\mathcal{B}}, \Sigma, \delta_{\mathcal{B}}, q_{0_{\mathcal{B}}}, F_{\mathcal{B}})$. We construct a nondeterministic finite automaton with epsilon transitions $\mathcal{C}$ recognizing the language $L(\mathcal{A})L(\mathcal{B})$ in the following way:

- Take $Q_{\mathcal{C}} = Q_{\mathcal{A}} \cup Q_{\mathcal{B}}$.
- Take $q_{0_{\mathcal{C}}} = q_{0_{\mathcal{A}}}$.
- Take $F_{\mathcal{C}} = F_{\mathcal{B}}$.
- Define $\delta_{\mathcal{C}}$ such that, for all $q \in Q_{\mathcal{C}}$ and for all $a \in \Sigma \cup \{\epsilon\}$,

$$\delta_{\mathcal{C}}(q,a) = \begin{cases} \delta_{\mathcal{A}}(q,a) & \text{if } q \in Q_{\mathcal{A}} \text{ and } q \notin F_{\mathcal{A}}; \\ \delta_{\mathcal{A}}(q,a) & \text{if } q \in F_{\mathcal{A}} \text{ and } a \neq \epsilon; \\ \delta_{\mathcal{A}}(q,a) \cup \{q_{0_{\mathcal{B}}}\} & \text{if } q \in F_{\mathcal{A}} \text{ and } a = \epsilon; \text{ and} \\ \delta_{\mathcal{B}}(q,a) & \text{if } q \in Q_{\mathcal{B}}. \end{cases}$$

■

The "concatenation" finite automaton $\mathcal{C}$ is depicted in Figure 1.8b. Again, the fact that we used nondeterministic finite automata with epsilon transitions in this proof does not matter, since any regular language model of computation will work equally well.

Kleene Star. For our next result, pertaining to the Kleene star, we consider just one finite automaton instead of two. However, the construction process is similar to that which we just saw for concatenation. Since the Kleene star is essentially repeated concatenation, upon reaching a final state of the finite automaton $\mathcal{A}$, we will jump backward to allow us to cycle through the computation again if we desire.

There is one technicality, though: we can't jump backward directly to the original initial state of $\mathcal{A}$, since if that initial state has a looping transition, we might be able to mistakenly accept words not in the original language. Thus, we will jump backward to a new state, and from there we can transition to the original initial state of $\mathcal{A}$.

Theorem 1.29

The class of regular languages is closed under the operation of Kleene star.

Proof. Suppose we are given a finite automaton, denoted $\mathcal{A} = (Q_{\mathcal{A}}, \Sigma, \delta_{\mathcal{A}}, q_{0_{\mathcal{A}}}, F_{\mathcal{A}})$. We construct a nondeterministic finite automaton with epsilon transitions $\mathcal{A}'$ recognizing the language $L(\mathcal{A})^*$ in the following way:

- Take $Q_{\mathcal{A}'} = Q_{\mathcal{A}} \cup \{q_0\}$.
- Take $q_{0_{\mathcal{A}'}} = q_0$.
- Take $F_{\mathcal{A}'} = \{q_0\}$.
- Define $\delta_{\mathcal{A}'}$ such that, for all $q \in Q_{\mathcal{A}'}$ and for all $a \in \Sigma \cup \{\epsilon\}$,

$$\delta_{\mathcal{A}'}(q,a) = \begin{cases} \delta_{\mathcal{A}}(q,a) & \text{if } q \in Q_{\mathcal{A}} \text{ and } q \notin F_{\mathcal{A}}; \\ \delta_{\mathcal{A}}(q,a) & \text{if } q \in F_{\mathcal{A}} \text{ and } a \neq \epsilon; \\ \delta_{\mathcal{A}}(q,a) \cup \{q_0\} & \text{if } q \in F_{\mathcal{A}} \text{ and } a = \epsilon; \text{ and} \\ \{q_{0_{\mathcal{A}}}\} & \text{if } q = q_0 \text{ and } a = \epsilon. \end{cases}$$

■

The "Kleene star" finite automaton $\mathcal{A}'$ is depicted in Figure 1.8c and, again, the usual disclaimer about nondeterministic finite automata with epsilon transitions applies to this proof.

Complement. Having established closure for each of the regular operations, we can now shift our focus toward other common language operations, and we will begin by considering the operation of *complement*. Much like with sets, the complement of a language L is a language containing all words that do *not* belong to L; that is,

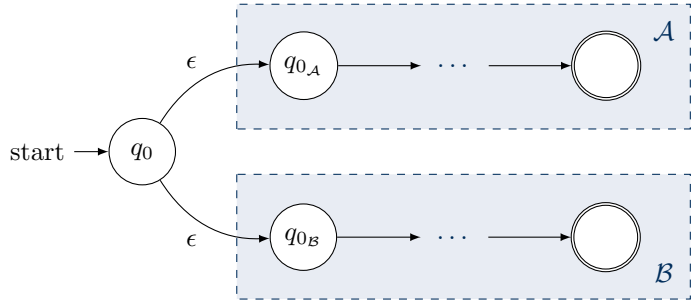
$$\overline{L} = \{w \mid w \notin L\}.$$

Thus, to say that regular languages are closed under complement means that, if we were to take a regular language L and consider all words that do *not* belong to L, the resultant language would also be regular: perhaps a surprising result at first, but one with a rather easy proof!

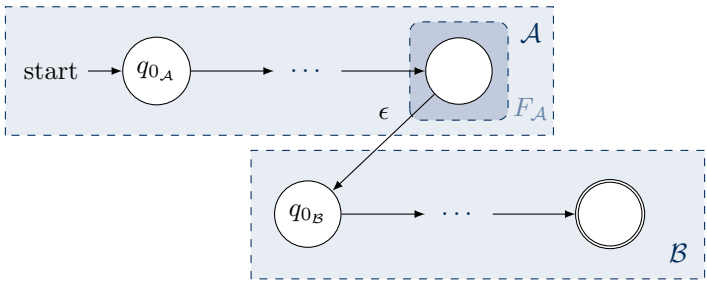
Theorem 1.30

The class of regular languages is closed under the operation of complement.

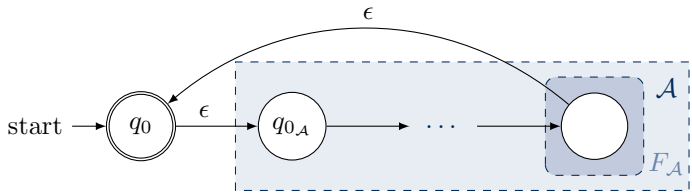
Proof. Suppose we are given a deterministic finite automaton $\mathcal{A} =$



(a) Finite automaton recognizing the union of two languages.



(b) Finite automaton recognizing the concatenation of two languages.



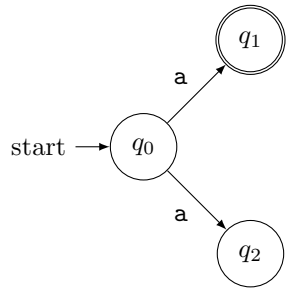
(c) Finite automaton recognizing the Kleene star of a language.

Figure 1.8. Finite automaton constructions for each of the three regular operations.

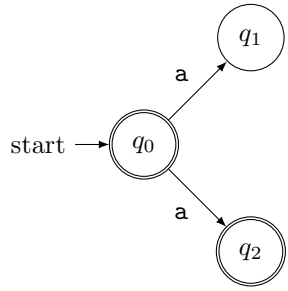
$(Q, \Sigma, \delta, q_0, F)$. The language of this automaton, $L(\mathcal{A})$, consists of all words that take us from the initial state q_0 to a final state in the subset F. Therefore, the complement of $L(\mathcal{A})$ consists of all words that *do not* take us to a final state.

We construct a finite automaton $\mathcal{A}' = (Q, \Sigma, \delta, q_0, F')$ recognizing the language $\overline{L(\mathcal{A})}$ by taking $F' = Q \setminus F$; that is, all non-final states in $\mathcal{A}$ are final states in $\mathcal{A}'$, and vice versa. ■

Take careful note that, unlike in the previous three closure proofs for union, concatenation, and Kleene star, the closure proof for complement requires that we start with a *deterministic* finite automaton. We cannot simply swap the final and non-final states in a nondeterministic finite automaton and expect the construction to go through with no issues. To see why this is the case, consider the following nondeterministic finite automaton:



Observe that this finite automaton recognizes the singleton language $L = \{\texttt{a}\}$, and so the complement of this finite automaton must *not* include the word `a` in its language. However, applying our complement construction directly to the nondeterministic finite automaton produces the following result:



Clearly, this "complement" finite automaton still accepts the word `a`! Thus, in order for us to correctly complement a nondeterministic finite automaton, we must first convert it into its equivalent deterministic form.

Intersection. For our last closure result, we will look at the operation of intersection. Recall that, with the union operation, we had to check whether some input word belonged to *either* of the languages $L(\mathcal{A})$ or $L(\mathcal{B})$. With the intersection operation, on the other hand, we must check whether the input word belongs to *both* languages.

We could establish closure via a nice construction known as a *product automaton*, but this isn't strictly necessary. In fact, we can get this new intersection closure result using a couple of the closure results we have already established! This is all thanks to *De Morgan's laws*, one of which allows us to reformulate intersection in terms of both union and complement:

$$L(\mathcal{A}) \cap L(\mathcal{B}) = \overline{\overline{L(\mathcal{A})} \cup \overline{L(\mathcal{B})}}.$$

Since we already know that the regular languages are closed under union and complement, we get the proof of closure under intersection for free.

Theorem 1.31

The class of regular languages is closed under the operation of intersection.

Proof. Suppose we are given two deterministic finite automata $\mathcal{A}$ and $\mathcal{B}$ recognizing languages $L(\mathcal{A})$ and $L(\mathcal{B})$, respectively. Since $L(\mathcal{A})$ and $L(\mathcal{B})$ are regular, we know that $\overline{L(\mathcal{A})}$ and $\overline{L(\mathcal{B})}$ are regular by closure under complement. We also know that $\overline{L(\mathcal{A})} \cup \overline{L(\mathcal{B})}$ is regular by closure under union. Therefore, $\overline{\overline{L(\mathcal{A})} \cup \overline{L(\mathcal{B})}}$ is regular again by closure under complement, and so $L(\mathcal{A}) \cap L(\mathcal{B})$ is regular. ■

Eventually, I will detail an alternative approach using the so-called product automaton.

1.5. PROVING A LANGUAGE IS NONREGULAR

At this point, it should be evident that finite automata and regular expressions are nice models to use when discussing computation in the abstract. They're easy to define, easy to reason about, and they have a lot of nice properties that we can use in proofs. However, they are not the be-all and end-all of theoretical computer science. (Otherwise, this would be a rather short course!)

Both finite automata and regular expressions suffer the drawback of not having any way to store or recall data. Finite automata don't have any storage mechanism, and regular expressions don't allow for lookback. As we said in the section introducing finite automata, once the finite automaton

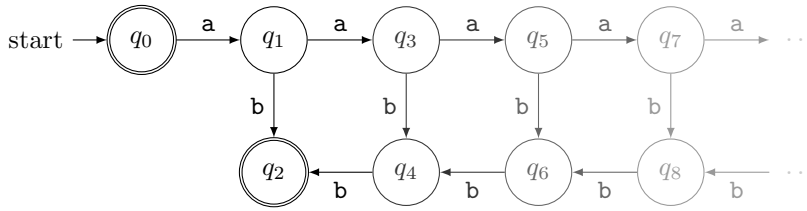


Figure 1.9. A "finite" automaton supposedly recognizing the language $L_{\mathtt{a}=\mathtt{b}}$.

reads a symbol and transitions to a state, it can never return to that symbol. For all intents and purposes, the symbol is lost forever, and the finite automaton doesn't even remember having read it. Likewise, once a regular expression matches a symbol in a word and moves on to the next symbol, it has no way of remembering any previous symbols that were matched.

Naturally, this means that there exist some languages that cannot be recognized by a finite automaton (or, equivalently, represented by a regular expression), and therefore such languages cannot be regular. For instance, this is the canonical example of a language that no finite automaton can recognize:

$$L_{\mathtt{a}=\mathtt{b}} = \{\mathtt{a}^n\mathtt{b}^n \mid n \geq 0\}.$$

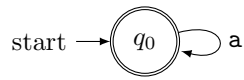
In this language, every word has an equal number of `a`s and `b`s, and all occurrences of `a` appear before the first occurrence of `b`. Some examples of words in this language are `ab`, `aaabbb`, `aaaaaabbbbbb`, and ϵ.

Why can't any finite automaton recognize this language? Because of that word *finite*. A finite automaton consists of a finite number of states, but in order to recognize this language, we would need to add a "chain" consisting of $2n + 1$ states to accept the word $\mathtt{a}^n\mathtt{b}^n$ for every $n \geq 0$; such a construction is illustrated in Figure 1.9. Since n has no upper bound, we would need an infinite number of states! No finite automaton can recognize this language, because no finite automaton has a way of keeping track of the value n or counting the symbols using only a finite number of states.

However, we can't totally rely on the claim that a finite automaton is incapable of recognizing a language if it has to count symbols. For instance, consider the language

$$L_{\mathtt{a}} = \{\mathtt{a}^n \mid n \geq 0\}.$$

This language contains an infinite number of words: one word for each $n \geq 0$, exactly like in $L_{\mathtt{a}=\mathtt{b}}$. But it's easy for a finite automaton to recognize $L_{\mathtt{a}}$, and using only one state!



Thus, it should hopefully be clear that we need to take a slightly more intricate approach in order to prove a language is not regular. There are many more nonregular languages than there are regular languages, so instead of focusing on some sort of property that a nonregular language might have, let's instead find a property every regular language must have. We can then prove a language is nonregular by showing that the language *doesn't* have that property.

1.5.1. The Pumping Lemma for Regular Languages

The property of regular languages that we will make use of is the following: for every regular language, if we take a word in the language of sufficient length, then we can repeat (or *pump*) a middle portion of that word an arbitrary number of times and produce a new word that belongs to the same regular language. This fact is known, appropriately enough, as the *pumping lemma* for regular languages.

Lemma 1.32 (Pumping lemma for regular languages)

For all regular languages L, there exists $p \geq 1$ where, for all $w \in L$ with $|w| \geq p$, there exists a decomposition of w into three parts $w = xyz$ such that

1. $|y| > 0$;
2. $|xy| \leq p$; and
3. for all $i \geq 0$, $xy^i z \in L$.

Clearly, the pumping lemma contains a lot of notation and terminology to take in at once—not to mention four alternating quantifiers in a row! Let's take a closer look at the lemma from three different perspectives.

An Informal Description. We'll begin by breaking the pumping lemma down piece-by-piece to see what it tells us.

- ***For all regular languages L,***
  We can take any regular language L, and it will satisfy the pumping lemma.
- ***there exists $p \geq 1$***
  Depending on the language L, the pumping lemma claims that there exists a constant p for that language. We call p the *pumping constant*.

(If you're curious, p is the number of states in the finite automaton recognizing L.)

- ***where, for all $w \in L$ with $|w| \geq p$,***
  We can take any word from L with length at least p, and it will satisfy the pumping lemma.
- ***there exists a decomposition of w into three parts $w = xyz$***
  Depending on the word w, the pumping lemma claims that w can be split into three parts, labelled x, y, and z. The y part is what we will use to do the pumping; the x and z parts are just the start and end parts of w that don't get pumped.
- ***such that 1. $|y| > 0$;***
  This condition ensures that the y part of w is nonempty, so that we have something to pump.
- ***2. $|xy| \leq p$;***
  This condition ensures that there exists some state in the finite automaton recognizing L that is visited more than once, and furthermore, we will visit that state during the computation before we finish reading the part y.

  (This condition is essentially an application of the pigeonhole principle.)
- ***and 3. for all $i \geq 0$, $xy^iz \in L$.***
  This is the actual pumping part of the pumping lemma. This condition ensures that, no matter how many copies of the y part we include in our word (even zero copies), the resulting word will still belong to the language.

A Formal Proof. Now that we have a greater understanding of what the pumping lemma says, let's take a look at the proof of the lemma. Remember, the property of all regular languages that we're relying on is that if we take a word of sufficient length from the language, then we can pump a middle portion of that word arbitrarily many times and always obtain a word that still belongs to the language. This means that if we consider a finite automaton recognizing that language, there must exist a loop somewhere within that automaton.

Proof of Lemma 1.32. Let $\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$ be a deterministic finite automaton recognizing the language L, and let p denote the number of states of $\mathcal{M}$.

Take a word $w = w_1 w_2 \dots w_n$ of length n from L, where $n \geq p$, and let $r_1, \dots, r_{n+1}$ be the accepting computation of $\mathcal{M}$ on w. Specifically, let $r_{i+1} = \delta(r_i, w_i)$ for all $1 \leq i \leq n$. Clearly, this accepting computation has length $n + 1 \geq p + 1$.

By the pigeonhole principle, there must exist at least two states in the first $p + 1$ states of the accepting computation that are the same. Say that the first occurrence of the same state is r_j and the second occurrence is r_ℓ. Since r_ℓ occurs within the first $p + 1$ states of the accepting computation, we know that $\ell \leq p + 1$.

Decompose the word w into parts $x = w_1 \dots w_{j-1}$, $y = w_j \dots w_{\ell-1}$, and $z = w_\ell \dots w_n$. As the part x is read, $\mathcal{M}$ transitions from state r_1 to state r_j. Likewise, as y is read, $\mathcal{M}$ transitions from r_j to r_j, and as z is read, $\mathcal{M}$ transitions from r_j to r_{n+1}. Since we are considering an accepting computation, r_{n+1} is a final state, and so $\mathcal{M}$ must accept the word $xy^i z$ for all $i \geq 0$. Moreover, we know that $j \neq \ell$, so $|y| > 0$. Lastly, since $\ell \leq p + 1$, we have that $|xy| \leq p$. Therefore, all three conditions of the pumping lemma are satisfied. ■

Diagrammatically, this proof can be approximated by Figure 1.10. The wavy transition lines in the figure denote some chain of transitions starting at one state and ending at another state, where we don't care about the states in between. From the figure, we can see that all of the states of the finite automaton between r_0 and r_j are used to read the part x, all of the states between r_j and r_{n+1} are used to read the part z, and there exists a loop of states that both starts and ends with r_j that is used to read the part y. We can take this loop as many times as we want while reading the input word, and taking one journey around the loop corresponds to "pumping" the word once.

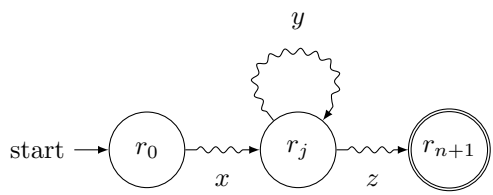


Figure 1.10. The pumping lemma for regular languages, presented diagrammatically.

Rules of the Pumping Lemma Game

1. Your opponent chooses $p \geq 1$ and claims it is the pumping constant for L.
2. You choose a word $w \in L$ with $|w| \geq p$ and claim this word can't be decomposed into parts $w = xyz$ that satisfy the three conditions of the pumping lemma.
3. Your opponent chooses a decomposition $w = xyz$ such that $|y| \geq 0$ and $|xy| \leq p$, satisfying the first two conditions automatically, and claims that this decomposition will also satisfy the third condition.
4. You choose $i \geq 0$ such that $xy^iz \notin L$.

If you complete Step 4, then you win the game!
If you can't find any $i \geq 0$ in Step 4, then you lose the game.
If any of the claims in Steps 1–3 are false, then the person who made the claim loses the game.

Figure 1.11. The pumping lemma game.

A Fun Game. Alternatively, we can think of the pumping lemma as an adversarial game, where we're trying to show that some language L is nonregular while our opponent is trying to show that L is, in fact, regular. If we win the game, then L is nonregular, while if our opponent wins, then L is regular. The rules of this game are given in Figure 1.11, so that you can play it at the next party you attend.

Using the Pumping Lemma. As we noted, every regular language must satisfy the pumping lemma, and so any language that does not satisfy the pumping lemma must not be regular. This means that we can use the lemma to prove a language is nonregular by contradiction: assuming the language were regular, it should satisfy the pumping lemma, but if we can somehow pump a sufficently long word to produce a word that does *not* belong to the language, our assumption of regularity must not hold.

Even though the pumping lemma looks complex, reducing it to a series of steps as we did here reveals that any proof of the nonregularity of a language simply has to follow each of the steps. As a result, all nonregularity proofs tend to share a similar structure.

Let's take a look at an example of a pumping lemma proof using our canonical nonregular language, $L_{\mathtt{a}=\mathtt{b}}$.

Example 1.33

Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, and consider the language $L_{\mathtt{a}=\mathtt{b}} = \{\mathtt{a}^n\mathtt{b}^n \mid n \geq 0\}$. We will use the pumping lemma to show that this language is nonregular.

Assume by way of contradiction that the language is regular, and let p denote the pumping constant given by the pumping lemma. We choose the word $w = \mathtt{a}^p\mathtt{b}^p$. Clearly, $w \in L_{\mathrm{a=b}}$ and $|w| \geq p$. Thus, there exists a decomposition $w = xyz$ satisfying the three conditions of the pumping lemma.

We consider three cases, depending on the contents of the part y of the word w:

1. The part y contains only `a`s. In this case, pumping y once to obtain the word xy^2z results in the word containing more `a`s than `b`s, and so $xy^2z \notin L_{\mathrm{a=b}}$. This violates the third condition of the pumping lemma.

2. The part y contains only `b`s. In this case, since the first p symbols of w are `a`s, we must have that $|xy| > p$. This violates the second condition of the pumping lemma.

3. The part y contains both `a`s and `b`s. Again, in this case, since the first p symbols of w are `a`s, we must have that $|xy| > p$. This violates the second condition of the pumping lemma.

In all cases, one of the conditions of the pumping lemma is violated. As a consequence, the language cannot be regular.

Let's step through each component of this proof. We began by assuming our language was regular. From this assumption, the pumping lemma tells us that there exists some value $p \geq 1$ that we can use in our next step: choosing an appropriate word from the language. We choose a word w that is sufficiently long; that is, of length at least p. (We chose $\mathtt{a}^p\mathtt{b}^p$ here, which makes for a good strategy: try to incorporate the value p into the chosen word in some way.) The pumping lemma then tells us that, since our word w is long enough, there exists some decomposition $w = xyz$ that satisfies the three conditions of the lemma. From here, the remainder of the proof consists of us checking every possible decomposition of w and finding some violated condition for each decomposition.

Two of the most common mistakes when using the pumping lemma are fixing a specific value for p and choosing a specific decomposition $w = xyz$. We must not do either of these! Fixing a specific value for p is not allowed because the statement of the pumping lemma tells us only that *there exists* a value p, not what this value is specifically. Likewise, choosing a specific decomposition is not allowed because the pumping lemma again tells us only that *there exists* a decomposition that satisfies the three conditions. Observe that the example we just saw keeps things as general as possible: it doesn't fix a specific value for p, and it considers all possible decompositions

before arriving at a conclusion.

A language doesn't necessarily have to count symbols in order to be nonregular. Since finite automata don't have any form of storage, they can't remember symbols they read earlier in an input word. This means that finite automata can't recall parts of a word, and so they can't recognize languages like $L_{\text{double}} = \{ww \mid w \in \Sigma^*\}$. Here, we prove that a similar language is nonregular: the language of palindromes, ww^{R}. (The notation w^{R} denotes the *reversal* of the word w.) Palindromes are words that read the same backward as they do forward.

Example 1.34

Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, and consider the language $L_{\text{pal}} = \{ww^{\text{R}} \mid w \in \Sigma^*\}$. We will use the pumping lemma to show that this language is nonregular.

Assume by way of contradiction that the language is regular, and let p denote the pumping constant given by the pumping lemma. We choose the word $w = \mathtt{a}^p\mathtt{bba}^p$. Clearly, $w \in L_{\text{pal}}$ and $|w| \geq p$. Thus, there exists a decomposition $w = xyz$ satisfying the three conditions of the pumping lemma.

Since the second condition of the pumping lemma tells us that $|xy| \leq p$, it must be the case that, in any decomposition, we have $xy = \mathtt{a}^k$ for some $k \leq p$. Consequently, we have $y = \mathtt{a}^\ell$ for some $1 \leq \ell \leq k$.

If we pump y once to obtain the word xy^2z, then we obtain the word $\mathtt{a}^{p+\ell}\mathtt{bba}^p$, which is no longer a palindrome. This violates the third condition of the pumping lemma. As a consequence, the language cannot be regular.

Lastly, recall the third condition of the pumping lemma: for all $i \geq 0$, $xy^iz \in L$. The third condition allows us not only to pump *up* by adding copies of y to the word, but also to pump *down* by removing y from the word. In some cases, pumping down can help us to prove a language is nonregular.

Example 1.35

Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, and consider the language $L_{\mathtt{a}>\mathtt{b}} = \{\mathtt{a}^i\mathtt{b}^j \mid i > j\}$. We will use the pumping lemma to show that this language is nonregular.

Assume by way of contradiction that the language is regular, and let p denote the pumping constant given by the pumping lemma. We choose the word $w = \mathtt{a}^{p+1}\mathtt{b}^p$. Clearly, $w \in L_{\mathtt{a}>\mathtt{b}}$ and $|w| \geq p$. Thus, there exists a decomposition $w = xyz$ satisfying the three conditions of the pumping lemma.

Since the second condition of the pumping lemma tells us that $|xy| \leq p$, it must be the case that, in any decomposition, we have $xy = \mathtt{a}^k$ for some $k \leq p$. Consequently, we have $y = \mathtt{a}^\ell$ for some $1 \leq \ell \leq k$.

If we pump y one or more times, then we will always end up with a word that contains more as than bs, and this word will always belong to the language $L_{\mathrm{a}>\mathrm{b}}$.

However, if we pump y down to obtain the word $xy^0z = xz$, then our word will be of the form $\mathtt{a}^{p+1-\ell}\mathtt{b}^p$. Since $\ell \geq 1$, our resultant word has at most as many as as bs, and so it no longer belongs to the language $L_{\mathrm{a}>\mathrm{b}}$. This violates the third condition of the pumping lemma. As a consequence, the language cannot be regular.

1.5.2. The Myhill–Nerode Theorem ◇

Following our discussion of the pumping lemma for regular languages, I will reveal that the pumping lemma doesn't actually work for all languages: there exist nonregular languages that satisfy the lemma's conditions. (See, e.g., Ehrenfeucht, Parikh, and Rozenberg [1981] and Johnsonbaugh and Miller [1990].) This is because the pumping lemma merely gives a necessary, but not sufficient, condition for regularity. To rectify this, I will talk about the Myhill–Nerode theorem, which gives both necessary and sufficient conditions for regularity.

Summary. Now that we've established that there exist both regular languages and nonregular languages, we can draw a diagram to represent the theory world as we know it currently. At this point, we're only familiar with two language classes: the class of regular languages and the class of finite languages, which is a subclass of the regular languages that we mentioned very briefly. We also only know about one machine model: finite automata.

> *Remark.* Finite languages are recognized by a special kind of deterministic finite automaton with no cycles.

As a result, our diagram in Figure 1.12 admittedly isn't very interesting right now, but as we continue through future chapters, we will expand and add to it.

CHAPTER NOTES

Many of the references given here and in later chapters, especially those that pertain to early work in formal languages and automata theory,

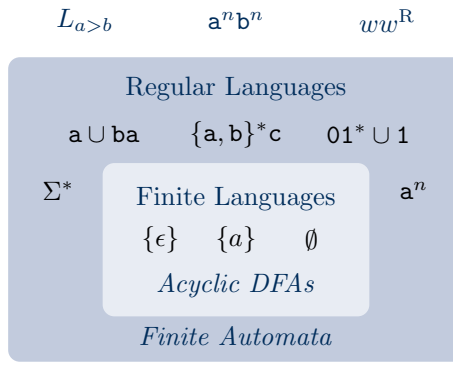


Figure 1.12. The hierarchy of language classes and models of computation, as we know it currently.

were sourced from the remarkable and comprehensive survey article written by Greibach [1981]. For readers who understand French, another in-depth survey article has been published by Perrin [1995].

The book chapter by Yu [1997] provides an excellent starting point for those wanting to learn more about the class of regular languages.

1.1. Regular expressions originated in the work of Kleene [1951, 1956], who framed the idea in terms of "regular events", which are now known as regular languages. Kleene defined three operations on events E and F: their "sum", denoted $E \vee F$; their "product", denoted EF; and the "iterate of E on F", denoted E^*F. These, in turn, became the regular operations with which we are familiar.

Brzozowski [1962] has published a detailed survey on the major results pertaining to regular expressions, many of which we discussed in this chapter.

Although we have used ϵ here to denote the empty word, some other authors use symbols like λ or Λ instead. Likewise, you may see the notation $\{\}$ being used to denote the empty language instead of $\emptyset$.

Extended regular expressions, or what we here have called regex, have also been studied in the literature under the name of "practical regular expressions"; see, for example, the work of Câmpeanu, Salomaa, and Yu [2003]. There is a vast body of material on extended regular expressions and their use in the professional world; see, for example, the book by Friedl [2006]. Cox [2007] discusses how regular expression matching algorithms are implemented and gives a comparative analysis of the performance of such implementations in various programming languages and software tools.

The word "Kleene" in "Kleene star" and "Kleene plus" is pronounced /ˈklemi/ (KLAY-nee), as that's how Stephen Kleene pronounced his last name.

> *As far as I am aware this pronunciation is incorrect in all known languages. I believe that this novel pronunciation was invented by my father.*
>
> — KEN KLEENE

1.2. The study of finite automata has its origins, of all places, in biology. McCulloch and Pitts [1943] were the first to develop a mathematical model of nervous system activity using ideas from propositional logic. Their work on "nerve nets" laid the foundation for modern research into the applications of neural networks to artificial intelligence.

It seems the first use of the phrase "theory of automata" came in a symposium talk delivered by von Neumann [1951], where he discussed the direction and future of the then-nascent field. The published version of von Neumann's talk is interesting since it is followed by a discussion between him and symposium attendees, including McCulloch—one of the authors of the aforementioned "nerve nets" paper! Von Neumann spent the final years of his life studying automata theory; for details about his contributions, see the memorial survey written by Shannon [1958].

Kleene [1951, 1956] drew a direct connection between the nerve nets of McCulloch and Pitts and the idea of a finite automaton, and established the fundamental result that a language is "representable by a nerve net" (i.e., recognized by a finite automaton) if and only if it is a regular language.

Huffman [1954a,b] formalized the notions of states and transition tables (which he called "flow tables") in the context of analyzing electronic circuits; this work originally appeared in his 1953 doctoral thesis. You may know Huffman better from his work in coding theory and his discovery of *Huffman coding*, which he made as a graduate student.

Building upon Huffman's work, both Mealy [1955] and Moore [1956] investigated further properties of electronic circuits and abstracted the study from circuits to "representations of circuit requirements", otherwise known as flow charts or finite automata.

Nondeterministic finite automata were introduced by Rabin and Scott [1959], and for this paper Rabin and Scott jointly received the Turing Award in 1976.

As we observed in comparing Figures 1.3 and 1.4 to Figures 1.5 and 1.6, respectively, recognizing the language of words whose nth-from-last symbol is 0 requires $n + 1$ states in the nondeterministic case and 2^n states in the deterministic case. Rabin and Scott [1959] first observed an exponential upper bound in the number of states between nondeterministic and deterministic finite automata recognizing the same language, but were uncertain whether this upper bound could be improved. Moore [1971] showed that the finite automata recognizing these "nth-from-last" languages meet this upper bound exactly. The question of how the size of a finite automaton is affected by converting between models or applying language operations gives rise to the study of *descriptional complexity*, also known as *state complexity*. For more details about this area of research, see the surveys by Holzer and Kutrib [2011] and by Gao, Moreira, Reis, and Yu [2016].

Nondeterministic finite automata with epsilon transitions were studied by Ott and Feinstein [1961], although in their paper they referred to such machines as "improper state diagrams". (For the curious, a "proper state diagram" to Ott and Feinstein is a deterministic finite automaton.)

McNaughton [1961] surveys many of the early major results in automata theory that we have covered here, and also discusses extensions of the model such as probabilistic automata. Readers who would like a more applied introduction to finite automata, with an eye to programming applications and connections to other areas of computer science, may be interested in the survey by Barnes [1972].

1.3. The idea of using epsilon closure to convert a nondeterministic finite automaton with epsilon transitions into one without was proposed by Ott and Feinstein [1961].

The subset construction for converting a nondeterministic finite automaton to a deterministic finite automaton is due to Rabin and Scott [1959]. Interestingly, the same fundamental idea was explored independently by Chomsky and Miller [1958], one year before the notion of a nondeterministic finite automaton was formalized!

The state elimination algorithm presented here for converting a finite automaton to a regular expression is inspired by that given by Brzozowski and McCluskey [1963].

The McNaughton–Yamada–Thompson algorithm for converting a regular expression to a finite automaton was first published by McNaughton and Yamada [1960] in a purely theoretical context and was later rediscovered by Thompson [1968] in the context of compiler theory. In their same paper, McNaughton and Yamada also gave a matrix-based

procedure to convert a finite automaton back to a regular expression, but their approach shares the same fundamental idea as the state elimination algorithm.

Kleene's theorem is so named because Kleene established an early equivalence between regular expressions and finite automata in his 1951 report. Copi, Elgot, and Wright [1958] give clearer explications and proofs of Kleene's "analysis and synthesis theorems" establishing the equivalence. Lee [1960] obtained a result matching Kleene's, but using a different abstract machine framework.

1.4. Closure of the class of regular languages under union, concatenation, Kleene star, complement, and intersection was established by Rabin and Scott [1959]. In their paper, Rabin and Scott referred to the concatenation operation as the "complex product" and the Kleene star operation, perhaps confusingly, as "closure".

1.5. The pumping lemma for regular languages was first given by Rabin and Scott [1959], and also appears in a different form in the paper of Bar-Hillel, Perles, and Shamir [1961].

Ritchie [1963] proved that the language of binary squares

$$L_{\text{sq2}} = \{(n^2)_2 \mid (n)_2 \text{ is the binary representation of } n \in \mathbb{N}\}$$

is not a regular language; as an intermediate step, he shows that the language $\{1^n 0^{n+1} 1 \mid n > 0\}$ is nonregular. Huzino and Shibata [1977] give an alternative proof of Ritchie's result.

Minsky and Papert [1966] show more generally that any language consisting of a set of binary representations of numbers is nonregular if it violates certain asymptotic density properties; specific examples of nonregular languages they give are

$$L_{k2} = \{(n^k)_2 \mid n \in \mathbb{N} \text{ and } k \geq 1\} \text{ and}$$
$$L_{\text{primes2}} = \{(p)_2 \mid p \text{ is a prime number}\}.$$

Although the pumping lemma for regular languages provides only a necessary, but not sufficient, condition for a language to be regular, Jaffe [1978] established a pumping condition for regularity that is both necessary and sufficient.

CHAPTER TWO

CONTEXT-FREE LANGUAGES

Recall that, in our discussion on regular languages, we introduced the notion of a regular expression. This expression essentially performed a kind of pattern matching to accept text in a certain form and reject all other text not of that form.

We can take this idea of matching patterns in text and modify it to work not just for individual words, but for the structure and composition of the entire text. This ability comes in the form of *grammars*, which provide us with a set of *rules* that we can follow to produce words that belong to a certain language. If a grammar produces all and only those words belonging to a certain language, then we say the grammar *generates* that language.

The idea of creating grammars for languages is nothing new; linguists have been using grammars to study natural languages for centuries, dating all the way back to the work of the ancient grammarian Pāṇini [c. 500 bce], who created an early grammar for Sanskrit. Mathematicians developed *rewriting rules* in the early 1900s to transform strings of symbols, and with the mid-century advent of computer science, grammars began to be applied to formal languages and programming languages.

If you look at the specification manual for any programming language, you will likely find tucked away somewhere in the documentation a grammar for that language. This grammar, which could number into the tens of pages, describes precisely what the structure of a program written in that language should look like. In fact, this grammar is exactly what the compiler relies on to check for syntax errors in your program!

As an example, let's consider the excerpt depicted in Figure 2.1, taken from the grammar found in the *Java Language Specification* [Gosling, Joy, Steele, and Bracha, 2005, chapter 18]. This part of the grammar checks code blocks such as if-else statements, for and while loops, and so on. Every *italicized* word corresponds to another rule in the grammar, while `monospaced` words are language keywords. For instance, the `if` rule checks that every if-else block in a program follows the syntax that the compiler expects: it begins with the keyword `if` together with a *parenthesized expression*, followed by a *statement*, and ending with an optional `else` block.

Statement:
Block
`assert` *Expression [* `:` *Expression]* `;`
`if` *ParExpression Statement [*`else` *Statement]*
`for` `(` *ForControl* `)` *Statement*
`while` *ParExpression Statement*
`do` *Statement* `while` *ParExpression* `;`
`try` *Block (Catches | [Catches]* `finally` *Block)*
`switch` *ParExpression* { *SwitchBlockStatementGroups* }
`synchronized` *ParExpression Block*
`return` *[Expression]* `;`
`throw` *Expression* `;`
`break` *[Identifier]*
`continue` *[Identifier]*
`;`
StatementExpression `;`
Identifier `:` *Statement*

Figure 2.1. An excerpt from the grammar in the *Java Language Specification*.

2.1. CONTEXT-FREE GRAMMARS

The Java grammar is an example of a *context-free grammar*. Such a grammar consists of a set of rules that we can use, in this instance, to generate valid programs in Java. These rules take on a very general form: observe, for example, that we can replace a *Statement* by a *Block*, or by the line `return` *[Expression]* `;`, or by a number of other combinations of keywords and rules, all of which are specified by the lines of the grammar following the *Statement* label.

Before we look at some more examples, let's formalize the notion of a context-free grammar. To construct a grammar, we need only four elements.

Definition 2.1 (Context-free grammar)

A context-free grammar is a tuple (V, Σ, R, S), where

- V is a finite set of elements called *nonterminal symbols*;
- Σ is a finite set of elements called *terminal symbols*, where $\Sigma \cap V = \emptyset$;
- R is a finite set of *rules*, where each rule consists of a nonterminal on the left-hand side and a combination of nonterminals and terminals on the right-hand side; and

- $S \in V$ is the *start nonterminal*.

In a context-free grammar, the set of nonterminal symbols V correspond to parts of a word that we have yet to "fill in" with terminal symbols from Σ. The set of rules R tell us how we can perform this "filling in". If we have a rule of the form $A \to \alpha$, then we can replace any instance of the symbol A in our word with whatever symbols make up α. The start nonterminal S is self-explanatory; it is the first thing in our word that we "fill in".

Returning to our Java grammar excerpt in Figure 2.1, we see that (for example) some of the nonterminals in the grammar include *Statement*, *Block*, *Identifier*, and *ParExpression*, while some of the terminals include `if`, `while`, `for`, and `;` (semicolon).

Importantly, we have in our definition of a context-free grammar that $\Sigma \cap V = \emptyset$; that is, the set of terminals and the set of nonterminals must be disjoint. This is to prevent the grammar from confusing terminals and nonterminals, and this is exactly why the Java language designers used uppercase letters in their nonterminals and lowercase letters in their terminals.

2.1.1. Language of a Context-Free Grammar

The sequence of rule applications we follow beginning with the start nonterminal S and ending with a completed word containing symbols from Σ is called a *derivation*. Each word of the form $(V \cup \Sigma)^*$ produced during a derivation is sometimes referred to as a *sentential form*.

For any nonterminal A and terminals u, w, and v, if we have a rule $A \to w$ in our grammar and some step of our derivation takes us from uAv to uwv, then we say that uAv *yields* uwv and we write $uAv \Rightarrow uwv$. We can represent a sequence of "yields" relations using similar notation; given words x and y, if $x = y$ or if there exists a sequence $x_1, x_2, \ldots, x_k$ where $k \geq 0$ such that

$$x \Rightarrow x_1 \Rightarrow x_2 \Rightarrow \cdots \Rightarrow x_k \Rightarrow y,$$

then we write $x \Rightarrow^* y$. This is very similar to the Kleene star notation, where the star indicates zero or more "yields" relations taking us from x to y.

With this, we can define the *language of a grammar* G over an alphabet Σ by $L(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$. In other terms, the language of a grammar contains all words that can be derived by that grammar beginning with the start nonterminal S.

Example 2.2

Consider the context-free grammar where $V = \{S, A\}$, $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, and R contains two rules:

$$S \to \mathtt{a}A\mathtt{b}$$
$$A \to \mathtt{a}A\mathtt{b} \mid \epsilon$$

Using this context-free grammar, we can generate words like

$$S \Rightarrow \mathtt{a}A\mathtt{b} \Rightarrow \mathtt{a}\,\epsilon\,\mathtt{b} = \mathtt{ab},$$
$$S \Rightarrow \mathtt{a}A\mathtt{b} \Rightarrow \mathtt{a}\,\mathtt{a}A\mathtt{b}\,\mathtt{b} \Rightarrow \mathtt{aa}\,\epsilon\,\mathtt{bb} = \mathtt{aabb}, \text{ and}$$
$$S \Rightarrow \mathtt{a}A\mathtt{b} \Rightarrow \mathtt{a}\,\mathtt{a}A\mathtt{b}\,\mathtt{b} \Rightarrow \mathtt{aa}\,\mathtt{a}A\mathtt{b}\,\mathtt{bb} \Rightarrow \mathtt{aaa}\,\epsilon\,\mathtt{bbb} = \mathtt{aaabbb},$$

and so on. For each step, the highlighted symbols indicate which symbols were added at that step. We get things started by replacing the S nonterminal by $\mathtt{a}A\mathtt{b}$, and from there we may replace the A nonterminal as many times as we like.

This context-free grammar generates all words over the alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ where the number of $\mathtt{a}$s is equal to the number of $\mathtt{b}$s, where there is at least one $\mathtt{a}$ and one $\mathtt{b}$, and where all $\mathtt{a}$s come before any $\mathtt{b}$s in the word. Thus, the language of this grammar is $L_{\mathtt{a}=\mathtt{b}} = \{\mathtt{a}^n\mathtt{b}^n \mid n \geq 1\}$.

Observe that the rule A in Example 2.2 included a vertical bar. This is simply a shorthand for writing multiple rules where each rule contains A on the left-hand side. Writing $A \to \mathtt{a}A\mathtt{b} \mid \epsilon$ is therefore equivalent to writing

$$A \to \mathtt{a}A\mathtt{b}$$
$$A \to \epsilon$$

There are very few limitations we must abide by when we write rules for a context-free grammar. All we need to ensure is that the left-hand side of each rule consists of exactly one nonterminal by itself. The right-hand side of each rule can contain any combination of terminals and nonterminals, including the empty word ϵ.

Example 2.3

Consider the context-free grammar where $V = \{S\}$, $\Sigma = \{(,)\}$, and R contains one rule:

$$S \to (S) \mid SS \mid \epsilon$$

This rule allows us to surround an occurrence of S with parentheses, to

"duplicate" an occurrence of S, or to replace some occurrence of S with ϵ, effectively removing that occurrence of S from the derivation.

Using this context-free grammar, we can generate a word like

$$\begin{aligned} S &\Rightarrow SS \\ &\Rightarrow (S)\,S \\ &\Rightarrow (\,(S)\,)S \\ &\Rightarrow ((\,\epsilon\,))S \\ &\Rightarrow (())S \\ &\Rightarrow (())\,(S) \\ &\Rightarrow (())(\,\epsilon\,) = (())(). \end{aligned}$$

Again, the highlighted symbols indicate which symbols were added at a given step.

This context-free grammar generates all words over the alphabet $\Sigma = \{(,)\}$ where each word contains *balanced parentheses*: every opening parenthesis is matched by a closing parenthesis, and each pair of parentheses is correctly nested. We can express the language of the grammar as

$$L_{()} = \{w \in \{(,)\}^* \mid \text{all prefixes of } w \text{ contain no more)s than (s, and } |w|_(= |w|_)\}.$$

Remark. Languages of balanced symbols are also known as *Dyck languages*, named for the mathematician Walther von Dyck, who studied the word problem for free groups. This problem can be thought of as a language where we must balance both parentheses `( )` and brackets `[ ]`. In the computer science context, such languages were originally called *D-events* [Schützenberger, 1962] before later being termed *Dyck sets* [Schützenberger, 1963].

Context-Free Languages. With our notion of a context-free grammar, it's easy for us to define a *context-free language*. Just like we defined a regular language to be a language represented by a regular expression, we can define a context-free language in terms of a context-free grammar.

Definition 2.4 (Context-free language)

If some language L is generated by a context-free grammar, then L is context-free.

Thus, both the language of words $\mathtt{a}^n\mathtt{b}^n$ and the language of balanced parentheses are context-free languages. As a shorthand, we denote the class of languages generated by a context-free grammar by CFG.

The class of context-free languages is remarkably less restrictive than the class of regular languages and, as we've seen, context-freeness allows us to perform certain simple actions like counting or matching symbols. Let's now consider a couple of other examples of context-free languages and their grammars.

Example 2.5

Consider the language $L = \{\mathtt{a}^{2i}\mathtt{b}^i\mathtt{c}^{j+2} \mid i, j \geq 0\}$ over the alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$. Here, we can see that the counts of as and bs are related, while the number of cs is independent of the number of as and bs.

Let's construct a context-free grammar generating words in L. Evidently, we will need two kinds of rules: one rule will generate the as and bs together, while the other rule will generate the cs. We can use the start nonterminal to apply these rules in the correct order. Our context-free grammar will therefore look like the following:

$$\begin{aligned} S &\to UV \\ U &\to \mathtt{aa}U\mathtt{b} \mid \epsilon \\ V &\to \mathtt{c}V \mid \mathtt{cc} \end{aligned}$$

Let's now take a look at each rule in turn.

- The first rule, S, ensures that we apply the U rule before the V rule. This in turn ensures that all as and bs occur before the cs in the generated word.
- The second rule, U, either recursively produces two as and one b or produces the empty word. This ensures that we maintain the correct count of $2i$ as and i bs.
- Finally, the third rule, V, either recursively produces one c or produces the symbols cc. This ensures that we have exactly $j + 2$ cs in our generated word.

Example 2.6

Recall our context-free language $L_{\mathtt{a}=\mathtt{b}} = \{\mathtt{a}^n\mathtt{b}^n \mid n \geq 1\}$. Here, let's consider a more general language:

$$L_{\text{mixeda}=\mathtt{b}} = \{w \in \{\mathtt{a}, \mathtt{b}\}^* \mid |w|_{\mathtt{a}} = |w|_{\mathtt{b}}\}.$$

Observe that the main difference with this language is that the order of as and bs no longer matters; we just need the same count of as and bs. Can we construct a context-free grammar for $L_{\text{mixeda}=\mathtt{b}}$?

Since order no longer matters, we just need our context-free grammar to generate a pair of as and bs each time we add terminal symbols. For this, we can use essentially the same rule as we used in our context-free grammar for $L_{\mathtt{a}=\mathtt{b}}$: $S \to \mathtt{a}S\mathtt{b} \mid \mathtt{b}S\mathtt{a}$. We also need a rule that allows us to mix the order of as and bs; for instance, to place two as or two bs next to each other, or to generate words with matching first and last symbols (like abba). For this, we can use a rule similar to one we included in our context-free grammar for $L_{()}$: $S \to SS$.

Thus, our context-free grammar will look like the following:

$$S \to SS \mid \mathtt{a}S\mathtt{b} \mid \mathtt{b}S\mathtt{a} \mid \epsilon$$

As an aside, one very common and popular question asks why these grammars and languages are given the name "context-free". To understand where this name comes from, we must take a closer look at the form of any rule in a context-free grammar. Definition 2.1 states that each rule in a context-free grammar consists of "a nonterminal on the left-hand side and a combination of nonterminals and terminals on the right-hand side", and if we were to represent this using symbols, we would get rules that are of the form

$$A \to \alpha,$$

where $A \in V$ and $\alpha \in (V \cup \Sigma)^*$. Now, if during some derivation we wish to replace an occurrence of the nonterminal A with whatever symbols are in α, we can just substitute A for the symbols in α directly. In other words, the symbols surrounding A (also known as the *context*) don't have any effect on the substitution, and so the process of replacing A with α is *free of context*! By contrast, if we had rules of the form

$$\beta A \gamma \to \beta\alpha\gamma,$$

where $A \in V$, $\alpha \in (V \cup \Sigma)^+$, and $\beta, \gamma \in (V \cup \Sigma)^*$, then we could replace A with α only when A appears within the context $\beta \,\square\, \gamma$. This gives rise to the notions of *context-sensitive grammars* and *context-sensitive languages*,

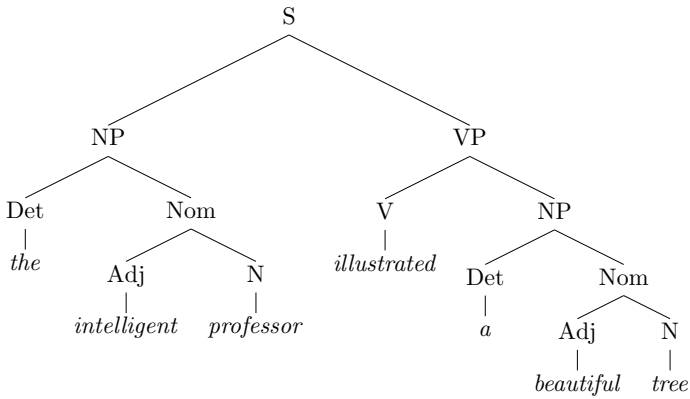


Figure 2.2. A parse tree for an English sentence.

which are interesting in their own right but omitted from our discussion here.

2.1.2. Ambiguity

If we are given a derivation of a word for some context-free grammar, we need not always represent it in a linear fashion like we did in previous examples. We could alternatively represent it as a tree structure, where the root of the tree corresponds to the start nonterminal S and each branch of the tree adds a new nonterminal or terminal symbol. We refer to such trees as *parse trees*.

Parse trees are very familiar to linguists: the idea is used all the time to break down sentences or phrases into their constituent components, like nouns, verbs, and so on. In doing so, linguists are able to study the structures of sentences in different languages. For example, consider the parse tree for an English sentence depicted in Figure 2.2. In this tree, the sentence (S) is broken down into a noun phrase (NP) and a verb phrase (VP); the noun phrase is broken down further into a determiner (Det) and a nominal (Nom); and so on. There are all kinds of rules specifying exactly how we can break down English sentences in this way.

In every parse tree, the root of the tree is the start nonterminal S, the leaves of the tree contain terminal symbols from Σ (or ϵ), and all other vertices of the tree contain nonterminal symbols from V. If a parse tree contains an internal (non-leaf) vertex A, and all the children of the vertex A are labelled $a_1, a_2, \ldots, a_n$, then the underlying grammar's rule set must contain a rule of the form $A \to a_1 a_2 \ldots a_n$.

For most grammars we deal with, there exists exactly one way to generate any given word in the language of the grammar, and thus exactly one parse

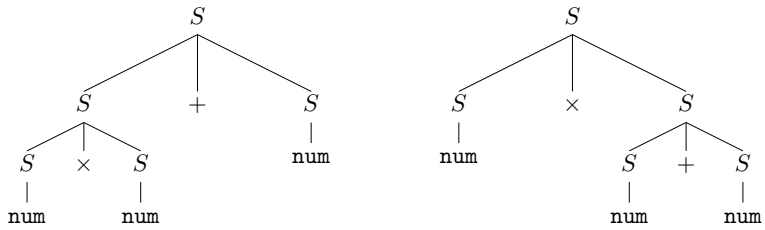
tree for each word. However, this is not always the case. There are some grammars that allow us to generate the same word in more than one way.

Perhaps one of the most well-known examples where this is the case—namely, from viral posts online that ask you to simplify $8 \div 2(2+2)$ or something similar—is the grammar generating the language of arithmetic expressions. If you recall grade school mathematics, you'll remember that there is an order of operations that specify the order in which we should apply arithmetic operations in a given expression. We first evaluate expressions in parentheses, then exponents, then multiplications and divisions, and finally additions and subtractions.

Let's consider a simplified set of operations, where we only use parentheses, addition, and multiplication. The grammar generating the language of arithmetic expressions using these three operators together with the standard set of numbers is as follows:

$$\begin{aligned} S &\to (S) \\ S &\to S + S \\ S &\to S \times S \\ S &\to \texttt{num} \end{aligned} \tag{2.1}$$

If we consider the expression $\texttt{num} \times \texttt{num} + \texttt{num}$, we discover that there exists more than one way to generate this expression, depending on whether we apply the $+$ rule or the $\times$ rule first. This is evidenced by the fact that there exist two parse trees for the same expression:



We don't need to do anything tricky in order to obtain these different parse trees. In fact, both parse trees can be obtained simply by applying rules to each nonterminal from left to right; that is, at some level of the parse tree where there exists two nonterminals, we apply a rule to the first (left) nonterminal before the second (right) nonterminal. This process is known as a *leftmost derivation*.

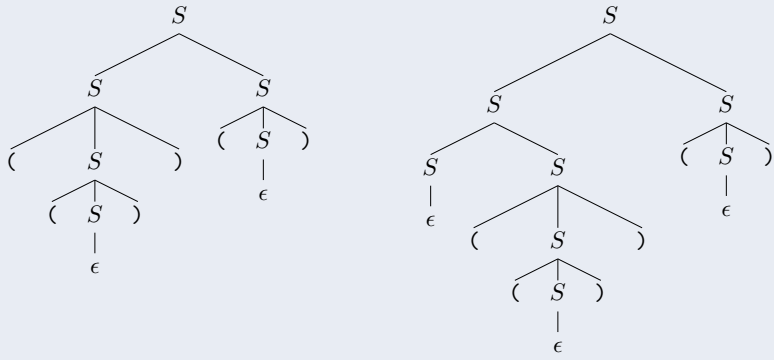
If there exists some word in the language of a grammar for which there is more than one leftmost derivation of that word, then we say that the word is derived *ambiguously*. Likewise, the grammar producing that word is itself ambiguous.

Definition 2.7 (Ambiguous context-free grammar)

A context-free grammar G is ambiguous if there exists some word $w \in L(G)$ that can be derived ambiguously.

Example 2.8

The grammar from Example 2.3 generating our language of words with balanced parentheses is ambiguous. Consider again the word `(())()`. There exist two different parse trees corresponding to leftmost derivations of this word:



Reducing and Removing Ambiguity. In certain cases, if we have an ambiguous context-free grammar, then we can create a context-free grammar for the same language that has reduced, or even no, ambiguity.

As an example, recall our three-operation arithmetic grammar:

$$\begin{aligned} S &\to \texttt{(}S\texttt{)} \\ S &\to S + S \\ S &\to S \times S \\ S &\to \texttt{num} \end{aligned} \tag{2.1}$$

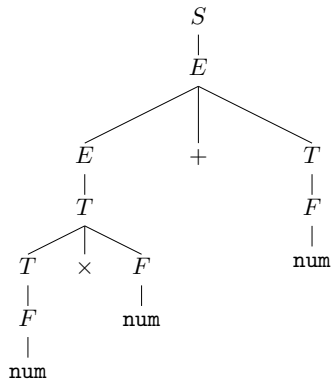
Nothing in this grammar forces us to use one rule before another, so we end up being able to derive the same word via different sequences of rule applications. However, we can construct an unambiguous grammar simply by adding a little more structure to our rules—specifically, by adding a few

more nonterminals:

$$\begin{aligned} S &\to E \\ E &\to E + T \mid T \\ T &\to T \times F \mid F \\ F &\to \texttt{(}E\texttt{)} \mid \texttt{num} \end{aligned} \tag{2.2}$$

Now, each nonterminal plays a particular role. The nonterminal S, as usual, serves as our starting point and produces an expression, E. Each expression consists of subexpressions E or additive terms T. Likewise, each term consists of subterms T or multiplicative factors F. Finally, each factor can be either a `num` or a parenthesized subexpression, starting the whole process over again.

Our revised grammar now allows us to draw one unambiguous parse tree for the expression $\texttt{num} \times \texttt{num} + \texttt{num}$:



We won't verify here that this grammar is in fact equivalent to our original one, though we can intuit that they both generate the same language. Suffice it to say that, with this revised grammar, we're able to guarantee that the addition rule is always applied before the multiplication rule.

> *Remark.* The reason why we didn't verify that our two context-free grammars are equivalent is because the problem of determining the equivalence of context-free grammars is impossible for a computer to solve in general. We will discuss this in greater depth in Sections 5.4 and 5.5.

Inherent Ambiguity. Unfortunately, there is no general procedure or algorithm for removing ambiguity from a context-free grammar; indeed, it isn't even possible to remove ambiguity in some cases. Some context-free languages are *inherently ambiguous*, meaning that any grammar generating the language will have some unavoidable ambiguous component to it.

Example 2.9

Let $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, and consider the language

$$L_{\text{twoequal}} = \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k \mid i, j, k \geq 0 \text{ and } i = j \text{ or } j = k\}.$$

This language contains all words that have either the same number of as and bs or the same number of bs and cs.

We can generate this language using the following grammar:

$$\begin{aligned} S &\to S_1 \mid S_2 \\ S_1 &\to S_1\mathtt{c} \mid A \\ A &\to \mathtt{a}A\mathtt{b} \mid \epsilon \\ S_2 &\to \mathtt{a}S_2 \mid B \\ B &\to \mathtt{b}B\mathtt{c} \mid \epsilon \end{aligned}$$

The rules S_1 and A generate words of the form $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^m$ and the rules S_2 and B generate words of the form $\mathtt{a}^m\mathtt{b}^n\mathtt{c}^n$, each where $m, n \geq 0$.

Now, consider words of the form $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$, where $n \geq 0$. All words of this form belong to the language L_{twoequal}, but each such word has two distinct derivations in this grammar: it can be generated either by the rules S_1 and A, or by the rules S_2 and B.

While the formal proof that this language is inherently ambiguous is quite long, we can intuitively see that (for instance) any grammar generating this language must have rules similar to S_1 and A to produce balanced pairs of as and bs followed by some number of cs. We can make a similar argument for the rules S_2 and B. Thus, any grammar for this language will include some degree of ambiguity.

2.1.3. Normal Forms

Up to now, we've imposed no restrictions on the form of each rule in our context-free grammars. As long as each rule of our context-free grammar looked like $A \to \alpha$, where A is a nonterminal symbol and α is a combination of terminal and nonterminal symbols, we were happy.

However, computers (and, by extension, the people who program computers) like having structure. For instance, a compiler for a programming language usually incorporates a context-free grammar into its workflow at some point during the compilation of a program, and having a structured grammar makes the compiler's job both easier and faster.

Therefore, at times, we might like to transform a context-free grammar into a more-structured *normal form*; that is, to modify the grammar in such a way that each rule takes a canonical form. There are a number of normal

forms to choose from, and each one comes with its own benefits.

2.1.3.1. Chomsky Normal Form. The *Chomsky normal form*, as the name suggests, was first studied by the linguist Noam Chomsky [1959a] as he attempted to develop a model for natural language using grammars. A grammar in Chomsky normal form is one where each rule either has two nonterminal symbols or one terminal symbol on the right-hand side.

Definition 2.10 (Chomsky normal form)

A context-free grammar is in Chomsky normal form if every rule in the grammar is of one of the two following forms:

1. $A \to BC$ for $A, B, C \in V$ with $B, C \neq S$; or
2. $A \to a$ for $A \in V$ and $a \in \Sigma$.

Additionally, we may allow the rule $S \to \epsilon$.

The main benefit of converting a grammar into Chomsky normal form comes in how we can represent and store derivations of words in memory. Since each rule derives either two nonterminal symbols or one terminal symbol, every parse tree will have a branching factor of either 2 or 1. This fact allows us to use efficient data structures for representing binary trees in memory, as well as to apply efficient algorithms to process parse trees and derivations. Moreover, the number of steps in a derivation using a grammar in Chomsky normal form is easy to bound: if the grammar generates a word w, then the derivation of w will contain $|w| - 1$ applications of a rule of the first form and $|w|$ applications of a rule of the second form.

Example 2.11

Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, and consider the following two grammars. Each grammar generates words consisting of one `b` surrounded on either side by zero or more `a`s. The grammar on the left is not in Chomsky normal form. The grammar on the right is in Chomsky normal form, and it is equivalent to the grammar on the left.

$$\begin{aligned} S &\to A\mathtt{b}A \\ A &\to A\mathtt{a} \mid \epsilon \end{aligned} \qquad\qquad \begin{aligned} S_0 &\to TA \mid BA \mid AB \mid \mathtt{b} \\ T &\to AB \\ A &\to AC \mid \mathtt{a} \\ B &\to \mathtt{b} \\ C &\to \mathtt{a} \end{aligned}$$

Every context-free grammar can be converted into a context-free grammar in Chomsky normal form, and the conversion process consists of five steps:

1. **START**: Replace the start nonterminal.

   Add a new start nonterminal S_0 together with a new rule $S_0 \to S$, where S is the start nonterminal of the original grammar.

   This ensures that the new start nonterminal S_0 will not occur on the right-hand side of any rule.

2. **TERM**: Remove nonsolitary terminals from the right-hand side of all rules.

   For each rule of the form $A \to \alpha_1 \dots a \dots \alpha_n$, where $A \in V$, $\alpha_1, \dots, \alpha_n \in V \cup \Sigma$ and $a \in \Sigma$, add a new rule $T_a \to a$ and replace the existing rule with one of the form $A \to B_1 \dots T_a \dots B_n$. If multiple terminals appear on the right-hand side of the rule, replace all terminals simultaneously.

   This ensures that the right-hand sides of all rules consist either of a single terminal or some number of nonterminals.

3. **BIN**: Split up groups of three or more nonterminals on the right-hand side of all rules.

   For each rule of the form $A \to B_1 B_2 \dots B_n$, where $A, B_1, \dots, B_n \in V$ and $n \geq 3$, replace the existing rule with the set of rules

   $$\begin{aligned} A &\to B_1 A_1, \\ A_1 &\to B_2 A_2, \\ &\vdots \\ A_{n-2} &\to B_{n-1} B_n. \end{aligned}$$

   This ensures that every parse tree produced by the resultant grammar will have a bounded branching factor.

4. **DEL**: Remove epsilon rules.

   Remove all rules of the form $A \to \epsilon$, where $A \neq S_0$. For all rules of the form $A \to BC$ where $A, B, C \in V$ and either B or C is *nullable* (i.e., where there exists a rule $B \to \epsilon$ or $C \to \epsilon$), replace the existing rule with one where the nullable nonterminal is removed.

5. **UNIT**: Remove unit rules.

   For each pair of rules of the form $A \to B$ and $B \to C$, where $A, B \in V$ and $C \in V^+$, replace the existing rule $A \to B$ with one of the form $A \to C$, unless this replacement produces a unit rule that has previously been removed.

The process of converting a context-free grammar to Chomsky normal form can be lengthy and tedious, so the job is often automated by a subroutine within a compiler or a similar program. However, we're here to learn, and learning is best done by doing. Thus, let's work through an example of this conversion process step by step.

Example 2.12

Consider the following grammar not in Chomsky normal form:

$$\begin{aligned} S &\to ASB \\ A &\to \mathtt{a}AS \mid \mathtt{a} \mid \epsilon \\ B &\to S\mathtt{b}S \mid A \mid \mathtt{bb} \end{aligned}$$

We will convert this grammar to an equivalent grammar in Chomsky normal form.

1. **START**: Replace the start nonterminal.

   We begin by adding a new start nonterminal and the rule $S_0 \to S$, which gives us the following:

$$\begin{aligned} S_0 &\to S \\ S &\to ASB \\ A &\to \mathtt{a}AS \mid \mathtt{a} \mid \epsilon \\ B &\to S\mathtt{b}S \mid A \mid \mathtt{bb} \end{aligned}$$

2. **TERM**: Remove nonsolitary terminals from the right-hand side of all rules.

   We have three rules to handle here: $A \to \mathtt{a}AS$, $B \to S\mathtt{b}S$, and $B \to \mathtt{bb}$. Adding the new rules $T_{\mathtt{a}} \to \mathtt{a}$ and $T_{\mathtt{b}} \to \mathtt{b}$ and making

the appropriate substitutions gives us the following:

$$
\begin{aligned}
S_0 &\to S \\
S &\to ASB \\
A &\to T_{\mathtt{a}}AS \mid \mathtt{a} \mid \epsilon \\
B &\to ST_{\mathtt{b}}S \mid A \mid T_{\mathtt{b}}T_{\mathtt{b}} \\
T_{\mathtt{a}} &\to \mathtt{a} \\
T_{\mathtt{b}} &\to \mathtt{b}
\end{aligned}
$$

3. **BIN**: Split up groups of three or more nonterminals on the right-hand side of all rules.

   Again, we have three rules to split up: $S \to ASB$, $A \to T_{\mathtt{a}}AS$, and $B \to ST_{\mathtt{b}}S$. Splitting these rules gives us the following:

$$
\begin{aligned}
S_0 &\to S \\
S &\to AS_1 \qquad & S_1 &\to SB \\
A &\to T_{\mathtt{a}}A_1 \mid \mathtt{a} \mid \epsilon \qquad & A_1 &\to AS \\
B &\to SB_1 \mid A \mid T_{\mathtt{b}}T_{\mathtt{b}} \qquad & B_1 &\to T_{\mathtt{b}}S \\
T_{\mathtt{a}} &\to \mathtt{a} \\
T_{\mathtt{b}} &\to \mathtt{b}
\end{aligned}
$$

4. **DEL**: Remove epsilon rules.

   This grammar has one obvious epsilon rule, which is $A \to \epsilon$. However, we must also modify the rules that contain the nullable nonterminal A: these rules are $S \to AS_1$, $A_1 \to AS$, and $B \to A$. For each of these rules, we add a new rule that is of the same form, but with the nonterminal A removed.

$$
\begin{aligned}
S_0 &\to S \\
S &\to AS_1 \mid S_1 \qquad & S_1 &\to SB \\
A &\to T_{\mathtt{a}}A_1 \mid \mathtt{a} \mid \not{\epsilon} \qquad & A_1 &\to AS \mid S \\
B &\to SB_1 \mid A \mid \epsilon \mid T_{\mathtt{b}}T_{\mathtt{b}} \qquad & B_1 &\to T_{\mathtt{b}}S \\
T_{\mathtt{a}} &\to \mathtt{a} \\
T_{\mathtt{b}} &\to \mathtt{b}
\end{aligned}
$$

Observe that removing the nullable nonterminal A from the rule $B \to A$ produced another epsilon rule, $B \to \epsilon$, so we must remove that rule as well. This also means that B is a nullable nonterminal, and so we must modify rules containing B: the only rule affected here is $S_1 \to SB$. For this rule, we again add a new rule that is of the same form, but with the nonterminal B removed.

$$
\begin{aligned}
&S_0 \to S \\
&S \to AS_1 \mid S_1 && S_1 \to SB \mid S \\
&A \to T_{\mathtt{a}}A_1 \mid \mathtt{a} && A_1 \to AS \mid S \\
&B \to SB_1 \mid A \mid \not{\epsilon} \mid T_{\mathtt{b}}T_{\mathtt{b}} && B_1 \to T_{\mathtt{b}}S \\
&T_{\mathtt{a}} \to \mathtt{a} \\
&T_{\mathtt{b}} \to \mathtt{b}
\end{aligned}
$$

5. **UNIT**: Remove unit rules.

   Lastly, we handle all of the unit rules in this grammar. We'll begin by removing the unit rule $S_0 \to S$ to obtain the following:

$$
\begin{aligned}
&S_0 \to AS_1 \mid S_1 \\
&S \to AS_1 \mid S_1 && S_1 \to SB \mid S \\
&A \to T_{\mathtt{a}}A_1 \mid \mathtt{a} && A_1 \to AS \mid S \\
&B \to SB_1 \mid A \mid T_{\mathtt{b}}T_{\mathtt{b}} && B_1 \to T_{\mathtt{b}}S \\
&T_{\mathtt{a}} \to \mathtt{a} \\
&T_{\mathtt{b}} \to \mathtt{b}
\end{aligned}
$$

   In removing this rule, we added a new unit rule $S_0 \to S_1$, so let's take care of that rule next. Note that straightforwardly performing the substitution on the right-hand side would again produce the unit rule $S_0 \to S$ that we already removed, so we omit that rule and obtain the following:

$$
\begin{aligned}
&S_0 \to AS_1 \mid SB \\
&S \to AS_1 \mid S_1 && S_1 \to SB \mid S \\
&A \to T_{\mathtt{a}}A_1 \mid \mathtt{a} && A_1 \to AS \mid S \\
&B \to SB_1 \mid A \mid T_{\mathtt{b}}T_{\mathtt{b}} && B_1 \to T_{\mathtt{b}}S \\
&T_{\mathtt{a}} \to \mathtt{a} \\
&T_{\mathtt{b}} \to \mathtt{b}
\end{aligned}
$$

We now remove the unit rule $S \to S_1$. Performing the substitution on the right-hand side would produce the (useless) unit rule $S \to S$, so we omit that rule and obtain the following:

$$
\begin{aligned}
S_0 &\to AS_1 \mid SB \\
S &\to AS_1 \mid SB \qquad & S_1 &\to SB \mid S \\
A &\to T_{\mathtt{a}}A_1 \mid \mathtt{a} & A_1 &\to AS \mid S \\
B &\to SB_1 \mid A \mid T_{\mathtt{b}}T_{\mathtt{b}} & B_1 &\to T_{\mathtt{b}}S \\
T_{\mathtt{a}} &\to \mathtt{a} \\
T_{\mathtt{b}} &\to \mathtt{b}
\end{aligned}
$$

We can now remove the unit rules $S_1 \to S$ and $A_1 \to S$, which both have the nonterminal S on the right-hand side. This produces the following:

$$
\begin{aligned}
S_0 &\to AS_1 \mid SB \\
S &\to AS_1 \mid SB \qquad & S_1 &\to SB \mid AS_1 \\
A &\to T_{\mathtt{a}}A_1 \mid \mathtt{a} & A_1 &\to AS \mid AS_1 \mid SB \\
B &\to SB_1 \mid A \mid T_{\mathtt{b}}T_{\mathtt{b}} & B_1 &\to T_{\mathtt{b}}S \\
T_{\mathtt{a}} &\to \mathtt{a} \\
T_{\mathtt{b}} &\to \mathtt{b}
\end{aligned}
$$

Lastly, we remove the unit rule $B \to A$, which gives us our Chomsky normal form grammar:

$$
\begin{aligned}
S_0 &\to AS_1 \mid SB \\
S &\to AS_1 \mid SB \qquad & S_1 &\to SB \mid AS_1 \\
A &\to T_{\mathtt{a}}A_1 \mid \mathtt{a} & A_1 &\to AS \mid AS_1 \mid SB \\
B &\to SB_1 \mid T_{\mathtt{a}}A_1 \mid \mathtt{a} \mid T_{\mathtt{b}}T_{\mathtt{b}} & B_1 &\to T_{\mathtt{b}}S \\
T_{\mathtt{a}} &\to \mathtt{a} \\
T_{\mathtt{b}} &\to \mathtt{b}
\end{aligned}
$$

2.1.3.2. Greibach Normal Form ⋄.

Together with the section on Chomsky normal form, I would also like to write a discussion of Greibach normal form [1965], its benefits, and how we can convert a context-free grammar into GNF.

2.2. PUSHDOWN AUTOMATA

When we first introduced finite automata as a computational model for regular languages, we emphasized the facts that finite automata have no method of storage and no ability to return to a previously read symbol. Naturally, these restrictions limited the kinds of languages the model is able to recognize, and we showed that such restrictions resulted in the model recognizing exactly the class of regular languages.

At the end of the previous lecture, we saw that there exist languages that are not regular, and therefore are not recognized by finite automata. We know now that the next "step" of our language hierarchy is the class of context-free languages. Thus, a new question arises: what kind of computational model is capable of recognizing context-free languages?

Since every context-free language is generated by a context-free grammar, and since we know that context-free grammars must "remember" which nonterminal and terminal symbols are being manipulated over the course of a derivation, any model of computation recognizing context-free languages must include a form of memory. What is the best form of memory to use in this situation? If we view a derivation as a parse tree, then the derivation progresses as we go deeper into the parse tree, and we can easily model the depth of a derivation using *stack* memory.

As a brief review, a stack is a data structure with two operations that manipulate data: *push* and *pop*. Pushing a symbol to a stack adds it to the top of the stack, above all other symbols already in the stack. Conversely, popping a symbol from a stack removes it from the top of the stack, leaving all other symbols untouched. (See Figure 2.3 for an example of pushing and popping.) As a result, a stack provides last-in-first-out, or LIFO, storage—by comparison, a data structure like a queue provides first-in-first-out, or FIFO, storage. We can view the symbol at the top of the stack at any time during a computation, but we cannot view any other symbols in the stack unless we pop the symbol currently at the top of the stack.

Since we're dealing with an abstract model of computation and not a real-world computer, we can make the assumption that our stack size is *unbounded*; that is, we can push as many symbols to the stack as we want without worrying about running out of space.

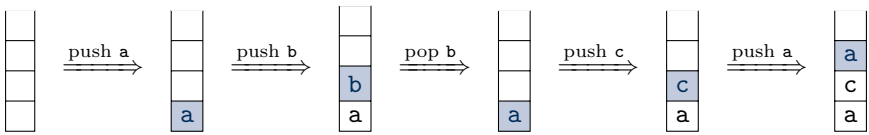


Figure 2.3. Pushing symbols to and popping symbols from a stack. At each step, the currently visible symbol at the top of the stack is highlighted.

Now that we have our form of storage established, we can define our model of computation. At its core, this model is a finite automaton with a stack added to it. Since the automaton is now able to push symbols to a stack, we give it an appropriate name: a *pushdown automaton*.

> *Remark.* The name "pushdown automaton" doesn't specifically come from its ability to push symbols, but rather from an older term for a stack: a *pushdown store*.

In addition to reading a symbol of its input word on a transition, a pushdown automaton can read from and write to the stack on the same transition. In order to perform this mixture of input and stack actions, we specify two alphabets for a pushdown automaton: the *input alphabet*, which contains symbols used in the input word, and the *stack alphabet*, which contains symbols the pushdown automaton can use in its stack. This allows us to combine actions on the input word and actions on the stack in a single transition, without risking confusion over the meaning of any particular alphabet symbol. The transitions of a pushdown automaton may additionally use ϵ in place of either the input word action (i.e., when we don't read a symbol of the input word) or the stack action (i.e., when we don't push to/pop from the stack). Just like we denoted a finite automaton's alphabet by Σ, we will use Σ to denote a pushdown automaton's input alphabet. Likewise, we will use Γ to denote the stack alphabet.

In order for our model of computation to use two alphabets at once, we must modify its transition function accordingly. Recall that a finite automaton (with epsilon transitions) transitions on a pair (q, a), where $q \in Q$ and $a \in \Sigma \cup \{\epsilon\}$. By comparison, a pushdown automaton transitions on a tuple (q, a, b), where $q \in Q$, $a \in \Sigma \cup \{\epsilon\}$, and $b \in \Gamma \cup \{\epsilon\}$. Thus, a pushdown automaton uses both the current symbol of its input word (or ϵ) as well as the top symbol of its stack (or ϵ) to determine the state to which it will transition. After transitioning, the pushdown automaton will be in a possibly different state, and it will have a possibly different symbol at the top of its stack.

Lastly, a pushdown automaton has no inherent mechanism for detecting whether its stack is empty. To make our lives easier when it comes to keeping track of the stack contents, we can incorporate a special "bottom of stack" symbol $\perp$ into the transitions of a pushdown automaton in such a way that $\perp$ is both the first symbol pushed to the stack and the last symbol popped from the stack.

> *Remark.* We don't require this special symbol, since pushdown automata can accept either by being in a final state or by having an empty stack after reaching the end of its input word. As it turns out, these two methods of acceptance are equivalent, and our approach effectively combines the two.

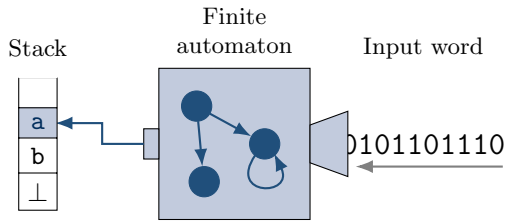


Figure 2.4. An illustration of a pushdown automaton.

Having established all of the technical details, we can now formulate the definition of a pushdown automaton. Since a pushdown automaton is essentially just a finite automaton with a stack, we can start by copying the text from Definition 1.14 and adding to it the necessary components for handling the stack: the stack alphabet, a mechanism for popping symbols (if necessary) from the stack, and a mechanism for pushing symbols (if necessary) to the stack.

Definition 2.13 (Pushdown automaton)

A pushdown automaton is a tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where

- Q is a finite set of *states*;
- Σ is the *input alphabet*;
- Γ is the *stack alphabet*;
- $\delta \colon Q \times (\Sigma \cup \{\epsilon\}) \times (\Gamma \cup \{\epsilon\}) \to \mathcal{P}(Q \times (\Gamma \cup \{\epsilon\}))$ is the *transition function*;
- $q_0 \in Q$ is the *initial* or *start state*; and
- $F \subseteq Q$ is the set of *final* or *accepting states*.

Figure 2.4 illustrates how we might visualize a pushdown automaton. Observe that our definition doesn't mention the "bottom of stack" symbol $\perp$, since it isn't strictly necessary. However, since we will use it to aid in our understanding, we will assume that $\perp \in \Gamma$ and that $\perp$ is not used in any transition of δ except for those exiting q_0 and those entering any final state.

> *Remark.* Here is another moment for grammatical pedantry. Just like the distinction between the singular "finite automat*on*" and the plural "finite automat*a*", it is never correct to write something like "*a* pushdown automat*a*" in reference to a single instance of such a model.

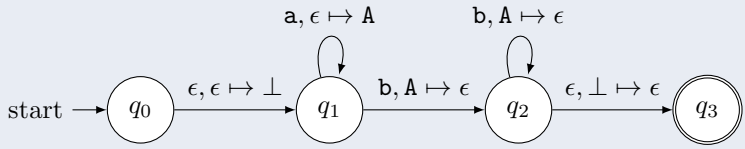
Example 2.14

Consider a pushdown automaton $\mathcal{M}$ where $Q = \{q_0, q_1, q_2, q_3\}$, $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, $\Gamma = \{\bot, \mathtt{A}\}$, $q_0 = q_0$, $F = \{q_3\}$, and δ is defined as follows:

Σ:		$\mathtt{a}$			$\mathtt{b}$			ϵ	
Γ:	$\bot$	$\mathtt{A}$	ϵ	$\bot$	$\mathtt{A}$	ϵ	$\bot$	$\mathtt{A}$	ϵ
q_0	—	—	—	—	—	—	—	—	$\{(q_1, \bot)\}$
q_1	—	—	$\{(q_1, \mathtt{A})\}$	—	$\{(q_2, \epsilon)\}$	—	—	—	—
q_2	—	—	—	—	$\{(q_2, \epsilon)\}$	—	$\{(q_3, \epsilon)\}$	—	—
q_3	—	—	—	—	—	—	—	—	—

In the transition function table, the top row indicates the input symbol being read and the second-from-top row indicates the symbol to be popped from the stack. Each entry of the table is an ordered pair where the first element is the state being transitioned to and the second element is the symbol being pushed to the stack.

The pushdown automaton $\mathcal{M}$ can be represented visually as follows:



Notice that each transition has a label of the form $a, B \mapsto C$; this means that, upon reading an input symbol a and popping a symbol B from the stack, the pushdown automaton pushes a symbol C to the stack.

Between states q_0 and q_1, the pushdown automaton pushes the symbol $\bot$ to the stack to mark the bottom. In state q_1, the pushdown automaton reads some number of $\mathtt{a}$s and pushes the same number of $\mathtt{A}$s to the stack. Between states q_1 and q_2, as well as in state q_2, the pushdown automaton reads some number of $\mathtt{b}$s and pops the same number of $\mathtt{A}$s from the stack. Finally, between states q_2 and q_3, the pushdown automaton pops $\bot$ from the stack only if there are no more input symbols to read and no more stack symbols to process.

After some observation, we can see that our pushdown automaton accepts all input words of the form $\mathtt{a}^n\mathtt{b}^n$ where $n \geq 1$.

You may have noticed in our definition that the transition function maps to the power set of state/stack symbol pairs, which makes the pushdown automaton nondeterministic. This was not done by mistake. Unlike finite automata, where the deterministic and nondeterministic models are equivalent in terms of recognition power, deterministic pushdown automata actually

recognize *fewer* languages than nondeterministic pushdown automata. In the interest of full generality, then, we take all of our pushdown automata to be nondeterministic, even if we don't need to use nondeterminism.

2.2.1. Computations and Accepting Computations

Let us now consider precisely what it means for a pushdown automaton to accept an input word. As we had with finite automata, one of the main conditions for acceptance is that there exists some sequence of states through the automaton where it begins reading its input word in an initial state and finishes reading in an accepting state. Since pushdown automata also come with a stack, though, we must account for the contents of the stack over the course of the computation. Specifically, we assume that the stack is empty at the beginning of the computation and, on each transition, the pushdown automaton can modify the top symbol of its stack appropriately.

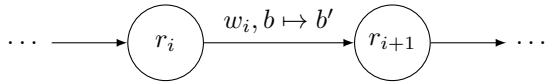
Definition 2.15 (Accepting computation of a pushdown automaton)

Let $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, F)$ be a pushdown automaton, and let $w = w_0 w_1 \ldots w_{n-1}$ be an input word of length n where $w_0, w_1, \ldots, w_{n-1} \in \Sigma$. The pushdown automaton $\mathcal{M}$ accepts the input word w if there exists a sequence of states $r_0, r_1, \ldots, r_n \in Q$ and a sequence of stack contents $s_0, s_1, \ldots, s_n \in \Gamma^*$ satisfying the following conditions:

1. $r_0 = q_0$ and $s_0 = \epsilon$;
2. $(r_{i+1}, b') \in \delta(r_i, w_i, b)$ for all $0 \leq i \leq (n-1)$, where $s_i = bt$ and $s_{i+1} = b't$ for some $b, b' \in \Gamma \cup \{\epsilon\}$ and $t \in \Gamma^*$; and
3. $r_n \in F$.

The second condition is rather notation-heavy, but the underlying idea describes exactly how a pushdown automaton transitions between states: starting in a state r_i with a symbol b at the top of the stack, the pushdown automaton reads an input symbol w_i and pops the symbol b from the stack. The transition function then sends the pushdown automaton to a state r_{i+1} and pushes the symbol b' to the stack.

Indeed, the second condition corresponds exactly to having the following transition in the pushdown automaton:

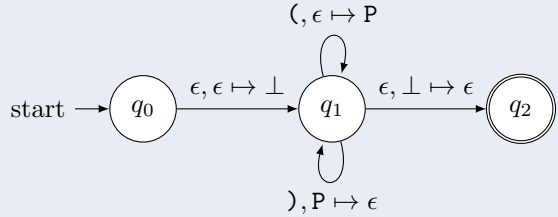


2.2.2. Language of a Pushdown Automaton

Pushdown automata recognize languages just as finite automata do, and the set of all input words accepted by a pushdown automaton is referred to as the language of that automaton. We denote the class of languages recognized by a pushdown automaton by PDA.

Example 2.16

Consider $L_{()}$, our language of balanced parentheses from earlier. Suppose $\Sigma = \{(,)\}$ and $\Gamma = \{\bot, \mathtt{P}\}$. A pushdown automaton recognizing this language is as follows:



As the transitions show, after pushing the symbol $\bot$ to the stack, the pushdown automaton reads left and right parentheses. Every time a left parenthesis `(` is read, the pushdown automaton pushes a symbol `P` to the stack. Likewise, every time a right parenthesis `)` is read, the pushdown automaton pops a symbol `P` from the stack to account for some left parenthesis being matched.

Note that, if the input word contains more right parentheses than left parentheses, then the pushdown automaton will not be able to pop a symbol `P` from the stack. Similarly, if the input word contains more left parentheses than right parentheses, then it will not be able to pop the symbol $\bot$ from the stack. In either case, it becomes stuck in state q_1 and unable to accept the input word.

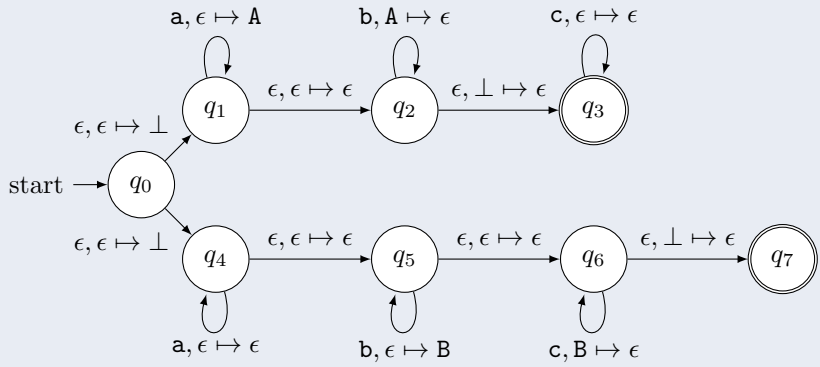
Example 2.17

Let $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, and consider the language

$$L_{\text{twoequal}} = \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k \mid i, j, k \geq 0 \text{ and } i = j \text{ or } j = k\}.$$

A pushdown automaton recognizing L_{twoequal} must have two "branches": one branch to handle the case where $i = j$, and one branch to handle the case where $j = k$. Since we don't know in advance which branch we will need to take, we can use the nondeterminism inherent in the pushdown automaton model.

A pushdown automaton recognizing this language would therefore look like the following, where the upper branch handles the case $i = j$ and the lower branch handles the case $j = k$:



2.3. EQUIVALENCE OF MODELS

You may recall from our discussion of regular languages that we proved a number of exciting results: deterministic and nondeterministic finite automata are equivalent in terms of recognition power, regardless of whether epsilon transitions are involved, and each of these models is itself equivalent in recognition power to regular expressions. These results allowed us to establish Kleene's theorem, which characterized the class of regular languages in terms of several different models of computation.

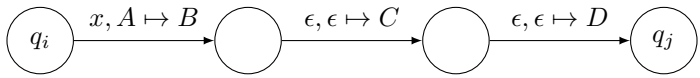
Now that we're focusing on context-free languages, and now that we have two ways of representing context-free languages—namely, context-free grammars and pushdown automata—it would be nice to establish a connection between the two representations. This brings us to yet another exciting result, which will be the focus of this section. Since the overall proof is quite lengthy, we will split the proof of the main result into two parts.

2.3.1. CFG ⇒ PDA

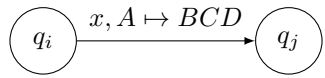
For the first half of our main result, we will show that we can convert any context-free grammar into a pushdown automaton recognizing the language generated by the grammar. Specifically, given a context-free grammar G, we will construct a pushdown automaton $\mathcal{M}$ that functions as a *top-down parser* on its input word w; that is, beginning with the start nonterminal S, $\mathcal{M}$ will repeatedly apply rules from R to check whether w can be generated via a leftmost derivation. If so, then $\mathcal{M}$ will accept w.

Remark. We could alternatively construct $\mathcal{M}$ to act as a *bottom-up parser*, where it applies rules backward starting from the input word w to see if the start nonterminal S can be reached. The outcome is the same, though, so we will not discuss this alternative construction here.

Note that, for the purposes of this proof, we will "condense" multiple transitions of our pushdown automaton into one transition; that is, if we have some sequence of transitions



then we will depict this sequence of transitions as one single transition of the form



and we replace the symbol A on the stack with the symbols BCD, in that order from bottom to top.

Lemma 2.18

Given a context-free grammar G generating a language $L(G)$, there exists a pushdown automaton $\mathcal{M}$ such that $L(\mathcal{M}) = L(G)$.

Proof. Suppose we are given a context-free grammar $G = (V, \Sigma_G, R, S)$. We construct a pushdown automaton $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, F)$ that recognizes the language generated by G in the following way:

- The set of states is $Q = \{q_S, q_R\}$. The first state, q_S, corresponds to the point during the computation at which the context-free grammar G begins to generate the word. The second state, q_R, corresponds to the remainder of the computation where G applies rules from its rule set.
- The input alphabet is $\Sigma = \Sigma_G$. If $\mathcal{M}$ accepts its input word, then the word could be generated by G, and therefore it must consist of terminal symbols.
- The stack alphabet is $\Gamma = V \cup \Sigma_G$. We will use the stack of $\mathcal{M}$ to keep track of where we are in the leftmost derivation of the word.
- The initial state is $q_0 = q_S$.
- The final state is $F = \{q_R\}$.
- The transition function δ consists of three types of transitions:

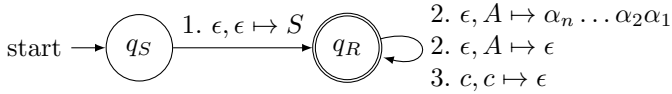
1. **Initial transition**: $\delta(q_S, \epsilon, \epsilon) = \{(q_R, S)\}$. This transition initializes the stack by pushing to it the start nonterminal S, and then moves to the state q_R for the remainder of the computation.
2. **Nonterminal transition**: $\delta(q_R, \epsilon, A) = \{(q_R, \alpha_n \dots \alpha_2 \alpha_1)\}$ for each rule of the form $A \to \alpha_1 \alpha_2 \dots \alpha_n$, where $A \in V$ and $\alpha_i \in V \cup \Sigma_G$ for all i. Transitions of this form simulate the application of a given rule by popping the left-hand side (A) from the stack and pushing the right-hand side ($\alpha_1 \alpha_2 \dots \alpha_n$) to the stack in its place in reverse order. Pushing the symbols in reverse ensures that the next symbol we need to read (α_1) is at the top of the stack.

   Note that if $n = 0$, then the transition will be of the form $\delta(q_R, \epsilon, A) = \{(q_R, \epsilon)\}$.
3. **Terminal transition**: $\delta(q_R, c, c) = \{(q_R, \epsilon)\}$ for each terminal symbol $c \in \Sigma_G$. Transitions of this form compare a terminal symbol on the stack to the current input word symbol. If the two symbols match, then the computation continues.

During the computation, after the initial transition is followed, $\mathcal{M}$ follows either nonterminal transitions or terminal transitions until its stack is empty or it runs out of input word symbols. If a nonterminal symbol A is at the top of the stack, $\mathcal{M}$ nondeterministically chooses one of the rules for A and follows the corresponding transition. If a terminal symbol c is at the top of the stack, $\mathcal{M}$ performs the comparison between input and stack symbol as described earlier.

By this construction, we can see that $\mathcal{M}$ finishes its computation with an empty stack and no input word symbols of w left to read whenever $S \Rightarrow^* w$, and so $\mathcal{M}$ accepts the input word w if w can be generated by the context-free grammar G. Therefore, $L(\mathcal{M}) = L(G)$ as desired. ■

Visually, we can think of the pushdown automaton constructed in the proof of Lemma 2.18 in the following way, where the number of each transition corresponds to its type:



Note that we don't require the symbol $\perp$ here, since we're only using the stack to keep track of where we are in the grammar's derivation.

Example 2.19

Consider the following context-free grammar G, where $V = \{S, A\}$ and $\Sigma_G = \{0, 1, \#\}$:

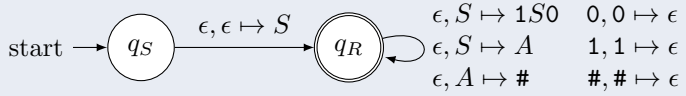
$$S \to 0S1 \mid A$$
$$A \to \#$$

This grammar generates words of the form $0^n\#1^n$, where $n \geq 0$.

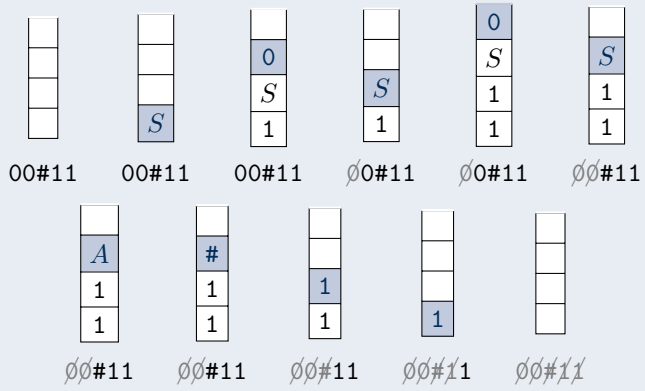
We convert the context-free grammar G to a pushdown automaton $\mathcal{M}$. Take $Q = \{q_S, q_R\}$, $\Sigma = \Sigma_G$, $\Gamma = V \cup \Sigma_G$, $q_0 = q_S$, and $F = \{q_R\}$. Finally, add the following transitions to δ:

- $\delta(q_S, \epsilon, \epsilon) = \{(q_R, S)\}$. This initial transition pushes the start nonterminal S to the stack.
- $\delta(q_R, \epsilon, S) = \{(q_R, 1S0), (q_R, A)\}$. These nonterminal transitions account for the S rules.
- $\delta(q_R, \epsilon, A) = \{(q_R, \#)\}$. This nonterminal transition accounts for the A rule.
- $\delta(q_R, 0, 0) = \{(q_R, \epsilon)\}$, $\delta(q_R, 1, 1) = \{(q_R, \epsilon)\}$, and $\delta(q_R, \#, \#) = \{(q_R, \epsilon)\}$. These terminal transitions match the terminal symbols on the stack to the input word symbols.

This pushdown automaton $\mathcal{M}$ looks like the following:



As an illustration of the computation of $\mathcal{M}$, let's look at the stack as $\mathcal{M}$ reads an example input word 00#11. We can see that G generates this word by the derivation $S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 00A11 \Rightarrow 00\#11$.

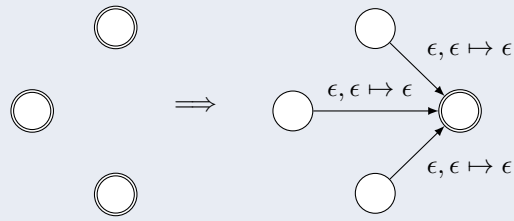


2.3.2. PDA ⇒ CFG

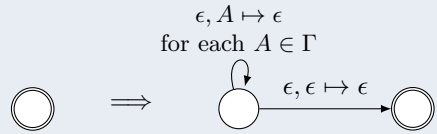
Now, we consider the other half of our main result. In order to convert a pushdown automaton to a context-free grammar, we must first ensure the pushdown automaton has certain properties: namely, the pushdown automaton must have a single accepting state, it must empty its stack before accepting, and each transition of the pushdown automaton must either push to or pop from the stack, but not both simultaneously. Let us refer to a pushdown automaton with these properties as a *simplified pushdown automaton*.

Fortunately, it's easy to convert from a pushdown automaton to a simplified pushdown automaton.

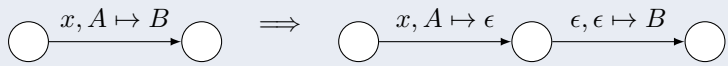
- To ensure the pushdown automaton has a single accepting state, we make each original accepting state non-accepting and add epsilon transitions from those states to a new single accepting state.



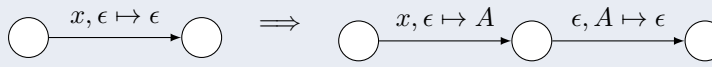
- To ensure the pushdown automaton empties its stack before accepting, we add a state immediately before the accepting state that removes all symbols from the stack.



- To ensure that each transition of the pushdown automaton either pushes to or pops from the stack, but not both, we split each transition that both pushes and pops into two separate transitions.



Additionally, if we have an epsilon transition that neither pushes nor pops, then we replace it with two "dummy" transitions that push and then immediately pop the same stack symbol.



With a simplified pushdown automaton, we can now perform the conversion to a context-free grammar.

Lemma 2.20

Given a simplified pushdown automaton $\mathcal{M}$ recognizing a language $L(\mathcal{M})$, there exists a context-free grammar G such that $L(G) = L(\mathcal{M})$.

Proof. Suppose we are given a simplified pushdown automaton $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}})$. We will construct a context-free grammar $G = (V, \Sigma_G, R, S)$ that generates the language recognized by $\mathcal{M}$.

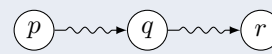
For each pair of states p and q in $\mathcal{M}$, our grammar will include a rule A_{pq} that simulates the computation of $\mathcal{M}$ starting in state p with some stack contents and ending in state q with the same stack contents. (Note that the stack may be manipulated during this computation; we just ensure that the contents of the stack are the same at the beginning and the end.)

We construct G in the following way:

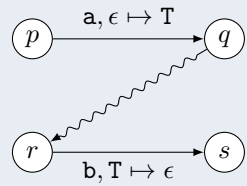
- The set of nonterminal symbols is $V = \{A_{pq} \mid p, q, \in Q\}$.
- The set of terminal symbols is $\Sigma_G = \Sigma$.
- The start nonterminal is $S = A_{q_0 q_{\text{accept}}}$ (i.e., the rule corresponding to the computation starting in state q_0 and ending in state q_{accept}).
- The set of rules R consists of three types of rules:
  1. For each state $q \in Q$, add the rule $A_{qq} \to \epsilon$ to R.

  2. For each triplet of states $p, q, r \in Q$, add the rule $A_{pr} \to A_{pq}A_{qr}$ to R.

  3. For each quadruplet of states $p, q, r, s \in Q$, input symbols $\mathtt{a}, \mathtt{b} \in \Sigma \cup \{\epsilon\}$, and stack symbol $\mathtt{T} \in \Gamma$, if $(q, \mathtt{T}) \in \delta(p, \mathtt{a}, \epsilon)$ and $(s, \epsilon) \in \delta(r, \mathtt{b}, \mathtt{T})$, then add the rule $A_{ps} \to \mathtt{a}A_{qr}\mathtt{b}$ to R.

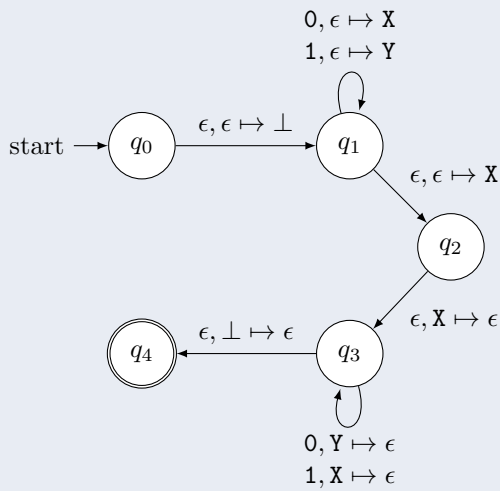


The first type of rule is a "dummy" rule that essentially corresponds to staying in the state q and adding nothing to the derivation. The second type of rule breaks down the overall computation into smaller components, taking into account intermediate states. Finally, the third type of rule adds terminal symbols to the derivation depending on the components of the overall computation.

With these rules, we can establish that the rule $A_{q_0 q_{\text{accept}}}$ generates a word w if and only if, starting in the state q_0 with an empty stack, the computation of $\mathcal{M}$ on w ends in the state q_{accept} also with an empty stack. Therefore, w is generated by the context-free grammar G if $\mathcal{M}$ accepts w, and $L(G) = L(\mathcal{M})$ as desired. ■

Example 2.21

Consider the following simplified pushdown automaton $\mathcal{M}$, where $\Sigma = \{0, 1\}$ and $\Gamma = \{X, Y\}$:



This pushdown automaton recognizes words of the form $w \cdot \overline{w}^{\mathrm{R}}$, where $\overline{w}$ is w with 0s and 1s swapped.

We convert the pushdown automaton $\mathcal{M}$ to a context-free grammar G. Let $V = \{A_{00}, A_{01}, A_{02}, A_{03}, A_{04}, A_{11}, A_{12}, A_{13}, A_{14}, A_{22}, A_{23}, A_{24}, A_{33}, A_{34}, A_{44}\}$ and take $\Sigma_G = \Sigma$. We also take $S = A_{04}$, since q_0 is the initial state and q_4 is the accepting state of $\mathcal{M}$. Finally, we add the following rules to the rule set R:

- Type 1 rules: $A_{00} \to \epsilon$, $A_{11} \to \epsilon$, $A_{22} \to \epsilon$, $A_{33} \to \epsilon$, and $A_{44} \to \epsilon$.
- Type 2 rules:

$$
\begin{aligned}
&A_{01} \to A_{00}A_{01} \mid A_{01}A_{11}\\
&A_{02} \to A_{00}A_{02} \mid A_{01}A_{12} \mid A_{02}A_{22}\\
&A_{03} \to A_{00}A_{03} \mid A_{01}A_{13} \mid A_{02}A_{23} \mid A_{03}A_{33}\\
&A_{04} \to A_{00}A_{04} \mid A_{01}A_{14} \mid A_{02}A_{24} \mid A_{03}A_{34} \mid A_{04}A_{44}\\
&A_{12} \to A_{11}A_{12} \mid A_{12}A_{22}\\
&A_{13} \to A_{11}A_{13} \mid A_{12}A_{23} \mid A_{13}A_{33}\\
&A_{14} \to A_{11}A_{14} \mid A_{12}A_{24} \mid A_{13}A_{34} \mid A_{14}A_{44}\\
&A_{23} \to A_{22}A_{23} \mid A_{23}A_{33}\\
&A_{24} \to A_{22}A_{24} \mid A_{23}A_{34} \mid A_{24}A_{44}\\
&A_{34} \to A_{33}A_{34} \mid A_{34}A_{44}
\end{aligned}
$$

- Type 3 rules:

$$
\begin{aligned}
&A_{13} \to \mathtt{0}A_{13}\mathtt{1} \mid \mathtt{1}A_{13}\mathtt{0} \mid \epsilon A_{22}\epsilon \text{ (or just } A_{22})\\
&A_{04} \to \epsilon A_{13}\epsilon \text{ (or just } A_{13})
\end{aligned}
$$

As an illustration, let's see how G derives an example input word $\mathtt{001011}$. Beginning from the start nonterminal $A_{q_0 q_{\text{accept}}} = A_{04}$, the derivation proceeds in the following way:

$$
\begin{aligned}
A_{04} &\Rightarrow A_{13}\\
&\Rightarrow \mathtt{0}A_{13}\mathtt{1}\\
&\Rightarrow \mathtt{0}\,\mathtt{0}A_{13}\mathtt{1}\,\mathtt{1}\\
&\Rightarrow \mathtt{00}\,\mathtt{1}A_{13}\mathtt{0}\,\mathtt{11}\\
&\Rightarrow \mathtt{001}\,A_{22}\,\mathtt{011}\\
&\Rightarrow \mathtt{001}\,\epsilon\,\mathtt{011} = \mathtt{001011}.
\end{aligned}
$$

2.3.3. CFG = PDA

Since we know by Definition 2.4 that a language is context-free if there exists a context-free grammar generating the language, we can combine the previous two lemmas to get the main result of this section.

Theorem 2.22

A language C is context-free if it satisfies any of the following equivalent properties:

1. There exists a context-free grammar G such that $L(G) = C$; or
2. There exists a pushdown automaton $\mathcal{M}$ such that $L(\mathcal{M}) = C$.

We can think of this result as the final piece to obtain the context-free analogue of Kleene's theorem for the regular languages. Since context-free grammars generate context-free languages, and since context-free grammars can be converted to pushdown automata and vice versa, both models correspond to the exact same language class. Unfortunately, this result doesn't get a nice name like Kleene's theorem did, but perhaps the lack of a name is justified when you consider the diagram we get isn't as interesting as the one we had for the regular languages:

$$\text{CFG} \longleftrightarrow \text{PDA}$$

(Not exactly a *Scutum Fidei* as before, but maybe a *Gladius Fidei*?)

We're not yet finished, though. Thanks to the equivalence between context-free grammars and pushdown automata, we can establish an important result that relates the class of context-free languages to the class of regular languages.

Theorem 2.23

Every regular language is also a context-free language.

Proof. Every regular language is recognized by some finite automaton. Since a finite automaton is a pushdown automaton that does not use the stack, every regular language is also recognized by some pushdown automaton. Furthermore, by Theorem 2.22, every regular language is generated by some context-free grammar. Therefore, every regular language is context-free. ■

Of course, we already know that there exist some context-free languages that are not regular, so this inclusion only works in one direction.

2.4. CLOSURE PROPERTIES

Much like in the chapter on regular languages, here I intend to summarize the closure properties of various operations applied to context-free languages. This section may prove to be more interesting, since certain operations turn out not to be closed for the class of context-free languages.

Intersection. While it is true that context-free languages are closed under union, it is in fact *not* true that they're also closed under intersection.

Theorem 2.24

The class of context-free languages is not closed under intersection.

Proof. Consider the languages

$$L_1 = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^m \mid m, n \geq 0\} \text{ and}$$
$$L_2 = \{\mathtt{a}^m\mathtt{b}^n\mathtt{c}^n \mid m, n \geq 0\}.$$

Both of these languages are context-free, so if the class of context-free languages were closed under intersection, the language $L_1 \cap L_2$ must also be context-free. However, we can see that

$$L_1 \cap L_2 = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\}.$$

This language is not context-free, and we can reason informally about this fact as follows: we can use the stack of a pushdown automaton to count n `a`s and match these symbols to n `b`s, but after this point we can no longer use the stack to count an equal number of `c`s. ■

Remark. In the following section, we will formally prove that the language $L_1 \cap L_2 = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\}$ is non-context-free.

Complement. We saw in our study of regular language closure properties that, if closure holds under both union and intersection, then closure must also hold under complement by De Morgan's laws. Since the class of context-free languages is not closed under intersection, it is therefore also not closed under complement.

Theorem 2.25

The class of context-free languages is not closed under complement.

Proof. Follows as a consequence of non-closure of context-free languages

under intersection. ■

2.5. PROVING A LANGUAGE IS NON-CONTEXT-FREE

At the end of our discussion on regular languages, we saw that there exist certain languages that are nonregular, and we also saw that we can prove a language is nonregular by using the pumping lemma. One of the biggest obstacles we observed that results in a language being nonregular was, broadly speaking, having to count or otherwise keep track of symbols. Fortunately, by augmenting our machine model with a stack and creating a pushdown automaton, we were able to overcome this obstacle. Surely, this means that we can now recognize any language we want, right?

Well, not exactly. While the stack goes a long way in helping us to recognize more than just the class of regular languages, it isn't the magic solution we need in order to recognize *any* language. Consider, for example, the language

$$L_{\mathtt{a=b=c}} = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\}.$$

We know that a pushdown automaton can accept words of the form $\mathtt{a}^n\mathtt{b}^n$ by pushing one symbol to the stack for each `a` that is read, and then popping one symbol from the stack for each `b` that is read. When it comes to recognizing words of the form $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$, however, we run into a problem: after we read all of the `b`s in the word, our stack will be empty and we will therefore have forgotten the value of n by the time we have to count the `c`s! We also can't cheat our way around this problem by, for example, pushing two symbols to the stack for each `a` we read; if we try that, then reading either `b` or `c` would require us to pop the same symbol, and we can draw a conclusion that such an approach would result in the pushdown automaton accidentally accepting words where `b`s and `c`s are either out of order or having mismatched counts.

Thus, there do indeed exist languages that are not context-free, and so we require a technique to prove the non-context-freeness of a language. Fortunately, we're mostly familiar with such a technique already: the pumping lemma for regular languages is a special case of the more general pumping lemma for context-free languages.

2.5.1. The Pumping Lemma for Context-Free Languages

You might be wondering at this point what we mean by the pumping lemma for regular languages being a "special case". Since regular languages are, in a sense, simpler than context-free languages, our formulation of the pumping lemma for regular languages was accordingly simpler: pumping the middle portion of any sufficiently long word in a regular language results in us

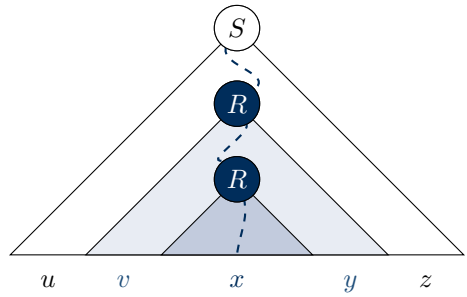


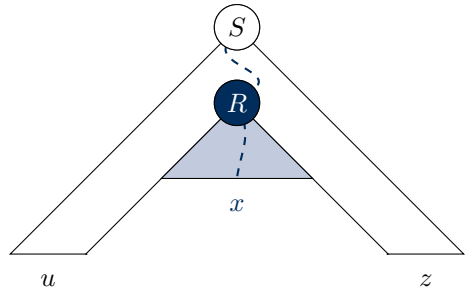
Figure 2.5. A path through a parse tree that visits some nonterminal symbol R more than once.

obtaining another word that also belongs to the language. Pumping this middle portion of the word essentially corresponds to us traversing a loop somewhere in the finite automaton recognizing the language.

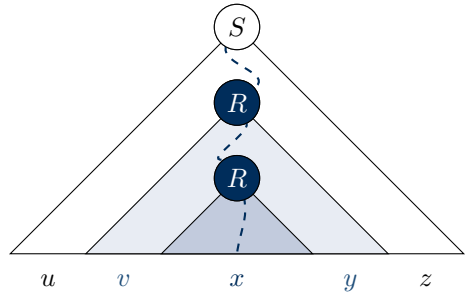
With context-free languages, however, we can't just pump one portion of the word. To understand why not, recall that we can represent the derivation of a word in a context-free language using a parse tree. If our word is sufficiently long, then the parse tree will be rather deep. Then, since we have only a finite number of both rules and nonterminal symbols, the pigeonhole principle tells us that there must exist some path from the root of the parse tree to a leaf of the parse tree where some nonterminal symbol R appears more than once along that path.

> *Remark.* "Sufficiently long" in this context is measured in terms of both the maximum number of symbols on the right-hand side of any rule of the context-free grammar and the size of the set of nonterminal symbols.

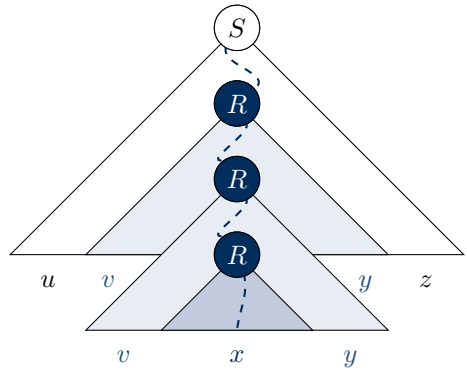
Suppose that we decompose our word w not into three parts as we did with the regular languages, but into five parts, denoted $uvxyz$. This decomposition, represented as a parse tree, is shown in Figure 2.5. Considering the subtree rooted at the first occurrence of R, we can regard everything "outside of" this subtree as the u and z portions of the word, while everything "inside of" this subtree comprises the vxy portion of the word. Since we know that R reappears at some point within this subtree, we can further consider the subtree rooted at the second occurrence of R, where everything "inside of" this subtree is the middle x portion of the word. Observe that we can repeat the first subtree rooted at R as many times as we want by appending the subtree to some later occurrence of R. In doing this, we are effectively pumping the segment of the subtree "between" both occurrences of R when we perform this repetition; that is, we are pumping the v and y portions of the word together, as depicted in Figure 2.6.



(a) "Pumping" the parse subtree rooted at R zero times.



(b) "Pumping" the parse subtree rooted at R one time.



(c) "Pumping" the parse subtree rooted at R two times.

Figure 2.6. Three examples of the pumping lemma for context-free languages applied to the parse tree depicted in Figure 2.5.

With this idea in mind, the formal statement of the pumping lemma for context-free languages is quite similar to that of the pumping lemma for regular languages, modulo the appropriate changes.

Lemma 2.26 (Pumping lemma for context-free languages)

For all context-free languages L, there exists $p \geq 1$ where, for all $w \in L$ with $|w| \geq p$, there exists a decomposition of w into five parts $w = uvxyz$ such that

1. $|vy| > 0$;
2. $|vxy| \leq p$; and
3. for all $i \geq 0$, $uv^i xy^i z \in L$.

I plan on adding a little more motivation for the proof here, before we jump right into the fine details.

Proof of Lemma 2.26. Let $G = (V, \Sigma, R, S)$ be a context-free grammar such that $L(G) = L$, and let $b \geq 2$ denote the *branching factor* of G; that is, the maximum number of terminal and nonterminal symbols occurring on the right-hand side of any rule of G. If the height of some parse tree of G is h, then the length of any word derived using that parse tree will be at most b^h.

Let $n = |V|$ denote the number of nonterminal symbols of G, and take $p = b^{n+1}$. By our earlier observation, any word generated by G whose parse tree contains no path with a repeated nonterminal must have length at most b^n. Moreover, since $b \geq 2$, we must have that $b^{n+1} > b^n$.

Let w be any word in $L(G)$ where $|w| \geq p$, and let T be a parse tree for w of minimal size. We know, again by our earlier observation, that T must have a height of at least $n + 1$. Choose some path in T with length at least $n + 1$, and take R to be the deepest nonterminal in the parse tree that occurs more than once in this path.

Decompose the word w into five parts, $uvxyz$, such that the subtree rooted at the upper occurrence of R has height at most $n + 1$ and the parts u and z are outside of the subtree rooted at the upper occurrence of R. We must have that $|vy| > 0$, since otherwise there would exist a smaller parse tree for w, which contradicts our assumption that T was of minimal size. Furthermore, the yield of this subtree, vxy, is a subword with length at most $p = b^{n+1}$. Finally, the word uxz is in L since we could replace the subtree rooted at the upper occurrence of R

with the subtree rooted at the lower occurrence of R that yields only the part x, and for $i \geq 1$, all words of the form uv^ixy^iz are in L since we can place copies of the subtree rooted at the upper occurrence of R at each subsequent occurrence of R. Therefore, all three conditions of the pumping lemma are satisfied. ■

Using the Pumping Lemma. Just like before, we can write a proof that some language is non-context-free by simply following a common set of steps. As a consequence, all non-context-freeness proofs share a similar structure. To see such an example of a proof, let's revisit the language we introduced at the beginning of this section.

Example 2.27

Let $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, and consider the language

$$L_{\mathtt{a}=\mathtt{b}=\mathtt{c}} = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\}.$$

We will use the pumping lemma to show that this language is non-context-free.

Assume by way of contradiction that the language is context-free, and let p denote the pumping constant given by the pumping lemma. We choose the word $w = \mathtt{a}^p\mathtt{b}^p\mathtt{c}^p$. Clearly, $w \in L_{\mathtt{a}=\mathtt{b}=\mathtt{c}}$ and $|w| \geq p$. Thus, there exists a decomposition $w = uvxyz$ satisfying the three conditions of the pumping lemma.

Observe that the first condition of the pumping lemma requires that *either* part v or part y is nonempty; potentially both could be nonempty. We consider two cases, depending on the contents of the parts v and y of the word w:

1. Both part v and part y contain some number of a single alphabet symbol; that is, v contains only `a`s, only `b`s, or only `c`s, and likewise for y. (Note that v and y do not need to contain the *same* alphabet symbol; for example, v could contain only `a`s and y could contain only `b`s.)

   In this case, pumping v and y once to obtain the word uv^2xy^2z results in the word containing unequal numbers of `a`s, `b`s, and `c`s. This violates the third condition of the pumping lemma.

2. Either part v or part y contains some number of multiple alphabet symbols; that is, either v or y contains both `a`s and `b`s, or both `b`s and `c`s.

In this case, pumping v and y once to obtain the word uv^2xy^2z results in the word containing symbols out of order. This violates the third condition of the pumping lemma.

In all cases, one of the conditions of the pumping lemma is violated. As a consequence, the language cannot be context-free.

By a similar line of reasoning, we can show that the language

$$L_{\mathrm{a=c/b=d}} = \{\mathtt{a}^n\mathtt{b}^m\mathtt{c}^n\mathtt{d}^m \mid m, n \geq 1\}$$

is non-context-free, since there's no way for us to separate the counts of as/cs and bs/ds using only one stack.

Recall that, before, we used the pumping lemma for regular languages to show that the language $L_{\mathrm{pal}} = \{ww^{\mathrm{R}} \mid w \in \Sigma^*\}$ was nonregular. We can easily show that the language of palindromes is context-free. However, suppose we modify the language of palindromes so that the reversed occurrence of the word w is instead just a repetition of w. This "doubled" language has an almost identical structure to the language of palindromes, but it happens to be non-context-free!

Example 2.28

Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, and consider the language

$$L_{\mathrm{double}} = \{ww \mid w \in \Sigma^*\}.$$

We will use the pumping lemma to show that this language is non-context-free.

Assume by way of contradiction that the language is context-free, and let p denote the pumping constant given by the pumping lemma. We choose the word $s = \mathtt{a}^p\mathtt{b}^p\mathtt{a}^p\mathtt{b}^p$. Clearly, $s \in L_{\mathrm{double}}$ and $|s| \geq p$. Thus, there exists a decomposition $s = uvxyz$ satisfying the three conditions of the pumping lemma.

Observe that the second condition of the pumping lemma requires that $|vxy| \leq p$. Here, we will consider two cases, depending on the contents of the middle portion vxy of the word s:

1. If vxy occurs entirely within the first half of s (that is, within the first occurrence of $\mathtt{a}^p\mathtt{b}^p$), then as a consequence of the fact that $|vxy| \leq p$, we must have one of the following subcases: vxy contains all as, vxy contains all bs, or vxy contains both as and bs, where all as occur before bs.

   In any of these three subcases, pumping v and y once to obtain the word uv^2xy^2z results in the first half of the word differing from

the second half of the word. We can make an analogous argument if vxy occurs entirely in the second half of s. This violates the third condition of the pumping lemma.

2. If vxy straddles both halves of s, then v and y must contain different symbols as a consequence of the fact that $|vxy| \leq p$. Pumping v and y down to obtain the word $uv^0xy^0z = uxz$ results in the first half containing fewer `b`s than the second half, and the second half containing fewer `a`s than the first half. This violates the third condition of the pumping lemma.

In all cases, one of the conditions of the pumping lemma is violated. As a consequence, the language cannot be context-free.

2.5.2. Ogden's Lemma ◇

Sometimes the pumping lemma fails in that non-context-free languages may satisfy the lemma's conditions (see, e.g., Wise [1976] and Horváth [1978]). Ogden's lemma [1968] is a stronger formulation of the pumping lemma and, although it is still imperfect (see, e.g., Boasson and Horváth [1978], Bader and Moura [1982], and Kracht [2004]), it allows us to establish non-context-freeness for more languages as well as to prove inherent ambiguity rather more easily.

Summary. At this point, let's now revisit the diagram we introduced in Figure 1.12 and add to it the class of context-free languages. On the one hand, we know that all of the languages we previously showed to be nonregular belong to our new context-free class, but on the other hand, we also now know that there exist languages that aren't context-free. Therefore, sadly, our diagram is still incomplete, but the end is in sight: we will take care of the final classes in the next chapter.

CHAPTER NOTES

A COMPREHENSIVE EARLY SURVEY of the theory of context-free languages can be found in the book by Ginsburg [1966].

2.1. The study of context-free grammars has its origins in the more general study of *phrase structure grammars*, which are a means of performing *immediate constituent analysis*; that is, dividing up a sentence into constituent parts, much like we did with the parse tree in Figure 2.2. In linguistics, immediate constituent analysis dates back to the work

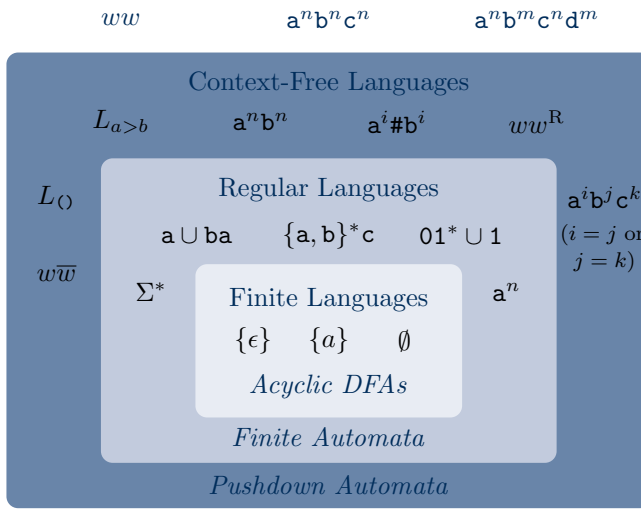


Figure 2.7. The hierarchy of language classes and models of computation, as we know it currently.

of Bloomfield [1933], and attempts to formalize the notion were made by Harris [1946] and Wells [1947]. One can argue that the point at which linguistics and mathematics merged began with the work of Chomsky [1956], which was inspired by the earlier *canonical systems* of Post [1943] and the *string rewriting systems* of Thue [1914].

The name "context-free" seems to be due to Chomsky [1959b], although this terminology took some time to catch on: Chomsky previously referred to context-free grammars as "type 2 grammars" [1959a], while Bar-Hillel, Perles, and Shamir [1961] called them "simple phrase structure grammars" and Ginsburg and Rice [1962] referred to the languages generated by such grammars as "definable sets".

Parikh [1961] was the first to demonstrate the existence of inherently ambiguous context-free grammars; specifically, he showed that no unambiguous grammar exists for the context-free language

$$\{\mathtt{a}^n\mathtt{b}^m\mathtt{a}^{n'}\mathtt{b}^m \mid m, n, n' \geq 1\} \cup \{\mathtt{a}^n\mathtt{b}^m\mathtt{a}^n\mathtt{b}^{m'} \mid m, m', n \geq 1\}.$$

As we noted, Chomsky normal form was introduced by Chomsky [1959a]. In fact, Chomsky proved an even stronger result in his paper: every context-free language has a context-free grammar where not only do all rules take the form $A \to a$ or $A \to BC$, where $A, B, C \in V$, $a \in \Sigma$, and $B \neq C$, but also if the grammar has rules of the form $A \to \alpha_1 B \alpha_2$ and $A \to \gamma_1 B \gamma_2$, then either $\alpha_1 = \gamma_1 = \epsilon$ or $\alpha_2 = \gamma_2 = \epsilon$.

Lange and Leiß [2009] observed a connection between the order in which steps are applied to convert a grammar to Chomsky normal form and the size of the resultant grammar, where size is measured in terms of the number of symbols needed to write the grammar. If we denote the size of the original grammar by $|G|$, a suboptimal ordering of steps (specifically, where **DEL** comes before **BIN**) produces a resultant grammar having a size of $2^{2|G|}$ in the worst case, while the order in which we apply the steps here produces a grammar of size $|G|^2$ in the worst case. It turns out we cannot do better than this quadratic factor, which is incurred via the **UNIT** step.

2.2. The earliest mention of a machine that uses pushdown storage to perform a computation seems to be in an article by Burks, Warren, and Wright [1954], where the authors give an algorithm to check the well-formedness of a parenthesis-free Boolean formula; that is, a formula written in the Polish notation of Łukasiewicz [1929]. A more explicit construction of a machine using pushdown storage was given by Newell and Shaw [1957], who described a machine that manages memory usage by keeping addresses of free memory locations on an "available-space list". The addresses of recently freed memory locations are pushed to the front of this list, and memory needs are handled by popping addresses, again, from the front of the list. (On an unrelated note, this so-called "Logic Theorist" machine described by Newell, Shaw, and Herbert A. Simon was the first description of a computer that makes use of automated reasoning, and is considered by some to be the first-ever implementation of artificial intelligence.)

The term "pushdown store" itself is due to Oettinger [1961]. The first use of pushdown storage pertaining to the class of context-free languages seems to be due to Chomsky [1962].

We observed that nondeterministic pushdown automata are more powerful than deterministic pushdown automata in that the nondeterministic model recognizes more languages. The study of deterministic context-free languages was initiated by Ginsburg and Greibach [1966].

2.3. The equivalence in recognition power between context-free grammars and pushdown automata was established by Chomsky [1962] and Chomsky and Schützenberger [1963], and independently by Evey [1963a,b]. However, each of their approaches to showing equivalence is rather more complex than the one presented here.

2.4. Closure of the class of context-free languages under union, as well as non-closure under intersection and complement, was established by Scheinberg [1960].

2.5. The pumping lemma for context-free languages is due to Bar-Hillel, Perles, and Shamir [1961].

We know from the previous chapter that the language of binary representations of prime numbers, L_{primes2}, is nonregular. Hartmanis and Shank [1968] further established that this language cannot be accepted by any pushdown automaton, and therefore it is also non-context-free. The same result, proved via a different approach, was published independently by Schützenberger [1968].

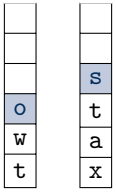
CHAPTER THREE

DECIDABLE AND SEMIDECIDABLE LANGUAGES

When we introduced pushdown automata, we saw that augmenting our machine with a stack gave it a rudimentary form of memory, and it could use this memory to recognize a larger set of languages. However, despite the fact that the stack has an unbounded capacity (and is therefore able to store as many symbols as it wants), we are still limited by the fact that it's a stack, meaning it can only access the top symbol at any given time.

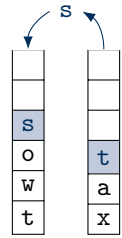
So, how do we overcome this limitation? Let's try adding not one but *two* stacks to our machine! It may not sound like a huge or meaningful change—after all, what can we get with another stack that we didn't already have with the first stack?—but, just like with heads, it turns out that two stacks are better than one.

Suppose we now have two stacks available for us to use. For the purposes of this example, the stacks have been initialized with some symbols already in them.

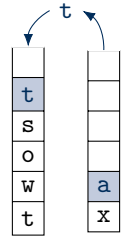


Even with two stacks, we can still only access the top symbol of each stack: in the first stack, we can access the symbol `o`, and in the second stack, we can access the symbol `s`.

But what happens if we use the two stacks in tandem? Suppose we pop one symbol from the second stack, say `s`, and push it to the first stack.



Now, we have access to the symbol `t` that was previously beneath `s` in the second stack, and we haven't lost the symbol `s` since it's safely stored in the first stack! Similarly, if we pop that symbol `t` from the second stack and push it to the first stack, we can get access to the symbol `a` in the second stack.



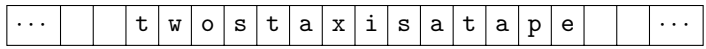
It's looking like having two stacks is far more meaningful than we might've initially thought. We can push and pop symbols between the two stacks in order to get access to *any* symbol we've stored in the machine's memory, instead of only the most recent symbol at the top.

Indeed, if we took our two stacks and we aligned them horizontally in such a way that the "bottoms" of each stack were on the left and right edges...

...we would get a new form of storage: a *tape*.

With a tape, we can move left and right through each cell and access each symbol stored on the tape whenever we want. Indeed, this is exactly what we were doing when we pushed and popped symbols between our two stacks: whatever symbol we're reading on our tape is the symbol found at the top of our second stack.

Just like our stacks have unbounded capacity, our tape has infinite length. We can write as many symbols to the tape as we want, and we can write them to either the left side or the right side of the tape.



Since we can now access any symbol of the tape that we want, tape cells that do not contain a symbol become an important consideration. With stacks, we need not worry about blank spaces: since we can only push to or pop from the top of a stack, we have no opportunity to leave gaps and so we never encounter a situation where two symbols are separated by a blank space. If we remove a symbol from a tape, however, the symbols to the left and to the right of the removed symbol do not adjust their positions to compensate. Thus, we notate blank spaces on a tape using the symbol ␣ .

···	␣	␣	t	w	o	s	t	a	x	␣	=	␣	t	a	p	e	␣	␣	···

Now that we know the basics of working with tapes, we're ready to replace the stack on our machine with a tape. In doing so, we will obtain one of the most powerful abstract models of computation possible: even more powerful than any real-world computer!

3.1. TURING MACHINES

THE FOCUS OF THIS SECTION, the *Turing machine*, is a model of computation that consists of three main components: a finite automaton and an infinite-length tape, connected to one another by an input head. The finite automaton keeps track of where we are in the computation, while the tape serves as the machine's memory throughout the computation.

At the beginning of a computation, the tape holds the input word given to the Turing machine, and all other cells of the tape are blank. Since the input word is initially stored on the tape, we can assume that the input alphabet Σ is a subset of the tape alphabet Γ. The input head of the Turing machine starts on the leftmost symbol of the input word. It can move along the tape, and it can both read from and write to cells of the tape. In this way, we can use the tape to store and modify not only the input word, but also any auxiliary information we need to use during the computation.

To model the movement of the Turing machine's input head along the tape, we must account for the direction of movement in the transition function. To figure out the next step of the computation, our transition function will take as input our current state and the tape symbol the input head reads in the current cell, and it will produce as output the state we will transition to, the tape symbol the input head will write to the current cell, and the direction in which the input head will move: one cell leftward (L) or one cell rightward (R).

Another key difference that sets Turing machines apart from finite automata and pushdown automata is in how they accept or reject input words. Unlike finite automata or pushdown automata, which eventually run out of symbols by reaching the end of the input word, a Turing machine could possibly read the symbols on its tape as many times as it wants.

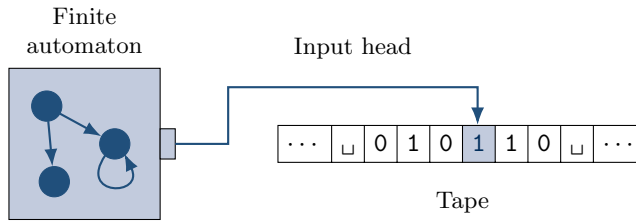


Figure 3.1. An illustration of a Turing machine.

Therefore, we must fix two special "accept" and "reject" states where, if the computation of the Turing machine ever enters one of those states, it immediately halts the computation and accepts or rejects the input word accordingly. Note that if the Turing machine doesn't enter either of these states during its computation, then it will continue to compute indefinitely.

Apart from these changes, the formal definition of a Turing machine is quite similar to our definitions for finite automata and pushdown automata.

Definition 3.1 (Turing machine)

A Turing machine is a tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, where

- Q is a finite set of *states*;
- Σ is the *input alphabet* (where $\text{␣} \notin \Sigma$);
- Γ is the *tape alphabet* (where $\text{␣} \in \Gamma$ and $\Sigma \subseteq \Gamma$);
- $\delta : (Q \setminus \{q_{\text{accept}}, q_{\text{reject}}\}) \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the *transition function*;
- $q_0 \in Q$ is the *initial* or *start state*;
- $q_{\text{accept}} \in Q$ is the *final* or *accepting state*; and
- $q_{\text{reject}} \in Q$ is the *rejecting state*, where $q_{\text{reject}} \neq q_{\text{accept}}$.

Figure 3.1 depicts a typical visualization of a Turing machine.

> *Remark.* Now is a good point for one more moment of grammatical pedantry. The present model of computation is called a "Turing machine", as it was named after the famed mathematician and father of computer science, Alan Turing. Unfortunately, for whatever reason, a nonzero percentage of the computer science community refers to this model as a "Turning machine". This is, of course, incorrect, and offenders should be given a biography of Turing to read immediately.

Example 3.2

Consider the language $L_{\mathtt{a=b}} = \{\mathtt{a}^n\mathtt{b}^n \mid n \geq 1\}$. Even though we know this language is context-free, and is therefore recognized by a pushdown automaton, let's construct a Turing machine recognizing the language.

The idea behind our Turing machine is as follows. Given an input word of the form $\mathtt{a}^n\mathtt{b}^n$ on the tape, the input head will move back and forth, replacing all as with Xs and all bs with Ys. In a sense, the input head is "marking" as and bs as it sees them. Each time the input head replaces an a with an X, it will move rightward in an attempt to find a matching b that it can replace with a Y. The input head will then move leftward and repeat the process until no more as remain.

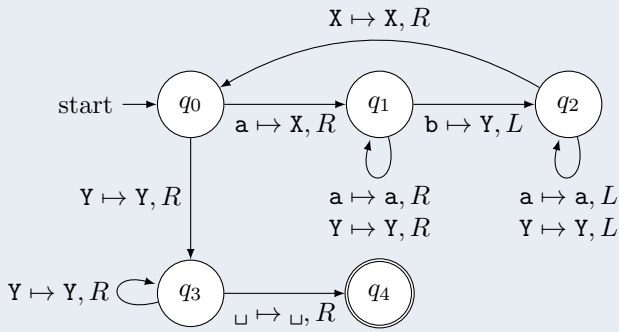
We formally define the Turing machine as follows:

- $Q = \{q_0, q_1, q_2, q_3, q_4, q_R\}$;
- $\Sigma = \{\mathtt{a}, \mathtt{b}\}$;
- $\Gamma = \{\mathtt{a}, \mathtt{b}, \mathtt{X}, \mathtt{Y}, \sqcup\}$;
- $q_0 = q_0$;
- $q_{\text{accept}} = q_4$;
- $q_{\text{reject}} = q_R$; and
- δ is specified by the following table:

Γ	a	b	X	Y	$\sqcup$
q_0	$(q_1, \mathtt{X}, R)$	—	—	$(q_3, \mathtt{Y}, R)$	—
q_1	$(q_1, \mathtt{a}, R)$	$(q_2, \mathtt{Y}, L)$	—	$(q_1, \mathtt{Y}, R)$	—
q_2	$(q_2, \mathtt{a}, L)$	—	$(q_0, \mathtt{X}, R)$	$(q_2, \mathtt{Y}, L)$	—
q_3	—	—	—	$(q_3, \mathtt{Y}, R)$	$(q_4, \sqcup, R)$
q_4	×	×	×	×	×
q_R	×	×	×	×	×

Note that we can't have any transitions from either state q_4 or q_R, since those are the accepting and rejecting states, respectively. Thus, we fill those rows with the symbol ×. Also, instead of indicating all undefined transitions explicitly, we just assume that any undefined transition in the table (—) automatically leads to state q_R.

We can draw this Turing machine graphically, just like a finite automaton or a pushdown automaton. To reduce the number of transitions we need to draw, we will omit the state q_R and all transitions leading to it, and we will just assume (again) that all transitions not included automatically lead to state q_R. The Turing machine looks like the following:



Now, suppose we give the input word aaabbb to this Turing machine. The actions of the Turing machine are depicted in Figure 3.2. The input head starts its computation in state q_0 on the leftmost symbol of the input word and, moving from the top to the bottom of each column, we highlight the state of the machine and the input head's current tape cell at each step. Since the computation halts in the accepting state q_4, we know that the machine accepts the word aaabbb.

3.1.1. Configurations and Accepting Configurations

You may have noticed in Figure 3.2 that the computation of the Turing machine on the given input word took up a *lot* of space on the page. This is because we had to draw the current state, the entire tape, and the position of the input head on the tape, all at each step. Fortunately, however, there is a more concise way to present this information.

All we need to specify a particular stage of the computation is the current state, the current tape contents, and the current input head position, and we can represent all of this using a single sequence of symbols. This sequence is called a *configuration* of the Turing machine. If the Turing machine is currently in a state q, its tape contains the symbols $X_1X_2 \cdots X_{i-1}X_i \cdots X_{n-1}X_n$, and its input head is at cell i of the tape, then the configuration of the Turing machine at this moment is written

$$X_1X_2 \cdots X_{i-1}qX_i \cdots X_{n-1}X_n.$$

If we can get from a configuration C_i to a configuration C_{i+1} in a single computation step, then we say that C_i *yields* C_{i+1} and we write $C_i \vdash C_{i+1}$. Formally, given $a, b, c \in \Gamma$, $u, v \in \Gamma^*$, and $q_i, q_j \in Q$, we say that uaq_ibv yields $uacq_jv$ if $\delta(q_i, b) = (q_j, c, R)$. We can define the notion of "yields" for leftward moves similarly.

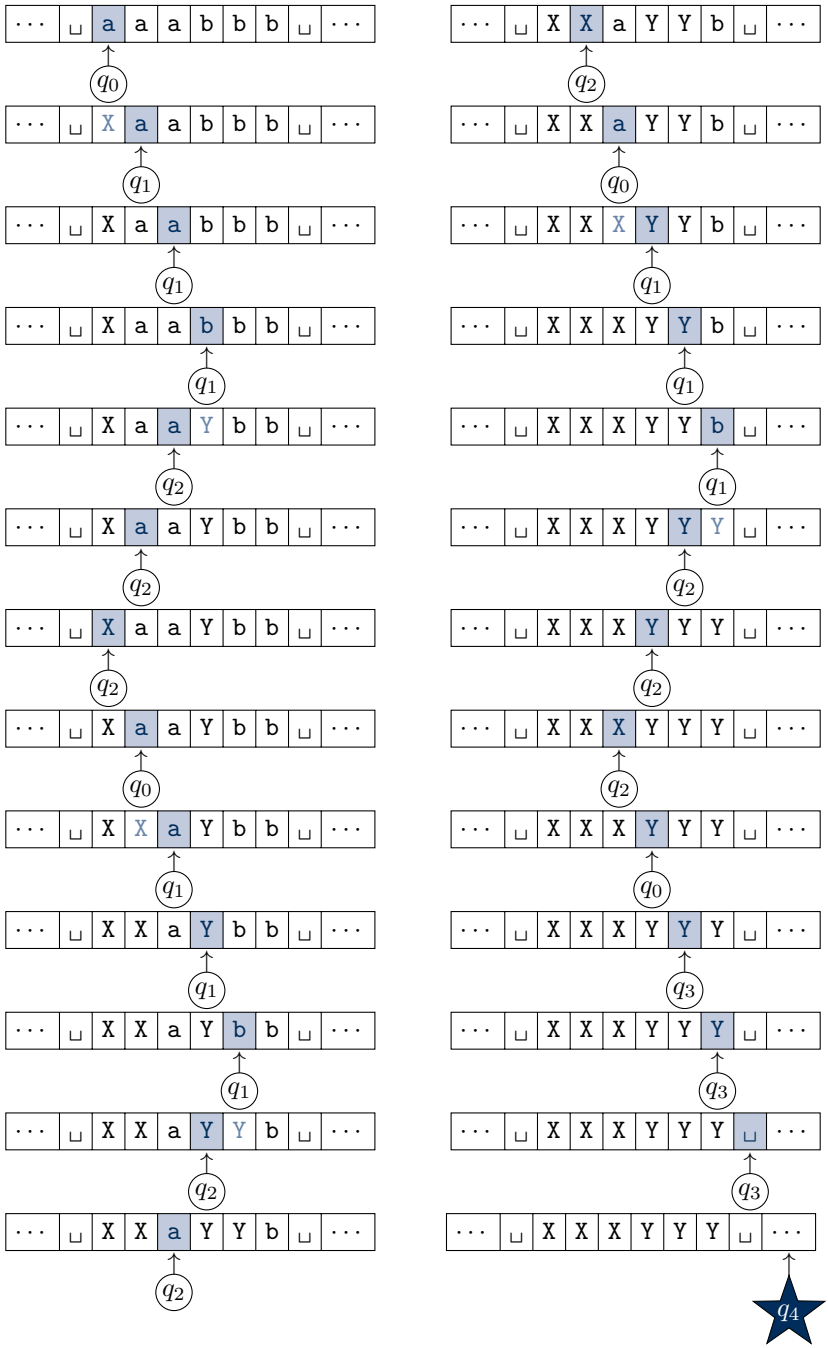


Figure 3.2. A trace of the behaviour of the Turing machine recognizing the language $L_{\mathtt{a}=\mathtt{b}}$ from Example 3.2.

Example 3.3

Recalling the Turing machine's computation from Example 3.2, the first five configurations of the machine are

$$q_0\texttt{aaabbb} \vdash \texttt{X}q_1\texttt{aabbb} \vdash \texttt{Xa}q_1\texttt{abbb} \vdash \texttt{Xaa}q_1\texttt{bbb} \vdash \texttt{Xa}q_2\texttt{aYbb} \vdash \cdots$$

By our earlier definition, we know that a Turing machine includes dedicated accept and reject states, and we know that the machine's computation either accepts or rejects once it enters the appropriate state. With the notion of configurations, we can specify exactly what it means for a Turing machine to accept or reject its input word.

If we have a Turing machine $\mathcal{M}$ and an input word w, then we say that the *start configuration* of $\mathcal{M}$ on w is q_0w. In this configuration, $\mathcal{M}$ is in its initial state q_0, and the input head of $\mathcal{M}$ is on the leftmost symbol of the input word.

Likewise, an *accepting configuration* is one where the current state of $\mathcal{M}$ is q_{accept}, and a *rejecting configuration* is one where the current state of $\mathcal{M}$ is q_{reject}. Note that, in either configuration, we only care about the state and not the tape contents. This is because once we enter the accepting or rejecting state, the computation immediately halts, and so the tape contents don't have any effect on the accepting or rejecting configuration.

We can now formally define what it means for the Turing machine $\mathcal{M}$ to accept its input word w.

Definition 3.4 (Accepting computation of a Turing machine)

Let $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ be a Turing machine, and let w be an input word. The Turing machine $\mathcal{M}$ accepts the input word w if there exists a sequence of configurations $C_1, C_2, \ldots, C_n$ satisfying the following conditions:

1. C_1 is the start configuration of $\mathcal{M}$ on w;
2. $C_i \vdash C_{i+1}$ for all $1 \leq i \leq (n-1)$; and
3. C_n is an accepting configuration.

We can write a similar definition for a rejecting computation of a Turing machine by considering rejecting configurations in the third condition.

3.1.2. Language of a Turing Machine

Just as with other types of automata, every Turing machine recognizes some language. The set of all input words accepted by a Turing machine $\mathcal{M}$ is

referred to as the language of the machine $\mathcal{M}$, written $L(\mathcal{M})$. However, the class to which some Turing machine's language belongs is determined by the machine's behaviour on each input word.

We noted earlier on that, unlike with finite automata and pushdown automata, a Turing machine cannot run out of input symbols. Indeed, it can read the symbols on its tape as many times as it wants. The machine's computation immediately halts once the machine enters either the accepting or the rejecting state, but there's no guarantee that it will ever enter either of these states during its computation. We must account for the possibility that the machine simply never ends its computation; that is, the machine enters a *loop*.

Taking this outcome into account, we see that Turing machines can recognize two "types" of languages: languages where the Turing machine always gives us an accept/reject answer for each input word, and languages where the Turing machine might enter a loop and give no answer for certain input words.

Decidable Languages. Let's first focus on the scenario where the machine always either accepts or rejects every input word its given. If the Turing machine accepts all words that belong to its language and rejects all words that don't belong to its language, then we say that the machine *decides* its language.

Definition 3.5 (Decidable language)

Given a Turing machine $\mathcal{M}$, we say that the language $L(\mathcal{M})$ is decidable if,

- whenever $w \in L(\mathcal{M})$, then $\mathcal{M}$ accepts w; and
- whenever $w \notin L(\mathcal{M})$, then $\mathcal{M}$ rejects w.

We denote the class of decidable languages by D. In the literature, the class of decidable languages is sometimes called the class of *recursive languages*, where the term "recursive" comes from the origins of computer science and its connections to recursive functions and sets.

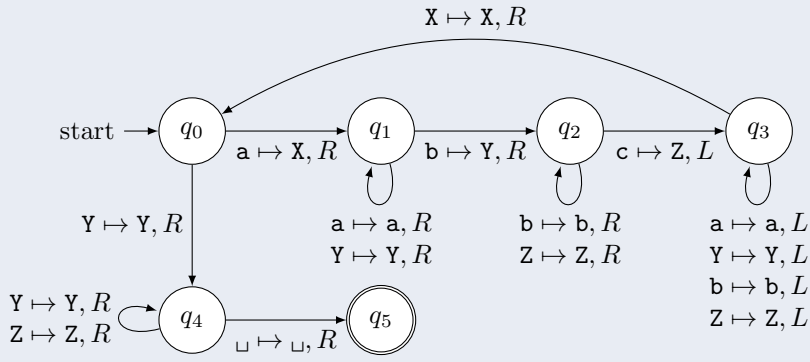
Example 3.6

Consider the language $L_{\mathtt{a=b=c}} = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 1\}$. We know that this language is non-context-free, so we can't construct a pushdown automaton recognizing the language. Let's instead construct a Turing machine recognizing the language.

Our Turing machine will function in much the same way as the

machine we constructed to recognize words of the form $\mathtt{a}^n\mathtt{b}^n$; moving from left to right, we will match symbols up to one another by replacing the symbols on the tape.

Suppose our alphabets are $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ and $\Gamma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{X}, \mathtt{Y}, \mathtt{Z}, \sqcup\}$. The Turing machine, then, will look like the following:



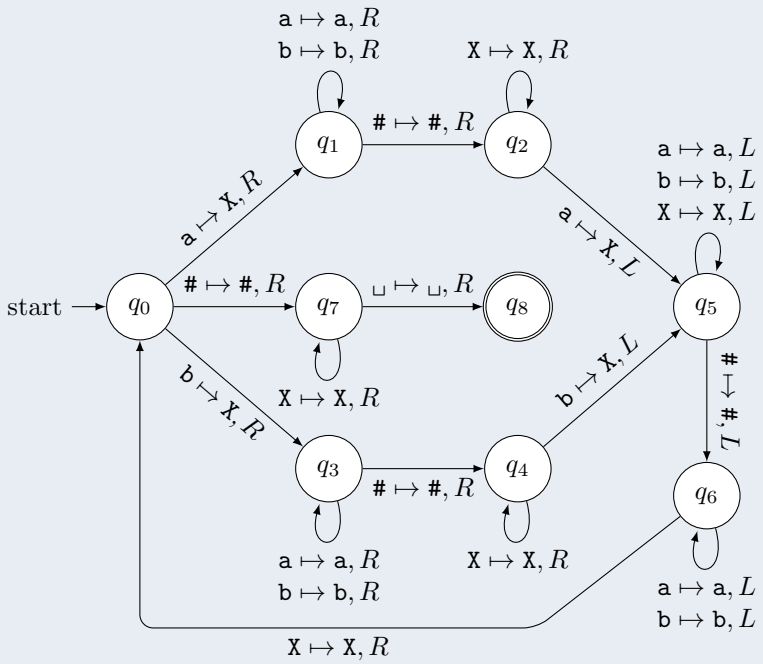
This Turing machine decides the language $L_{\mathtt{a=b=c}}$ because (i) if some input word w belongs to this language, then the machine will accept it; and (ii) if some input word w does not belong to this language, then the machine will (implicitly) go to the rejecting state q_{reject}.

Example 3.7

Consider the language $L_{\text{double\#}} = \{w\mathtt{\#}w \mid w \in \Sigma^*\}$. This language is very similar to the language L_{double}, which we previously showed was non-context-free. The key difference with this language, though, is the presence of a separator symbol # between the two occurrences of w.

Let's construct a Turing machine for the language $L_{\text{double\#}}$. The idea behind this Turing machine is to move back and forth between each copy of w, marking off each symbol in the first copy and using the states of the machine to "remember" this symbol as we move to the second copy. If the machine is able to match every symbol in both copies of the word, then it accepts. Otherwise, it (implicitly) rejects.

Suppose our alphabets are $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ and $\Gamma = \{\mathtt{a}, \mathtt{b}, \mathtt{X}, \sqcup\}$. The Turing machine will look like the following:



This Turing machine decides the language $L_{\text{double\#}}$ because it accepts all words of the form $w\#w$ and (implicitly) rejects all other words.

Just like we were able to prove in the previous chapter that every regular language is context-free by showing that each regular language is recognized by some finite automaton, we can similarly prove that every context-free language is decidable by showing that for any context-free language we're given, there exists some Turing machine capable of deciding that language.

Theorem 3.8

Every context-free language is also a decidable language.

I still need to write this proof.

Semidecidable Languages. As we mentioned before, unlike with finite automata and pushdown automata where the computation of the machine immediately ends after running out of input symbols to read, there's no requirement specifying that the computation of a Turing machine must come to an end. It's possible that, given certain input words, the machine

could find itself in an infinite loop as it never transitions to its accepting or rejecting states. In this case, the Turing machine still has a language—as always, it's the set of input words for which the Turing machine has an accepting computation—but we can't say that the Turing machine decides this language, since it may not explicitly reject all words not belonging to its language. If this is the case, then we instead say that the machine *semidecides* its language.

Definition 3.9 (Semidecidable language)

Given a Turing machine $\mathcal{M}$, we say that the language $L(\mathcal{M})$ is semidecidable if,

- whenever $w \in L(\mathcal{M})$, then $\mathcal{M}$ accepts w; and
- whenever $w \notin L(\mathcal{M})$, then either $\mathcal{M}$ rejects w or $\mathcal{M}$ enters an infinite loop.

Remark. The prefix "semi-" comes from a Latin prefix meaning "half". Thus, the word "semidecidable" can be taken to mean "half decidable", in that a Turing machine that semidecides its language is only capable of properly deciding half of the possible outcomes: acceptance, but not rejection.

We denote the class of semidecidable languages by SD. In the literature, the class of semidecidable languages is sometimes called the class of *recognizable languages* or *recursively enumerable languages*. The term "recursively enumerable" again comes from a connection to mathematics and the notion of recursively enumerable sets. In a machine-oriented context, a recursively enumerable language is one for which a Turing machine can list (or *enumerate*) every word in the language.

We can come up with any number of artificial examples of semidecidable languages, but few such examples are interesting; in fact, it's often the case that making a small change to the Turing machine recognizing such a language results in that language becoming decidable anyway. Therefore, we'll skip the examples for now. In the next lecture, we'll focus on some more natural examples of semidecidable languages, where "natural" means that the language models some inherent property or quality of the machine that recognizes it.

For now, though, we can still prove some nice properties about this class of languages. Directly from the definitions, we get a connection between decidable and semidecidable languages.

Theorem 3.10

Every decidable language is also a semidecidable language.

Proof. Both decidable languages and semidecidable languages are recognized by Turing machines. If some Turing machine $\mathcal{M}$ decides its language, then by Definition 3.5, every input word $w \in L(\mathcal{M})$ is accepted by $\mathcal{M}$ and every input word $w \notin L(\mathcal{M})$ is rejected by $\mathcal{M}$. But this behaviour exactly matches that specified in Definition 3.9, and so we can say that $\mathcal{M}$ semidecides its language as well. ■

3.1.3. Computing Functions

When we think about the definition of a Turing machine, we realize that it has all the components we would expect a standard computer to have: a Turing machine can receive input, it can process this input, it can store data on its tape, and it can produce output by writing something to its tape before finishing its computation. It is evident that Turing machines are a step above our previous models of finite automata and pushdown automata. While these weaker models can receive input, the most they can do with that input is read the symbols and eventually give us one of two answers: "accept" or "reject". In essence, finite automata and pushdown automata are more *recognizers* than they are *computers*.

Since a Turing machine can do much more than recognize its input, we might as well use it to its full potential by performing actual computations, and among the simplest of computations is taking a value n and producing a value $f(n)$ for some mathematical function f. This gives rise to the notion of *computable functions*, which, as the name suggests, are functions that can be computed on a Turing machine.

Definition 3.11 (Computable function)

A function $f\colon \Sigma^* \to \Sigma^*$ is computable if there exists some Turing machine that, given an input word w, finishes its computation with $f(w)$ written to its input tape and nothing else.

There are many examples of computable functions. Just to name a few: every constant function, arithmetic functions such as addition and multiplication, number-theoretic functions such as greatest common divisor and least common multiple, and every function with a finite domain are all computable.

Let's now consider some concrete examples of how a Turing machine computes a function.

Example 3.12

The function $f(n) = 2n$ is a computable function. A Turing machine $\mathcal{M}_{2n}$ can take as input a word consisting of n copies of `1` and produce as output a word consisting of $2n$ copies of `1` in the following way:

1. Erase the leftmost `1` from the tape.
2. Move rightward past the remaining `1`s in the input word, plus one blank cell, plus any existing `1`s in the output word.
3. Write a `1` to the rightmost blank cell, then move rightward and write a second `1`.
4. Move leftward to the leftmost `1` in the input word.
5. Repeat these steps n times.

I plan to add a couple more examples of computable functions here; say, computing $f(n) = n!$, or showing how a Turing machine would add two numbers together.

Many computer scientists have written about the "right" kind of characterization, or set of properties, that ensures a function is computable by a Turing machine. For example, Enderton [1977] specifies three properties a function must possess in order for it to be computable:

1. The description of how to compute the function must be a finite-length list of exact instructions.
2. Given an input w in the domain of the function, the computation must produce the output $f(w)$ after a finite number of computation steps.
3. Given an input w not in the domain of the function, the computation may loop forever or get stuck, but it must not produce an incorrect output.

Looking back at our examples, we can see that each of the functions we studied obeys these three criteria, and so we can reasonably call these functions computable. The exercise of determining what makes a function computable straddles the border between computer science and philosophy; in Section 3.6, we'll dive into a much deeper discussion of what exactly a Turing machine is capable of computing.

3.2. VARIANTS OF TURING MACHINES

The definition of a Turing machine that we gave earlier in this lecture is by no means a canonical definition. When we chose to give our machine a deterministic transition function, or a single tape, or a two-way infinite tape, we made those choices simply to fix *some* definition of a Turing machine. Since we are just introducing Turing machines for the first time in this lecture, we decided to go with a relatively easy-to-understand definition: the single, two-way infinite tape allowed us to use our "two stacks" perspective to motivate the definition, and deterministic transition functions are more straightforward to reason about.

Just like we defined different types of finite automata, nothing is stopping us from modifying our definition of a Turing machine. However, one of the most remarkable results in theoretical computer science is that we can make pretty much *any* modification to our definition of a Turing machine, and it won't affect the recognition power of the model. The Turing machine is, in a sense, the Platonic form of a computer; nearly any variant definition we choose grants us sufficient power to recognize the large classes of decidable and semidecidable languages.

To illustrate, here we will make a number of modifications to our Turing machine definition, and we will then prove that each modified definition is equivalent to our original definition. This will give us a wide array of variant Turing machine models that we can choose from when we're trying to recognize certain languages.

3.2.1. Nondeterministic Turing Machines

The first natural modification we can make to our definition is to take the transition function δ to be nondeterministic. Just like with our other models of computation, a nondeterministic transition function for a Turing machine will map a pair of state and tape symbol to the power set of tuples of state, tape symbol, and input head movement.

Definition 3.13 (Nondeterministic Turing machine)

A nondeterministic Turing machine is a tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, where everything is defined as in Definition 3.1 except for the transition function, which is

$$\delta : (Q \setminus \{q_{\text{accept}}, q_{\text{reject}}\}) \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\}).$$

We saw that deterministic and nondeterministic finite automata are equivalent in terms of recognition power, while nondeterministic pushdown automata are able to recognize more languages than deterministic push-

down automata. With Turing machines, we return to equivalence: adding nondeterminism to a Turing machine does not give it more recognition power.

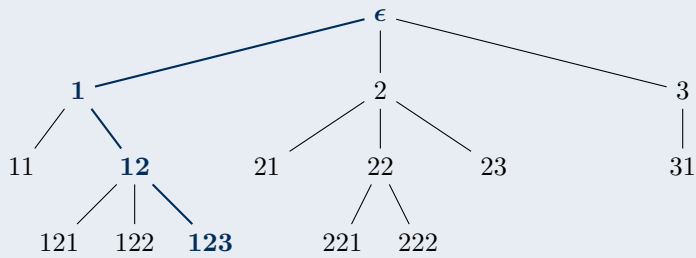
Theorem 3.14

Given a nondeterministic Turing machine $\mathcal{M}$, we can construct a deterministic Turing machine $\mathcal{M}'$ such that $L(\mathcal{M}') = L(\mathcal{M})$.

Proof. Let $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ be a nondeterministic Turing machine. We will construct a deterministic Turing machine $\mathcal{M}'$ that uses three tapes to simulate the nondeterministic computation of $\mathcal{M}$.

- The first tape will be the *input tape*, and it will contain the input word given to $\mathcal{M}$. The contents of the input tape will not be changed during the computation.
- The second tape will be the *simulation tape*, and it will simulate the contents of $\mathcal{M}$'s tape as $\mathcal{M}$ performs its nondeterministic computation.
- The third tape will be the *address tape*, and it will keep track of where we are in the nondeterministic computation tree of $\mathcal{M}$.

Before we proceed, let's consider how we can represent the nondeterministic computation tree of $\mathcal{M}$ on a linear form of storage like a tape. If we take b to denote the maximum number of branches in the tree (that is, the maximum number of nondeterministic transitions $\mathcal{M}$ can follow at any given point in its computation), then we can assign a unique address to each vertex of the tree over the alphabet $\Sigma_b = \{1, 2, \ldots, b\}$. The address is determined by tracing the branches we must follow in order to get from the root to that vertex; for example, the vertex with address 123 can be reached by starting at the root of the tree and taking the first branch, followed by the second branch, followed by the third branch. As a consequence of this convention, the root of the tree receives the address ϵ.



The address tape, then, contains symbols from the alphabet Σ_b. Each symbol on the address tape will tell $\mathcal{M}'$ which branch of the nondeterministic computation tree it must follow in its next computation step. Note that the contents of the address tape do not necessarily need to correspond to a vertex of the tree; if the address tape contains an invalid address, then $\mathcal{M}'$ simply aborts its attempted simulation of that branch.

Having defined the three tapes, we can describe how $\mathcal{M}'$ simulates the computation of $\mathcal{M}$:

1. Initialize the tapes in the following way:
   (a) Copy the contents of the input tape to the simulation tape; that is, write the input word w given to $\mathcal{M}$ to the simulation tape.
   (b) Leave the address tape empty.
2. Use the simulation tape to perform the following steps:
   (a) For each computation step of $\mathcal{M}$, read the next symbol of the address tape to determine which branch of the nondeterministic computation tree to follow.
   (b) If there are no more symbols remaining on the address tape, or if the address tape contains an invalid address, or if $\mathcal{M}$ enters a rejecting configuration, then abort the attempted simulation of this branch and go to step 3.
   (c) If $\mathcal{M}$ enters an accepting configuration, then accept w.
3. Write to the address tape the sequence of symbols over Σ_b that comes next in lexicographic order and go to step 2. ■

Since every deterministic Turing machine is a nondeterministic Turing machine that doesn't use nondeterminism during its computation, we immediately obtain the other direction of the relationship between these models. Therefore, we can conclude that deterministic and nondeterministic Turing machines are equivalent.

3.2.2. Multitape Turing Machines

In the proof of Theorem 3.14, we saw that we could simulate the computation of a nondeterministic Turing machine using a deterministic Turing machine with multiple tapes. It's natural to wonder whether including additional tapes gave us some kind of advantage in this simulation process—after all, giving our computational model two stacks instead of one stack led us to

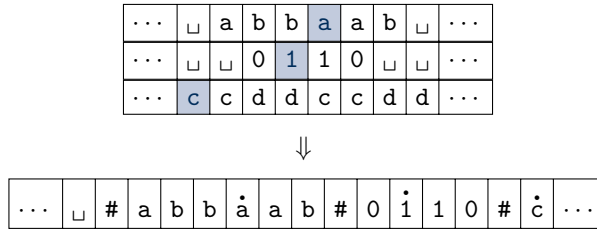


Figure 3.3. Simulating the computation of a k-tape Turing machine (top) with a single-tape Turing machine (bottom). The positions of the k physical input heads are indicated by highlighted cells, while the corresponding k virtual input heads are indicated by dots.

the idea of a Turing machine—but as it turns out, we can perform exactly the same computations using only a single tape.

First, let's define our *multitape Turing machine* model. Again, we only need to modify the transition function: instead of mapping a pair of state and *one* tape symbol to a tuple of state, *one* tape symbol, and *one* input head movement, we will transition on k tape symbols and k input head movements, where k denotes the number of tapes used by the machine.

Definition 3.15 (k-tape Turing machine)

A k-tape Turing machine is a tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, where everything is defined as in Definition 3.1 except for the transition function, which is

$$\delta : (Q \setminus \{q_{\text{accept}}, q_{\text{reject}}\}) \times \Gamma^k \to Q \times \Gamma^k \times \{L, R\}^k.$$

The idea allowing us to establish one direction of the equivalence between multitape and single-tape Turing machines is that we can simulate having many tapes T_i by storing all of the contents on a single tape T and separating "tape i" from the other "tapes" using a special symbol. Since our single tape has only one input head, we will also simulate the position of each input head on "tape i" using a special marker on the corresponding tape symbol to act as a virtual input head.

Theorem 3.16

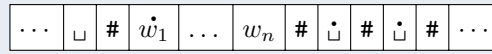
Given a k-tape Turing machine $\mathcal{M}$, we can construct a single-tape Turing machine $\mathcal{M}'$ such that $L(\mathcal{M}') = L(\mathcal{M})$.

Proof. Suppose the given Turing machine $\mathcal{M}$ has k tapes, and the

tape alphabet is denoted by Γ. Our single-tape Turing machine $\mathcal{M}'$ will simulate $\mathcal{M}$'s computation on an input word $w = w_1 \dots w_n$ in the following way:

1. Take the tape alphabet of $\mathcal{M}'$ to be $\Gamma' = \Gamma \cup \dot{\Gamma} \cup \{\#\}$, where $\dot{\Gamma}$ consists of all alphabet symbols of Γ augmented with a dot and # is a special boundary marker.

2. Write the boundary marker # and the symbols of w to the tape of $\mathcal{M}'$, including the dotted symbol $\dot{w}_1$. Then, write $k + 1$ copies of # each separated by a dotted blank space.



3. For each step of the computation, $\mathcal{M}'$ scans its entire tape from the first occurrence of # to the $(k+1)$st occurrence of # to read the symbols on all k tapes of $\mathcal{M}$. Then, $\mathcal{M}'$ makes a second pass along its tape to update any symbols that were changed by the transition function of $\mathcal{M}$. This update includes changing occurrences of dotted symbols in accordance with the changed positions of each virtual input head.

   If any of the virtual input heads of $\mathcal{M}'$ move onto an occurrence of #, then $\mathcal{M}$ must have moved the corresponding input head of that tape onto a blank space. In this case, $\mathcal{M}'$ writes a blank space to this cell of its tape and shifts the symbols of all subsequent cells rightward by one position. ■

The process of condensing the contents of all k tapes onto a single tape is illustrated in Figure 3.3. Naturally, a k-tape Turing machine can simulate the computation of a single-tape Turing machine by using only one of its k tapes. This again gives us the other direction of the relationship between the models, and again establishes the equivalence of the models.

3.2.3. One-Way Infinite Tape Turing Machines

In our definition of a Turing machine, we assumed that the tape used by the machine is *two-way infinite*; that is, there is an infinite number of cells to the left and to the right of the input head, and the input head can therefore move to an infinite number of positions of the tape in either direction.

We didn't have to define our storage in this way, though. We could have alternatively defined the tape to act more like the stack of a pushdown automaton: just like the stack has a fixed bottom boundary forcing us to

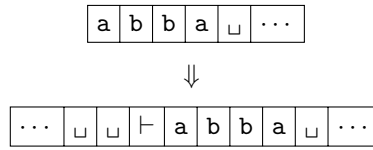


Figure 3.4. Simulating the computation of a one-way infinite tape Turing machine (top) with a two-way infinite tape Turing machine (bottom).

push symbols only above that boundary, the tape could have a fixed left boundary forcing us to write symbols only to the right of that boundary. We call such a tape *one-way infinite*, since at any position along the tape, the input head has a finite number of cells to its left and an infinite number of cells to its right.

Because of the way the computation of the Turing machine begins, the initial position of the input head of a Turing machine with a one-way infinite tape will be on the first symbol of the input word, and the left boundary of the tape will be to the input head's immediate left. In this cell, the input head cannot make a leftward move; if the transition function tells the input head to move left, then the input head will simply remain in the same cell by making a rightward move immediately followed by a leftward move.

> *Remark.* Some authors alternatively assert that, if the input head of a one-way infinite tape Turing machine moves beyond the left boundary of the tape, then the machine "crashes" and the computation cannot continue.

If we want to simulate the computation of a one-way infinite tape Turing machine using a two-way infinite tape Turing machine, then the required conversion seems straightforward: we just need to write a special symbol to one cell of the two-way infinite tape to act as the left boundary, and modify the transition function to handle the case where the input head moves onto this left boundary marker.

Theorem 3.17

Given a one-way infinite tape Turing machine $\mathcal{M}$, we can construct a two-way infinite tape Turing machine $\mathcal{M}'$ such that $L(\mathcal{M}') = L(\mathcal{M})$.

Proof. Suppose the given one-way infinite tape Turing machine $\mathcal{M}$ receives as input a word $w = w_1 \dots w_n$ and has a tape alphabet Γ. Our two-way infinite tape Turing machine $\mathcal{M}'$ will simulate $\mathcal{M}$'s computation on w in the following way:

1. Take the tape alphabet of $\mathcal{M}'$ to be $\Gamma' = \Gamma \cup \{\vdash\}$, where $\vdash$ is a

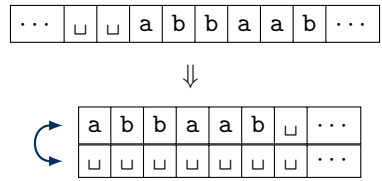


Figure 3.5. Simulating the computation of a two-way infinite tape Turing machine (top) with a one-way infinite tape Turing machine (bottom).

special left boundary marker.

2. Write the symbols of w to the tape of $\mathcal{M}'$, and write the left boundary marker $\vdash$ in the cell to the immediate left of the cell containing w_1.

3. Take the transition function of $\mathcal{M}'$ to be $\delta' = \delta$ and, for each state $q \in Q$, add a new transition $\delta'(q, \vdash) = (q, \vdash, R)$ to the transition function of $\mathcal{M}'$.

4. Create new states q'_{accept} and q'_{a}, and add new transitions to the transition function of $\mathcal{M}'$ as follows:

   - $\delta'(q_{\text{accept}}, \vdash) = (q'_{\text{accept}}, \vdash, R)$;
   - $\delta'(q_{\text{accept}}, c) = (q'_{\text{a}}, c, R)$ for all $c \in \Gamma' \setminus \{\vdash\}$; and
   - $\delta'(q'_{\text{a}}, d) = (q'_{\text{accept}}, d, L)$ for all $d \in \Gamma' \setminus \{\vdash\}$.

   Also, create new states q'_{reject} and q'_{r}, and add similar transitions on these states. Take the accepting and rejecting states of $\mathcal{M}'$ to be q'_{accept} and q'_{reject}, respectively. ■

The construction described in Theorem 3.17 is illustrated in Figure 3.4.

To obtain the other direction of the equivalence—that is, to simulate the computation of a two-way infinite tape Turing machine with a one-way infinite tape Turing machine—we can use our previous result establishing the equivalence of single-tape and multitape Turing machines.

While we won't go through the full proof here, the idea is to split the two-way infinite tape into a pair of one-way infinite tapes, where the "split point" occurs between the first symbol of the input word and the infinite blank spaces to the left of the input word. This action produces two one-way infinite tapes: one containing the input word, and one containing only blank spaces. Then, we modify the transition function of the one-way infinite tape Turing machine to operate on either the first or second tape, switching between the tapes each time the input head of the two-way infinite tape

Turing machine crosses the "split point" on its tape.

Splitting the two-way infinite tape into a pair of one-way infinite tapes gives us the setup depicted in Figure 3.5, where the two-way infinite tape is, in effect, folded in half. Ultimately, this construction completes the proof and establishes the equivalence between one-way and two-way infinite tape Turing machines.

3.3. CLOSURE PROPERTIES ◇

As with the previous chapters, I eventually plan to write a section summarizing the closure properties of various operations applied to the classes of decidable and semidecidable languages. Since Turing machines are so powerful, it's not surprising that nearly all operations are closed for both decidable and semidecidable languages, but it's worth explaining that semidecidable languages happen not to be closed under complement.

3.4. ENCODINGS OF TURING MACHINES ◇

In advance of the section on universal Turing machines, I plan to write a short exposition on arithmetization and ways in which we might encode a description of a Turing machine as a string of symbols $\langle \mathcal{M} \rangle$.

3.5. UNIVERSAL TURING MACHINES

Up to now, we have had to construct different specific Turing machines for each language we wished to recognize. In fact, we have had to construct specific machines for *every* language we wished to recognize in this course, whether that machine be a finite automaton, or a pushdown automaton, or indeed, a Turing machine.

However, we know that Turing machines are capable of performing quite complicated computations, and we know also that we can construct Turing machines that can simulate the computations of other variant Turing machines. What if we took this idea and generalized it as much as possible? That is, what if we constructed some Turing machine that could simulate the computation of *any* other Turing machine?

Alan Turing considered this exact idea in the paper that introduced the model of computation that would eventually be named after him. Turing described the process of constructing a machine $\mathcal{U}$ that is capable of simulating the computation of any other machine $\mathcal{M}$, so long as we give an appropriate encoding of $\mathcal{M}$ as part of the input to $\mathcal{U}$:

> *It is possible to invent a single machine which can be used to compute any computable sequence. If this machine U is supplied with a tape on the beginning of which is written the S.D. [standard description] of some computing machine M, then U will compute the same sequence as M.*
>
> — ALAN TURING

Here, like Turing did before us, we will show how to construct such a machine $\mathcal{U}$, which is nowadays called a *universal Turing machine*.

> *Remark.* It's important to note that, in this context, "universal" does not mean that the Turing machine $\mathcal{U}$ can compute *everything*. It only means that $\mathcal{U}$ can compute whatever other Turing machines can compute.

The main benefit of having such a machine is that we will no longer have to construct specific Turing machines for each language we consider; now, we can just give a high-level description of the Turing machine's computation, and we can feasibly "program" the universal Turing machine to perform its computation in a similar way. These high-level descriptions will be quite similar to what we saw in the proofs of Theorems 3.14, 3.16, and 3.17, where we simply listed the steps of the machine's computation instead of explicitly writing out each component of the machine.

Suppose that the input we give to our universal Turing machine $\mathcal{U}$ is of the form $\langle \mathcal{M}, w \rangle$, where $\mathcal{M}$ is an encoding of the Turing machine we wish to simulate and w is the input word given to $\mathcal{M}$. Given an input of this form, our machine $\mathcal{U}$ must satisfy three criteria:

1. $\mathcal{U}$ halts its computation on input $\langle \mathcal{M}, w \rangle$ if and only if $\mathcal{M}$ halts its computation on input w;
2. $\mathcal{U}$ enters its accepting state if and only if $\mathcal{M}$ enters its accepting state; and
3. $\mathcal{U}$ enters its rejecting state if and only if $\mathcal{M}$ enters its rejecting state.

We now go through with the construction of this machine $\mathcal{U}$.

Theorem 3.18

There exists a universal Turing machine $\mathcal{U}$ that, given an input $\langle \mathcal{M}, w \rangle$, is capable of simulating the computation of a Turing machine $\mathcal{M}$ on an input word w.

Proof. We will construct $\mathcal{U}$ in the form of a multitape Turing machine,

just as we did in our procedure to determinise a nondeterministic Turing machine.

For this construction, we need only three tapes:

- The first tape will initially contain the input $\langle \mathcal{M}, w \rangle$, and after the computation of $\mathcal{U}$ begins, it will simulate the contents of the tape of $\mathcal{M}$.
- The second tape will contain the encoding of the machine $\mathcal{M}$.
- The third tape will keep track of which state we are currently in during the computation of $\mathcal{M}$ by maintaining the current state as a binary number.

At the beginning of its computation, $\mathcal{U}$ will contain only the input $\langle \mathcal{M}, w \rangle$ on its first tape, and its other two tapes will be blank.

$\cdots$	$\sqcup$	$\langle$	...	$\mathcal{M}$	...	,	...	w	...	$\rangle$	$\sqcup$	$\cdots$
$\cdots$	$\sqcup$	$\sqcup$	...	$\sqcup$	...	$\sqcup$	...	$\sqcup$	...	$\sqcup$	$\sqcup$	$\cdots$
$\cdots$	$\sqcup$	$\sqcup$	...	$\sqcup$	...	$\sqcup$	...	$\sqcup$	...	$\sqcup$	$\sqcup$	$\cdots$

Now, we can describe how $\mathcal{U}$ simulates the computation of $\mathcal{M}$ on w:

1. Initialize the tapes in the following way:
   (a) Transfer the encoding of $\mathcal{M}$ from the first tape to the second tape by writing $\langle \mathcal{M} \rangle$ to the second tape and erasing it from the first tape.
   (b) Read the encoding of $\mathcal{M}$ on the second tape to determine the number of states in $\mathcal{M}$. If $\mathcal{M}$ contains k states, then write $\lceil \log_2(k) \rceil$ copies of `0` to the third tape.
   (Remember, the current state is being maintained as a binary number, so we need a logarithmic amount of bits to represent a given state's number k.)

After initialization, the tape will look like the following:

$\cdots$	$\sqcup$	$\sqcup$	...	$\sqcup$	...	$\langle$	...	w	...	$\rangle$	$\sqcup$	$\cdots$
$\cdots$	$\sqcup$	$\langle$	...	$\mathcal{M}$	...	$\rangle$	...	$\sqcup$	...	$\sqcup$	$\sqcup$	$\cdots$
$\cdots$	$\sqcup$	0	...	0	...	0	...	$\sqcup$	...	$\sqcup$	$\sqcup$	$\cdots$

2. Move the input head of the first tape to the first symbol of w. Move the input head of the second tape to the first symbol of $\langle \mathcal{M} \rangle$. Move the input head of the third tape to the first symbol of the sequence of 0s.

3. Repeat the following steps until $\mathcal{M}$ halts:

   (a) For each computation step of $\mathcal{M}$, scan the second tape to find a transition that matches the current input symbol and state written on the first and third tapes, respectively.

   (b) Modify the contents of the first and third tapes to reflect the transition that was just taken.

   (c) If no transition exists for the current state/symbol pair, then halt and go to step 4.

4. Transition to q_{accept} if $\mathcal{M}$ transitioned to its accepting state. Transition to q_{reject} if $\mathcal{M}$ transitioned to its rejecting state. ■

Let's now verify that this construction does, in fact, satisfy the three criteria we laid out earlier. First, we require that $\mathcal{U}$ halts its computation on input $\langle \mathcal{M}, w \rangle$ if and only if $\mathcal{M}$ halts its computation on input w. Since $\mathcal{U}$ uses the description of $\mathcal{M}$ to see what it would do after reading each symbol of w, $\mathcal{U}$ behaves in a manner identical to $\mathcal{M}$, and so $\mathcal{U}$ halts on its input if and only if $\mathcal{M}$ also halts on its input. Second, we require that $\mathcal{U}$ accepts if and only if $\mathcal{M}$ accepts, and we see immediately that this happens by Step 4 of the simulation procedure. In fact, $\mathcal{U}$ can't reach its accepting state unless $\mathcal{M}$ has done the same. Third, we require the same behaviour for rejecting inputs, and the argument in this case is nearly identical to the one we had for accepting inputs. Therefore, the universal Turing machine $\mathcal{U}$ satisfies all of our criteria and performs a correct simulation of any other Turing machine we provide as input!

3.6. THE CHURCH–TURING THESIS

Long ago, before digital computers as we know them existed and even before the phrase "computer science" entered the lexicon of humanity, mathematicians and logicians wanted to know whether it was possible to use mechanical procedures to solve mathematical problems. The desire for such procedures dates back to the 17th century and the time of Gottfried Wilhelm Leibniz, who dreamt of constructing a machine he called a *calculus ratiocinator* to automate the task of performing general mathematical calculations. (Although Leibniz is well-known for his work on *the* calculus of infinitesimals that one often learns in school, the word "calculus" in this

sense refers more generally to a system for performing calculations.)

Leibniz's ratiocinator was to use a formal language he called *characteristica universalis*—Latin for "universal character"—which he intended to be a general framework for expressing mathematical concepts in symbols. Indeed, the characteristica universalis might be considered the first programming language! Although Leibniz never succeeded in constructing his ratiocinator, his dream formed the precursor for much of the work done in the formalization of mathematics throughout the 19th and 20th centuries.

At the turn between these two centuries, David Hilbert delivered a presentation to the International Congress of Mathematicians [1900; 1901] wherein he outlined 10 (and later, an additional 13) unsolved problems that would come to guide the new century's study of mathematics. Of Hilbert's 23 problems, three remain unsolved to this day, while two are considered too vague to ever have a proper solution. But one problem in particular is relevant to our current topic: the *second problem*, which asks to prove that the axioms of arithmetic over the real numbers are consistent; that is, incapable of producing logical contradictions.

Much like Leibniz dreamt of a machine to solve any mathematical problem, Hilbert dreamt of a purely logical formal system where, starting from a given set of axioms, one would be able to prove any mathematical statement. Indeed, this dream motivated his second problem: if the axioms of arithmetic are consistent, then we can use pure logical rules to solve any mathematical problem we come across.

> *We hear within us the perpetual call:*
> *There is the problem. Seek its solution.*
> *You can find it by pure reason, for*
> *in mathematics there is no ignorabimus.*
>
> — DAVID HILBERT

In the following decades, Hilbert devoted considerable time to trying to establish a positive answer for his second problem. Unfortunately for him, Kurt Gödel would show via his *incompleteness theorems* [1931] not just that a proof of consistency for even a simpler system like the Peano arithmetic over the natural numbers is impossible to attain within the system itself, but also that such arithmetic systems *must* contain statements that can be neither proved nor disproved. In resolving the second problem in the negative, Gödel dealt a crushing blow to Hilbert's dream: in mathematics, it turns out there *is* some ignorabimus. Despite this substantial setback, Hilbert and others persisted in looking forward, and researchers continued pursuing modified and constrained forms of Hilbert's dream—even if we can't formalize all of mathematics, we can at least formalize some of mathematics.

A few years before Gödel announced his groundbreaking results, Hilbert—together with his doctoral student Wilhelm Ackermann—posed a more concrete question in their book, *Grundzüge der theoretischen Logik* [1928].

Their question, closer in spirit to Leibniz's dream, asked for

> [...] *a method which permits us to decide for any given formula in which domains of individuals it is universally valid (or satisfiable) and in which it is not.*
>
> — DAVID HILBERT AND WILHELM ACKERMANN

In other words, Hilbert and Ackermann wanted to know whether there exists a general procedure that takes a predicate logic formula and gives a "yes" or "no" answer as to whether that formula is true, no matter which predicates are used or which values are assigned to variables within the formula. Their question would come to be known as the *Entscheidungsproblem*, which is German for "decision problem". Although Gödel's work had brought down Hilbert's second problem, researchers at the time noted that, strictly speaking, the Entscheidungsproblem had not suffered the same fate. There remained, at least for the time being, a glimmer of hope.

The issue, however, was that there was not yet a universally agreed-upon definition of a "procedure" that could decide such a thing as the Entscheidungsproblem. The most appropriate definition was eventually taken to be that of an *effective method*. If we're given a class of problems, then a method for that class of problems is called effective if

1. the method consists of a finite number of exact instructions; and
2. the method always terminates and produces a correct answer when it is applied to a problem from its class.

In principle, an effective method is one that a human can perform on paper in a purely mechanical manner; it requires no creative thought or insight to arrive at an answer. It is computation in its purest form. If we view the class of problems in a way that allows us to map individual inputs to "yes" and "no" outputs, then we obtain a function for that class, and we say that such a function is *effectively calculable*.

> *Remark.* Compare our characterization of an effectively calculable function to the properties of a computable function given by Enderton [1977] in Section 3.1.3. Criteria 1 and 2 match almost exactly!

But what specific properties does an effectively calculable function possess? In a lecture given in 1934, whose notes were published decades later by Davis [1965], Gödel proposed that the notion of effective calculability could be captured by the so-called *general recursive* functions. In doing so, he provided at least an initial characterization of what it means for a function to be effectively calculable. However, this was to be only the first step in a

series of consequential results pertaining to the Entscheidungsproblem.

The first major breakthrough for the Entscheidungsproblem came in a series of papers by Alonzo Church, who framed the idea of effective calculability in terms of his *lambda calculus*. The lambda calculus is a logical system that allows us to express computations in terms of functions and their applications. As it turns out, the class of effectively calculable functions corresponds to the class of functions that are expressible in the lambda calculus; this was formally established by both Church [1936b] and Kleene [1936a,b]. Church ultimately proved that there does not exist any procedure to decide whether a given formula has an equivalent particular normal form in the lambda calculus [1936a; 1936b]. From this observation, Church struck at Hilbert's dream once again and concluded the following:

> *The general case of the Entscheidungsproblem of the engere Funktionenkalkül is unsolvable.*
> — ALONZO CHURCH

The second breakthrough came with a presentation by Alan Turing to the London Mathematical Society [1936]. Like Church, Turing showed:

> [...] *the Hilbertian Entscheidungsproblem can have no solution.*
> — ALAN TURING

However, Turing relied on a different formalization: a machine model, which later came to be known as our familiar *Turing machine*. The crux of Turing's argument was that the Entscheidungsproblem could be reformulated, or *reduced*, to a problem pertaining to a property of his machine, and one could then show that this problem is also unsolvable. Turing was aware of Church's work—indeed, he raced to deliver his presentation shortly after learning about Church's work in that same year—and Turing added as an appendix to his published paper a proof sketch showing that his machine formalization was equivalent to Church's lambda calculus, and therefore to the class of effectively calculable functions. Turing would go on to earn his doctorate under the supervision of Church just a couple of years later.

Despite this flurry of results and the fall of the Entscheidungsproblem, there remained a question: Gödel had the general recursive functions, Church had the lambda calculus, and Turing had his machines, but which of these formulations is best to use when we refer to effective calculability? This question was not truly settled until the following decade, when Stephen Kleene made the claim that *any* of these formulations is suitable. Kleene [1943] began by introducing what he calls *Church's thesis*:

> *Every effectively calculable function (effectively decidable predicate) is general recursive.*
> — STEPHEN KLEENE

Church's thesis connects the class of general recursive functions to the lambda calculus by stating, in our terminology, that any problem for which there exists a procedure that returns an answer on each input belonging to the problem's language is semidecidable.

Kleene later introduced in his book *Introduction to Metamathematics* [1952] a companion statement, which he calls *Turing's thesis*:

> [...] *that every function which would naturally be regarded as computable under [Turing's] definition, i.e. by one of his machines, is equivalent to Church's thesis* [...]
>
> — STEPHEN KLEENE

Turing's thesis is Kleene's encapsulation of what Turing himself expressed in the appendix of his paper: that his machine formalization is equivalent to the lambda calculus formalization given by Church. Consequently, anything that a Turing machine can do is effectively calculable, and therefore it is semidecidable.

Taken together, these two statements give us the *Church–Turing thesis*: the unifier between effective methods and Turing machines. In modern language, we can express the thesis as follows.

Church–Turing thesis

Any function that can be computed by an algorithm can also be computed on a Turing machine.

Note that we refer to this result as a "thesis" and not as a "theorem", since it is more definitional rather than a statement that we can formally prove.

Turing-Completeness. In recent times, the Church–Turing thesis has allowed researchers to prove that all sorts of formal models are capable of behaving like a machine running an algorithm. If some model of computation or some system of rules can be used in a way that allows it to simulate the computation of any Turing machine, then we say that model or system is *Turing-complete*.

We've already seen one example of something that is Turing-complete—the universal Turing machine. But since that is itself a kind of Turing machine, we shouldn't be too surprised. Instead, there are many more (and much weirder) examples of Turing-complete things in our daily lives:

- Most general-purpose programming languages, and some specialized languages (like LaTeX, the typesetting system used to create this book!)

- Microsoft Excel and Microsoft PowerPoint
- Conway's Game of Life and other cellular automata
- Enzyme-based DNA computers
- The cells of the human heart
- The Dwarf Fortress, Minecraft, and Minesweeper video games
- The Magic: The Gathering card game
- The x86 assembler instruction `mov`, by itself

You might now reasonably wonder whether the computers we use every day are Turing-complete. Well, the answer—strictly speaking—is no! This comes down to one simple reason: nobody has figured out how to equip a real-world computer with an infinite amount of memory. Thus, when we speak about something being Turing-complete, we often set aside the limitation of finite memory and focus on the computational power of the thing itself.

3.7. THE CHOMSKY HIERARCHY

I plan to expand this short section about the Chomsky hierarchy to tie together all that we have learned about language classes and models of computation.

The Chomsky hierarchy was initially proposed by Noam Chomsky [1956; 1959a] and further refined in Chomsky's joint work with the mathematician Marcel-Paul Schützenberger [1963]. There are two major differences between the hierarchy we developed and Chomsky's hierarchy: in our hierarchy, we included the class of finite languages as a subset of the regular languages, while Chomsky's hierarchy includes the class of *context-sensitive languages* that we briefly discussed, but otherwise left aside.

CHAPTER NOTES

3.1. The model of computation that bears Alan Turing's name was first presented in his groundbreaking 1936 paper, although in this paper Turing referred to his model as an *a-machine* (or *automatic machine*). It wasn't until the following year that the name "Turing machine" was bestowed on the model by Church [1937].

Table 3.1
THE CHOMSKY HIERARCHY

Language class	Model of computation	Grammar
Recursively enumerable	Turing machines	Type 0
Context-sensitive	Linear-bounded automata	Type 1
Context-free	Pushdown automata	Type 2
Regular	Finite automata	Type 3

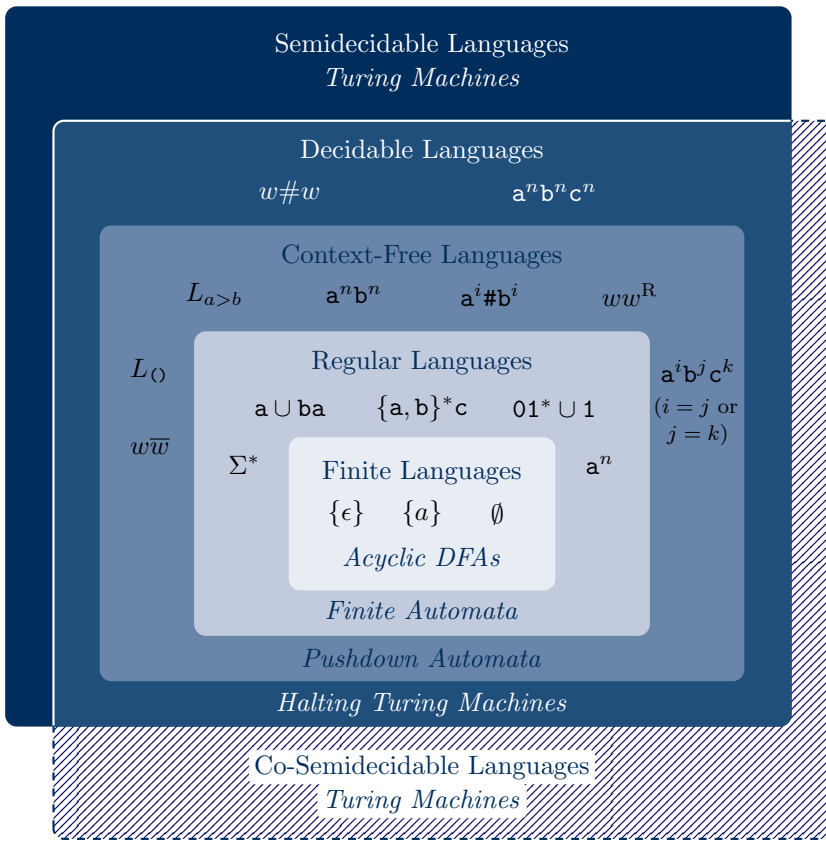


Figure 3.6. Our hierarchy of language classes and models of computation, based on the Chomsky hierarchy.

The book by Petzold [2008] breaks down Turing's 1936 paper sentence by sentence, providing a remarkably clear and detailed explanation of the techniques Turing used along with a wealth of background information and pointers to further readings, while another book edited by Copeland [2004] collects a number of Turing's influential writings from across his career.

For those seeking a biography of Alan Turing, the standard reference is the book by Hodges [1983], which also formed the basis for the 2014 film *The Imitation Game*. Other biographies have been written by Turing's mother, Sara [1959], and his nephew, Dermot [2015].

We indicated that the class of decidable languages was once known as the class of "recursive languages". While this terminology was commonplace in the early days of computer science, the word "recursive" is now strongly associated with the idea of recursion in algorithm design. Soare [1996] gives a historical accounting of the development of this terminology and makes an argument for why the word "recursive" should no longer be used to mean "decidable". Despite this, some authors continue to refer to the class of "recursive languages" to this day!

3.2. Nondeterministic Turing machines were introduced alongside the deterministic model in Turing's 1936 paper, though Turing called his nondeterministic model a *c-machine* (or *choice machine*) and only referred to the machine in an offhand remark.

The study of multitape Turing machines seems to have originated with the work of Minsky [1961], who used a special kind of 2-tape Turing machine to show that Post's decision problem for tag systems [1943] is undecidable. Minsky's work was inspired by that of Rabin and Scott [1959], who studied problems relating to finite automata receiving two tapes as input. Supposing that we have a k-tape Turing machine that halts on its input word in T computation steps, Hartmanis and Stearns [1965] showed that we may construct a 1-tape Turing machine that can simulate the original computation in at most T^2 computation steps, while Hennie and Stearns [1966] showed that a 2-tape Turing machine can simulate the computation in at most $T \log(T)$ computation steps.

One-way infinite tape Turing machines, believe it or not, also first appeared in Turing's 1936 paper; Turing described an example computation in which the first three symbols on the tape are fixed to be "əə0", and where the input head does not move leftward past the two ə symbols.

3.3. *Chapter notes will be added when this section is written.*

3.4. *Chapter notes will be added when this section is written.*

3.5. As we noted, the idea of the universal Turing machine was put forth in Turing's 1936 paper. Although our construction here uses three tapes, we could instead use two tapes by the result of Hennie and Stearns [1966] or one tape by the result of Hartmanis and Stearns [1965], each with a commensurate impact on the time required to complete our computation.

Many researchers have considered the question of how small a universal Turing machine can be while still retaining its universality property. (Note that, in measuring size, we just count the number of states plainly and assume the universal Turing machine does not require distinguished accepting or rejecting states.) Given a universal Turing machine with n states and m alphabet symbols, Shannon [1956] proved that it is possible to construct an equivalent universal Turing machine having only two states and at most $4mn + m$ alphabet symbols, or alternatively one having only two alphabet symbols and $(2^\ell - 1)n$ states, where ℓ is the smallest integer such that $m \leq 2^\ell$. Shannon also established in the same paper that it is impossible to construct a one-state universal Turing machine. Using the notation (n, m) to denote an n-state, m-symbol universal Turing machine, Minsky [1962] constructed a $(7, 4)$ machine while Rogozhin [1996] constructed seven small universal Turing machines, all the way down to a $(2, 18)$ machine. Wolfram [2002, chapter 11, section 12] described a $(2, 5)$ machine that is similar but, strictly speaking, not directly comparable to our universal Turing machine model, and he further conjectured the universality of a $(2, 3)$ machine. Five years later, Wolfram's conjecture was verified by Alex Smith, who was at the time an undergraduate student; Smith received a \$25 000 prize and, after protracted discussions about the correctness of his proof, published his results [2020]. Woods and Neary [2009] give a far more detailed account of the history of small universal Turing machines in their survey article.

3.6. Much has been written about Leibniz and his work in mathematics and logic; see, for example, the papers of Jourdain [1916] and Lenzen [2018] as well as the book by Davis [2000, chapter 1].

More information about Hilbert's dream and the quest to solve his second problem—more properly called *Hilbert's program*—is available in the survey by Zach [2007], who gives both historical and philosophical perspectives on Hilbert's early work in the 1890s, through to the consequences of Gödel's incompleteness theorems in the 1930s and 1940s, and leading up to modern results.

For an incredibly detailed discussion of the origins of computability theory and the development of the Church–Turing thesis, see the article by Soare [1996].

Although our focus in this section was on the work of Church and Turing, other contributions from this era should not be overlooked. Post [1936] proposed a model of computation that he called "Formulation 1", and his model behaves in a manner basically equivalent to that of a Turing machine. Remarkably, Post completed his work without having any knowledge of Turing's paper, which appeared in the same year. However, Post was aware of the work of both Gödel and Church, indicating that he expected his model to be logically equivalent to their formulation of effective calculability via general recursive functions.

Many unconventional models of computation have been shown to be Turing-complete. Microsoft Excel was studied by Gordon and Peyton Jones [2021] and Microsoft PowerPoint was studied by Wildenhain [2017]. Berlekamp, Conway, and Guy investigated the Game of Life in their well-known book [2004, chapter 25]. Cook [2004] established the computational universality of elementary cellular automata. Shapiro [2012] and Scarle [2009] put a biological spin on computation with their studies on enzyme-based DNA computers and human heart cells, respectively. The Turing-completeness of the Dwarf Fortress video game was established via a game map constructed by a player known as Jong89 [2009]; Minecraft was suggested to be Turing-complete in a Bytejacker interview with the game's creator, Markus "Notch" Persson [2010]; and Kaye [2000] showed that an infinite variant of Minesweeper possessed the property of Turing-completeness. Churchill, Biderman, and Herrick [2021] concluded a nearly decade-long study of the Magic: The Gathering card game. Lastly, Dolan [2013] showed that the `mov` instruction on its own is sufficient to compute anything a Turing machine can compute.

3.7. Noam Chomsky is perhaps best known for his work in linguistics, and his hierarchy was originally intended to act as a mathematical model for natural language. Although Chomsky's first work on formalizing natural language using mathematics appeared in 1956, the hierarchy proposed in his first paper does not correspond to the Chomsky hierarchy as we know it today and instead had more of a linguistic focus. Chomsky would later fix mathematical definitions in a technical report [1958] before publishing his full paper [1959a], which proposes a hierarchy that more closely resembles the one we have seen.

In the intervening decades, criticisms have been levelled against the Chomsky hierarchy, for example, from those claiming that the language classes are too coarse. In fact, this is something Chomsky himself recognized, remarking that the class of regular languages is too restrictive to contain a grammar for English [1956] while the

class of recursively enumerable languages is so broad that it is of no interest [1959a]. For further discussion of criticisms from a linguistic perspective, see the book chapter by Seuren [2013].

Some researchers have proposed refinements of the Chomsky hierarchy that include "in-between" classes such as the subregular languages or the mildly context-sensitive languages; see, for instance, the work of Jäger and Rogers [2012].

CHAPTER FOUR

DECISION PROBLEMS

A FUNDAMENTAL ASPECT of studying various models of computation is determining precisely what kinds of questions about that model can be answered algorithmically. For example, does there exist an algorithm that can tell us whether or not a deterministic finite automaton accepts a given input word? It seems obvious—after all, such an algorithm only needs to figure out whether the finite automaton, given the input word, would end up in an accepting state or not—but it's still something that we need to establish formally.

In the previous chapter, we introduced the notions of decidable and semidecidable languages. Using much of the same terminology we developed for reasoning about languages, we can talk more generally not just about languages but about *problems* pertaining to our various models of computation, and we can formally prove which problems can be decided in much the same way as languages can be decided.

A *decision problem* is a problem for which each input instance corresponds to either a "yes" or a "no" answer. At its core, a decision problem is a language, and the elements of the language are input instances that correspond to a "yes" answer. Each element of the language is an encoding of whatever model of computation we're considering, in some cases with an input word given to that model: in the case of regular languages, we use encodings of finite automata, and so on.

If there exists a *decision algorithm* that produces the correct "yes" or "no" answer for any instance of a given decision problem, and the algorithm halts on all inputs, then we say that the decision problem is *decidable*. If no such decision algorithm exists, then we say that the decision problem is *undecidable*.

> *Remark.* Note that "undecidable" doesn't mean the decision problem is impossible; "undecidable" only means "not decidable", as in "no decision algorithm exists that *always* halts and gives a yes/no answer".

There are a number of common decision problems that we can ask about a model of computation $\mathsf{X} \in \{\mathsf{RE}, \mathsf{DFA}, \mathsf{NFA}, \epsilon\text{-}\mathsf{NFA}, \mathsf{CFG}, \mathsf{PDA}, \mathsf{TM}, \dots\}$, and

Table 4.1
COMMON DECISION PROBLEMS AND THEIR DEFINITIONS

Membership	$A_{\mathsf{X}} = \{\langle \mathcal{B}, w\rangle \mid \mathcal{B}$ is an X that accepts input word $w\}$
Emptiness	$E_{\mathsf{X}} = \{\langle \mathcal{B}\rangle \mid \mathcal{B}$ is an X and $L(\mathcal{B}) = \emptyset\}$
Universality	$U_{\mathsf{X}} = \{\langle \mathcal{B}\rangle \mid \mathcal{B}$ is an X and $L(\mathcal{B}) = \Sigma^*\}$
Equivalence	$EQ_{\mathsf{X}} = \{\langle \mathcal{B}, \mathcal{C}\rangle \mid \mathcal{B}$ and $\mathcal{C}$ are both X and $L(\mathcal{B}) = L(\mathcal{C})\}$
Inclusion	$IN_{\mathsf{X}} = \{\langle \mathcal{B}, \mathcal{C}\rangle \mid \mathcal{B}$ and $\mathcal{C}$ are both X and $L(\mathcal{B}) \subseteq L(\mathcal{C})\}$

these problems are summarized in Table 4.1. In this chapter, we will get our first taste of "programming" Turing machines to solve each of these decision problems. Note that we no longer need to construct a specific Turing machine for a particular decision problem, because (as we now know) the Church–Turing thesis tells us that any function that can be computed by an algorithm can also be computed by a Turing machine.

The study of computation doesn't end with decision problems. As Table 4.2 illustrates, a number of types of computational problems exist depending on what specifically we would like to know about a problem. Decision problems are the most common formulation, but I will eventually write a few paragraphs about these other problem types, as they will appear in later chapters.

4.1. DECIDABLE PROBLEMS FOR REGULAR LANGUAGES

We begin by considering our common decision problems applied to models of computation that recognize the class of regular languages. Regular languages (and the associated models that recognize such languages) are very useful for practical applications since, as we will see, each of the common decision problems are decidable for this class. The downside, of course, is that the class of regular languages is much smaller than the other language classes we know, which in turn limits our expressive power.

Membership Problem

The membership problem is a fundamental decision problem for any model of computation, since so much that pertains to decidability boils down to simply being able to figure out whether some model accepts some input word. Here, we will obtain our first decidability result by showing that there exists an algorithm that allows us to determine whether or not some deterministic finite automaton $\mathcal{B}$ accepts its input word w.

The technique we will apply in the proof of this result (and others) involves the use of a Turing machine to simulate the computation of the

Table 4.2
TYPES OF COMPUTATIONAL PROBLEMS

Type of problem	**Description**	**Typical formulation**	**Example**
Decision problem	Answer to every instance is either "yes" or "no"	$f\colon \Sigma^* \to \{0,1\}$ $L = \{x \in \Sigma^* \mid f(x) = 1\}$	Primality testing: $2 \to$ "yes, prime"
Function problem	Answer to every instance exists, but is not necessarily binary (i.e., "yes" or "no")	$f\colon \Sigma^* \to \Sigma^*$ $L = \{(x, f(x))\}$	Prime factorization: $2021 \to \{(2021, 43 \times 47)\}$
Search problem	Answer is a relation of instance-solution pairs	$R \subseteq \Sigma^* \times \Sigma^*$ $A_x =$ set of solutions for $x \in \Sigma^*$ $L = \{(x,y) \mid (x,y) \in R,\ y \in A_x\}$	Nontrivial factors: $6 \to \{(6,2),(6,3)\}$
Counting problem	Answer is the number of solutions to a given search problem	$R \subseteq \Sigma^* \times \Sigma^*$ $A_x =$ set of solutions for $x \in \Sigma^*$ $L = \lvert\{(x,y) \mid (x,y) \in R,\ y \in A_x\}\rvert$	# of nontrivial factors: $16 \to 3$
Optimization problem	Answer is the "best possible" solution among a set of all feasible solutions	Find $x \in \Sigma^*$ such that $f(x)$ is minimal/maximal	Largest prime factor: $93 \to 31$

finite automaton. Then, whether the finite automaton accepts or doesn't accept the input word, the Turing machine returns the same result.

Theorem 4.1

A_{DFA} is decidable.

Proof. Construct a Turing machine $\mathcal{M}_{\text{ADFA}}$ that takes as input $\langle \mathcal{B}, w \rangle$, where $\mathcal{B}$ is a deterministic finite automaton and w is the input word to $\mathcal{B}$, and performs the following steps:

$\mathcal{M}_{\text{ADFA}}$

1. Simulate $\mathcal{B}$ on input w.
2. If the simulation ends in an accepting state of $\mathcal{B}$, then accept. Otherwise, reject.

This Turing machine accepts its input $\langle \mathcal{B}, w \rangle$ if and only if the deterministic finite automaton $\mathcal{B}$ accepts its input w, and the Turing machine rejects if and only if $\mathcal{B}$ rejects. Since the length of w, and therefore the simulation of $\mathcal{B}$, is finite, the Turing machine will never become trapped in an infinite computation. Thus, $\mathcal{M}_{\text{ADFA}}$ decides the membership problem for deterministic finite automata. ■

Since we know also that we can convert from nondeterministic finite automata to deterministic finite automata, and from regular expressions to deterministic finite automata, we get similar positive decidability results for these models.

Corollary 4.2

A_{NFA} and A_{RE} are decidable.

Proof. Given a nondeterministic finite automaton $\mathcal{B}$, we can convert it to an equivalent deterministic finite automaton $\mathcal{B}'$ and run $\mathcal{M}_{\text{ADFA}}$ from the proof of Theorem 4.1 on the input $\langle \mathcal{B}', w \rangle$.

Likewise, given a regular expression R, we can convert it to an equivalent deterministic finite automaton $\mathcal{S}$ and run $\mathcal{M}_{\text{ADFA}}$ on the input $\langle \mathcal{S}, w \rangle$. ■

Indeed, one consequence of Kleene's theorem is that a decision problem being decidable for the class DFA implies that it is also decidable for the classes NFA and RE (as well as the class ϵ-NFA, though this arguably already falls under the "nondeterministic" umbrella). Thus, going forward in this section, we will only focus on full proofs for the class DFA.

Emptiness Problem

Let's now turn to the emptiness problem. What does it mean for the language of a finite automaton to be empty? Obviously, it means the finite automaton doesn't accept any input words, but under what condition is this the case? Since any accepting computation of a finite automaton must conclude in a final state, we can reason that a finite automaton is incapable of accepting any input words only if there exists no path from its initial state to a final state.

We will use this reasoning in the algorithm to decide the emptiness problem for deterministic finite automata. Since every finite automaton has a finite set of states, we can traverse the transitions of the finite automaton starting from the initial state and mark each state as we encounter it. If, by the end of this traversal, we never mark a final state, then this implies there exists no path from the initial state to any final state.

Theorem 4.3

E_{DFA} is decidable.

Proof. Construct a Turing machine $\mathcal{M}_{\text{EDFA}}$ that takes as input $\langle \mathcal{B} \rangle$, where $\mathcal{B}$ is a deterministic finite automaton, and performs the following steps:

$\mathcal{M}_{\text{EDFA}}$

1. Mark the initial state of $\mathcal{B}$.
2. Mark all states that have an incoming transition from any previously marked state. Repeat until no new states are marked.
3. If no final state of $\mathcal{B}$ is marked, then accept. Otherwise, reject.

This Turing machine accepts its input $\langle \mathcal{B} \rangle$ if and only if the deterministic finite automaton $\mathcal{B}$ has no possible computation path leading from its initial state to a final state, and the Turing machine rejects if and only if an accepting computation path exists in $\mathcal{B}$. Since the number of states of $\mathcal{B}$ that could be marked is finite, the Turing machine will never become trapped in an infinite computation. Thus, $\mathcal{M}_{\text{EDFA}}$ decides the emptiness problem for deterministic finite automata. ■

Naturally, following the same lines of reasoning we developed for the membership problem, we can establish the following corollary.

Corollary 4.4

E_{NFA} and E_{RE} are decidable.

Universality Problem

The universality problem goes hand in hand with the emptiness problem we just studied. Indeed, the two problems are complementary: if $L = \Sigma^*$, then $\overline{L} = \emptyset$, where $\overline{L}$ denotes the complement of the language L as we saw before. Intuitively, this makes sense: if a finite automaton accepts *every* input word, then the complement of that finite automaton must accept *no* input words.

Because of this complementary relationship, all we need to do in order to decide the universality problem is complement the language of our given deterministic finite automaton. Then, we can simply reuse the Turing machine $\mathcal{M}_{\text{EDFA}}$ and the algorithm that we developed to decide the emptiness problem on the complemented finite automaton. In other words, in order to decide whether $L(\mathcal{B}) = \Sigma^*$, we simply need to decide whether $\overline{L(\mathcal{B})} = \emptyset$.

Theorem 4.5

U_{DFA} is decidable.

Proof. Construct a Turing machine $\mathcal{M}_{\text{UDFA}}$ that takes as input $\langle \mathcal{B} \rangle$, where $\mathcal{B}$ is a deterministic finite automaton, and performs the following steps:

$\mathcal{M}_{\text{UDFA}}$

1. Convert $\mathcal{B}$ to a deterministic finite automaton $\mathcal{B}'$ recognizing the language $\overline{L(\mathcal{B})}$ using the construction from the proof of Theorem 1.30.
2. Run $\mathcal{M}_{\text{EDFA}}$ from the proof of Theorem 4.3 on input $\langle \mathcal{B}' \rangle$.
3. If $\mathcal{M}_{\text{EDFA}}$ accepts, then accept. Otherwise, reject.

This Turing machine accepts its input $\langle \mathcal{B} \rangle$ if and only if $\mathcal{M}_{\text{EDFA}}$ accepts its input $\langle \mathcal{B}' \rangle$, where $\mathcal{B}'$ is the complement of the deterministic finite automaton $\mathcal{B}$. This occurs only when the language of $\mathcal{B}'$ is empty, and thus $\mathcal{B}$ must accept all input words in this case. Likewise, the Turing machine rejects if and only if $\mathcal{M}_{\text{EDFA}}$ rejects, which means that the language of $\mathcal{B}'$ is not empty and there exists at least one input word not accepted by $\mathcal{B}$. Since both the conversion process to obtain $\mathcal{B}'$ as well as the process of running $\mathcal{M}_{\text{EDFA}}$ are finite, the Turing machine will never become trapped in an infinite computation. Thus, $\mathcal{M}_{\text{UDFA}}$

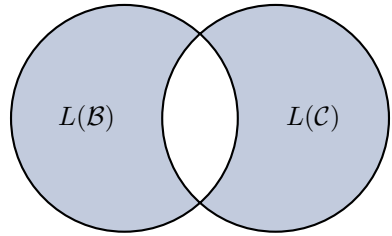


Figure 4.1. The symmetric difference of two languages $L(\mathcal{B})$ and $L(\mathcal{C})$.

decides the universality problem for deterministic finite automata. ■

Additionally, the following corollary holds as we would expect.

Corollary 4.6

U_{NFA} and U_{RE} are decidable.

Equivalence Problem

Given two deterministic finite automata $\mathcal{B}$ and $\mathcal{C}$, how can we test whether $L(\mathcal{B}) = L(\mathcal{C})$? In theory, we could give all the words in $L(\mathcal{B})$ as input to $\mathcal{C}$ and test their membership using our Turing machine $\mathcal{M}_{\text{ADFA}}$, and vice versa. However, this won't work out very well if either of $L(\mathcal{B})$ or $L(\mathcal{C})$ is infinite.

Instead, we can make the following useful observation: if $L(\mathcal{B}) = L(\mathcal{C})$, then every word in $L(\mathcal{B})$ and every word in $L(\mathcal{C})$ must appear in the intersection of the two languages. Equivalence means that no word belongs only to one of the two languages, and so in order to test whether two languages are equivalent, we need only test whether the non-intersecting parts of each language are empty: another application for our emptiness-testing Turing machine $\mathcal{M}_{\text{EDFA}}$!

The "non-intersecting part" of two languages is more properly referred to as the *symmetric difference* of the languages. Given two languages $L(\mathcal{B})$ and $L(\mathcal{C})$, their symmetric difference is the language

$$L(\mathcal{B}) \triangle L(\mathcal{C}) = \left(L(\mathcal{B}) \cap \overline{L(\mathcal{C})}\right) \cup \left(\overline{L(\mathcal{B})} \cap L(\mathcal{C})\right).$$

The symmetric difference of $L(\mathcal{B})$ and $L(\mathcal{C})$ is illustrated by the shaded regions of the Venn diagram shown in Figure 4.1.

Since we know from earlier that the class of regular languages is closed under union, complement, and intersection, we can combine all of these

closure properties to construct a deterministic finite automaton $\mathcal{D}$ whose language is $L(\mathcal{D}) = L(\mathcal{B}) \triangle L(\mathcal{C})$. We will use this finite automaton $\mathcal{D}$ in our decision algorithm for the equivalence problem.

The idea behind this decision algorithm, as we mentioned before, is to test equivalence by testing the emptiness of the symmetric difference language. If $L(\mathcal{D})$ is empty, then we know that all words belong to the intersection of $L(\mathcal{B})$ and $L(\mathcal{C})$, and therefore $L(\mathcal{B}) = L(\mathcal{C})$.

Theorem 4.7

EQ_{DFA} is decidable.

Proof. Construct a Turing machine $\mathcal{M}_{\text{EQDFA}}$ that takes as input $\langle \mathcal{B}, \mathcal{C} \rangle$, where $\mathcal{B}$ and $\mathcal{C}$ are deterministic finite automata, and performs the following steps:

$\mathcal{M}_{\text{EQDFA}}$

1. Construct a deterministic finite automaton $\mathcal{D}$ recognizing the language $L(\mathcal{D}) = L(\mathcal{B}) \triangle L(\mathcal{C})$.
2. Run $\mathcal{M}_{\text{EDFA}}$ from the proof of Theorem 4.3 on input $\langle \mathcal{D} \rangle$.
3. If $\mathcal{M}_{\text{EDFA}}$ accepts, then accept. Otherwise, reject.

This Turing machine accepts its input $\langle \mathcal{B}, \mathcal{C} \rangle$ if and only if $\mathcal{M}_{\text{EDFA}}$ accepts its input $\langle \mathcal{D} \rangle$, where $\mathcal{D}$ is the deterministic finite automaton recognizing the symmetric difference language. This occurs only when the symmetric difference language is empty, and thus all of the words in both $L(\mathcal{B})$ and $L(\mathcal{C})$ appear in the intersection of these two languages; that is, $L(\mathcal{B}) = L(\mathcal{C})$. Likewise, the Turing machine rejects if and only if $\mathcal{M}_{\text{EDFA}}$ rejects, which means that the symmetric difference language is nonempty and so $L(\mathcal{B}) \neq L(\mathcal{C})$. Since both the process of constructing $\mathcal{D}$ and the process of running $\mathcal{M}_{\text{EDFA}}$ are finite, the Turing machine will never become trapped in an infinite computation. Thus, $\mathcal{M}_{\text{EQDFA}}$ decides the equivalence problem for deterministic finite automata. ■

Of course, we immediately have the usual corollary to accompany this result.

Corollary 4.8

EQ_{NFA} and EQ_{RE} are decidable.

Inclusion Problem

Soon enough, we'll have a positive decidability result for the inclusion problem to go along with our collection of other positive decidability results.

4.2. DECIDABLE PROBLEMS FOR CONTEXT-FREE LANGUAGES

Moving on to the class of context-free languages, we will again consider each of the common decision problems in turn, but here we must make a decision: much like we chose to prove regular language decidability results with the deterministic finite automaton model, do we prove context-free decidability results with the pushdown automaton model or with the context-free grammar model?

If we were to go with the pushdown automaton approach, we would have the familiarity in our proofs of simulating the computation of a simpler machine on a Turing machine, exactly as we did in the previous section. On the other hand, we would need to devise some way of managing the stack of this machine as we simulate its computation, and this added complexity would make our proofs somewhat more difficult.

By contrast, going with the context-free grammar approach, we can easily simulate a derivation (which is a computation in disguise) just by manipulating the rules of the context-free grammar in an appropriate manner. Since we know that context-free grammars and pushdown automata are equivalent in terms of recognition power, we lose nothing by selecting the grammar model over the automaton model, and so we will make our lives easier and take the grammar approach in this section.

Membership Problem

As before, we begin with the simplest of decision problems. If we're given an input $\langle G, w \rangle$, where G is a context-free grammar and w is a word over the grammar's terminal alphabet Σ, the membership problem asks whether G is capable of generating the word w.

The naïve approach to determining whether G generates w would have us check every possible derivation using the rules of G. However, this isn't a good approach, since in the worst case our algorithm may need to check infinitely many derivations. In fact, if we tried this approach and G actually couldn't generate w, then our algorithm would keep checking derivations fruitlessly and would never halt! Thus, while the naïve approach *semidecides* the problem, it doesn't *decide* the problem.

We must somehow ensure that we only check some finite number of derivations in the process of testing membership. Thinking back to our

discussion of Chomsky normal form, we made one important observation that will help us greatly: recall that if a context-free grammar in Chomsky normal form generates a word w, then the derivation of w will take $2|w| - 1$ steps. This allows us to place an upper bound on the length of the derivation and, therefore, a limit on the number of steps performed by our membership-testing algorithm!

Using this observation, our decision algorithm will first convert its given context-free grammar G to Chomsky normal form and then only check candidate derivations of w that take $2|w| - 1$ steps. If w can indeed be generated by G, then its derivation will be found by the algorithm.

Theorem 4.9

A_{CFG} is decidable.

Proof. Construct a Turing machine $\mathcal{M}_{\text{ACFG}}$ that takes as input $\langle G, w \rangle$, where G is a context-free grammar and w is a word, and performs the following steps:

$\mathcal{M}_{\text{ACFG}}$

1. Convert G to an equivalent grammar G' in Chomsky normal form.
2. Check the length of w:
   - If $|w| = 0$, then list all derivations using G' that take a single step.
   - If $|w| \geq 1$, then list all derivations using G' that take $2|w| - 1$ steps.
3. If any of these derivations generate w, then accept. Otherwise, reject.

This Turing machine accepts its input $\langle G, w \rangle$ if and only if the context-free grammar G is capable of generating the word w; that is, if and only if $w \in L(G)$. Likewise, the Turing machine rejects if and only if no derivation exists that allows G to generate w; that is, if and only if $w \notin L(G)$. Since the process of converting G to Chomsky normal form as well as the process of listing all derivations of a given length are finite, the Turing machine will never become trapped in an infinite computation. Thus, $\mathcal{M}_{\text{ACFG}}$ decides the membership problem for context-free grammars. ■

Unsurprisingly, the same positive decidability result holds for the push-down automaton model.

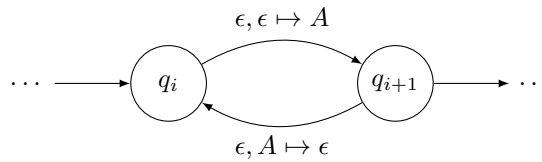
Corollary 4.10

A_{PDA} is decidable.

Proof. Given a pushdown automaton $\mathcal{P}$, we can convert it to an equivalent context-free grammar G and run $\mathcal{M}_{\text{ACFG}}$ from the proof of Theorem 4.9 on the input $\langle G, w \rangle$. ■

Reviewing our proof of Theorem 4.9 may make it more evident why we chose context-free grammars as our model in this section instead of pushdown automata: with grammars, we can obtain a concrete upper bound on how much work our Turing machine has to do before producing an answer.

On the other hand, if we're given a pushdown automaton as input, then we know we can simulate the computation of this automaton on a Turing machine in a rather straightforward manner. However, there's no way for us to predict in advance whether some branch of the pushdown automaton's computation tree will go on forever without accepting. Consider, for example, the branch of the computation tree that corresponds to following these two transitions repeatedly:



If our Turing machine naïvely simulates the computation of the pushdown automaton by descending into individual branches of the computation tree one-by-one, then it could fall into this infinite-length branch trap. As a consequence, we can't guarantee that our Turing machine decides whether or not the pushdown automaton accepts its input word, and so we can't establish a positive decidability result for the membership problem unless we put in the additional work to avoid infinite-length branches in the computation tree.

By comparison, doesn't our context-free grammar approach seem so much more appealing?

Emptiness Problem

For the emptiness problem, we again have a naïve approach to check whether $L(G) = \emptyset$ for some context-free grammar G: for all words w over the terminal alphabet Σ, verify that there exists no derivation $S \Rightarrow^* w$ from the start nonterminal S to some sequence of terminal symbols w. This is, of course,

not a good approach for the same reason as before: the pesky notion of infinity gets in our way. There is an infinite number of words over Σ for which we must test membership, so if the language $L(G)$ truly were empty, an algorithm following this approach would never halt. We would just keep checking and rejecting candidate words!

Instead, we will take an opposite approach to testing the emptiness of the grammar's language. Instead of checking that no sequence of *terminal* symbols is generated by the grammar, we will check, for each *nonterminal* symbol, whether that individual symbol yields some sequence of previously marked terminal symbols. If this is the case, then we will mark that nonterminal symbol. Then, whenever a marked nonterminal symbol appears in a derivation, we know that some sequence of terminal symbols will appear later in the derivation, and thus the grammar's language is nonempty.

Our decision algorithm for E_{CFG} works a lot like our decision algorithm for E_{DFA}, except backwards: while the algorithm for E_{DFA} marked states starting from the initial state and leading to a final state, our algorithm for E_{CFG} will mark symbols starting from the terminal symbols and returning to the start nonterminal.

Theorem 4.11

E_{CFG} is decidable.

Proof. Construct a Turing machine $\mathcal{M}_{\text{ECFG}}$ that takes as input $\langle G \rangle$, where G is a context-free grammar, and performs the following steps:

$\mathcal{M}_{\text{ECFG}}$

1. Mark all terminal symbols in G.
2. If G has a rule of the form $A \to \alpha_1 \dots \alpha_k$ and each symbol $\alpha_1, \dots, \alpha_k$ has already been marked, then mark the symbol A. Repeat until no new symbols are marked.
3. If the start nonterminal of G has not been marked, then accept. Otherwise, reject.

This Turing machine accepts its input $\langle G \rangle$ if and only if there exists no derivation $S \Rightarrow^* w$ in the context-free grammar G from the start nonterminal S to some sequence of terminal symbols w, and the Turing machine rejects if and only if such a derivation exists in G. Since the number of terminal symbols to mark and the number of rules of G to check are both finite, the Turing machine will never become trapped in an infinite computation. Thus, $\mathcal{M}_{\text{ECFG}}$ decides the emptiness problem for context-free grammars. ■

As we have come to expect, the same result holds for pushdown automata.

Corollary 4.12

E_{PDA} is decidable.

4.3. AN UNDECIDABLE PROBLEM FOR TURING MACHINES

After studying so many decision problems with positive decidability results, one might become convinced that our models of computation are capable of deciding anything we throw at it. Unfortunately, this is not the case: our regular and context-free models happen to be just simple enough to allow us to obtain the positive decidability results we saw in the previous sections, but turning to our strongest model of computation—the Turing machine—things begin to come apart.

But wait, since Turing machines are so powerful, doesn't that mean they can solve all kinds of problems? Indeed, that's true, but they can't *decide* many problems. Remember, in order to decide a problem, the Turing machine must always halt and either accept or reject the input word it was given. As we observed earlier, the most common issue the Turing machine may encounter comes in the "halt" step, since there's no guarantee that the machine will halt on every input word.

Indeed, even the most basic of decision problems is rendered undecidable on a Turing machine, simply because of that fundamental limitation that the machine may get caught in an infinite loop during its computation. We've proved that the membership problems for regular languages and for context-free languages are both decidable by virtue of the fact that the models recognizing such classes of languages always halt and either accept and reject their input words. For Turing machines, though, we lose this valuable decidability property.

Before we continue, let's review the notions of decidability and semidecidability from the previous chapter. Suppose that $\mathcal{M}$ is a Turing machine recognizing a language $L(\mathcal{M})$. Recall that:

- $L(\mathcal{M})$ is *decidable* if
  - whenever $w \in L(\mathcal{M})$, $\mathcal{M}$ accepts w, while
  - whenever $w \notin L(\mathcal{M})$, $\mathcal{M}$ rejects w; and
- $L(\mathcal{M})$ is *semidecidable* if
  - whenever $w \in L(\mathcal{M})$, $\mathcal{M}$ accepts w, while

– whenever $w \notin L(\mathcal{M})$, $\mathcal{M}$ either rejects w or enters an infinite loop.

It's quite straightforward to show that the membership problem for Turing machines, A_{TM}, is at least semidecidable. All we need to do is simulate the computation of $\mathcal{M}$ on w! No matter what $\mathcal{M}$ does—accept, reject, or loop forever—its behaviour matches our definition of semidecidability.

Theorem 4.13

A_{TM} is semidecidable.

Proof. Construct a Turing machine $\mathcal{M}_{\text{ATM}}$ that takes as input $\langle \mathcal{M}, w \rangle$, where $\mathcal{M}$ is a Turing machine and w is a word, and performs the following steps:

$\mathcal{M}_{\text{ATM}}$

1. Simulate $\mathcal{M}$ on input w.
2. If $\mathcal{M}$ ever enters its accepting state q_{accept}, then accept. If $\mathcal{M}$ ever enters its rejecting state q_{reject}, then reject.

This Turing machine accepts its input $\langle \mathcal{M}, w \rangle$ if and only if the Turing machine $\mathcal{M}$ accepts its input w, and the Turing machine rejects if and only if $\mathcal{M}$ rejects. If $\mathcal{M}$ enters an infinite loop on the input w, then $\mathcal{M}_{\text{ATM}}$ will likewise loop forever. Thus, $\mathcal{M}_{\text{ATM}}$ semidecides the membership problem for Turing machines. ■

The machine $\mathcal{M}_{\text{ATM}}$ that we constructed in the proof of Theorem 4.13 looks quite similar to the machine $\mathcal{M}_{\text{ADFA}}$ we constructed in the proof of Theorem 4.1 to decide A_{DFA}. However, unlike the machine $\mathcal{M}_{\text{ADFA}}$, this machine $\mathcal{M}_{\text{ATM}}$ only semidecides A_{TM}, since it has no way of explicitly rejecting its input word if it gets caught in an infinite loop. The machine $\mathcal{M}_{\text{ATM}}$ can't even guess that it's looping infinitely and "short circuit" its computation to automatically reject, for the plain reason that if the machine has performed 1 000 000 steps, say, then there's no general way of determining whether it will reach the accepting or rejecting state on step 1 000 001.

Note also that $\mathcal{M}_{\text{ATM}}$ is the canonical example of a universal Turing machine, since we can give it an encoding of any Turing machine $\mathcal{M}$ and any input word w and it will simulate the computation of that Turing machine on that input word.

Now, getting back to our main point, we must prove that A_{TM} is undecidable; that is to say, A_{TM} is *not decidable*. The technique we will use

is essentially a proof by contradiction: we will assume that we have some machine capable of deciding A_{TM}, and then show that the existence of such a machine leads to a logical absurdity when we give a nefariously crafted input word to that machine.

Theorem 4.14

A_{TM} is undecidable.

Proof. Assume by way of contradiction that A_{TM} is decidable, and suppose that $\mathcal{M}_{\mathrm{A}}$ is a Turing machine that decides A_{TM}. Given an encoding of a Turing machine $\mathcal{M}$ and an input word w, $\mathcal{M}_{\mathrm{A}}$ produces the same answer that $\mathcal{M}$ would produce:

- $\mathcal{M}_{\mathrm{A}}$ accepts $\langle \mathcal{M}, w \rangle$ if $\mathcal{M}$ accepts w; and
- $\mathcal{M}_{\mathrm{A}}$ rejects $\langle \mathcal{M}, w \rangle$ if $\mathcal{M}$ rejects w.

Note that we don't know exactly how $\mathcal{M}_{\mathrm{A}}$ decides the membership problem: we only assume that it is capable of doing it.

We now construct a new Turing machine $\mathcal{X}$ that relies on using $\mathcal{M}_{\mathrm{A}}$ as an intermediate step of its computation. The machine $\mathcal{X}$ takes as input $\langle \mathcal{M} \rangle$, where $\mathcal{M}$ is a Turing machine, and performs the following steps:

$\mathcal{X}$

1. Run $\mathcal{M}_{\mathrm{A}}$ on input $\langle \mathcal{M}, \langle \mathcal{M} \rangle \rangle$.
2. If $\mathcal{M}_{\mathrm{A}}$ accepts, then *reject*. If $\mathcal{M}_{\mathrm{A}}$ rejects, then *accept*.

Here, we see that $\mathcal{X}$ gives as input to $\mathcal{M}_{\mathrm{A}}$ exactly what $\mathcal{M}_{\mathrm{A}}$ expects to receive as input: an encoding of a Turing machine $\mathcal{M}$, together with an input word to $\mathcal{M}$ that happens to be the encoded description of the Turing machine $\mathcal{M}$ itself. This is allowed, since an encoded description of a Turing machine is nothing more than a sequence of symbols just like any other input word. After $\mathcal{M}_{\mathrm{A}}$ halts and produces its answer, $\mathcal{X}$ produces the opposite answer to whatever $\mathcal{M}_{\mathrm{A}}$ gave:

- $\mathcal{X}$ accepts $\langle \mathcal{M} \rangle$ if $\mathcal{M}$ rejects $\langle \mathcal{M} \rangle$; and
- $\mathcal{X}$ rejects $\langle \mathcal{M} \rangle$ if $\mathcal{M}$ accepts $\langle \mathcal{M} \rangle$.

Figure 4.2 depicts the behaviour of $\mathcal{X}$ visually. Observe in this figure that $\mathcal{M}_{\mathrm{A}}$ is a "black box", because as we remarked earlier, we don't know exactly how $\mathcal{M}_{\mathrm{A}}$ decides A_{TM}.

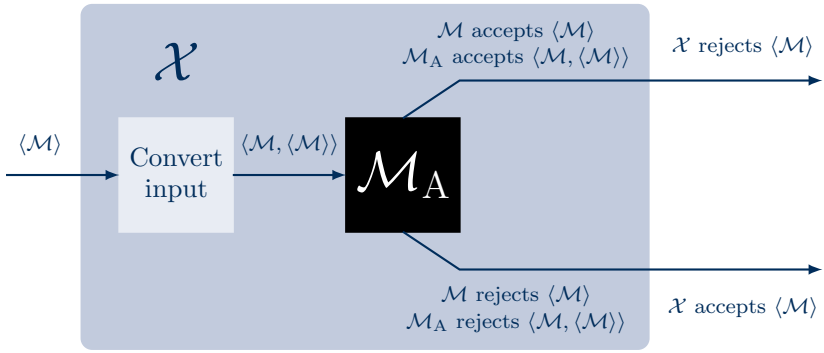


Figure 4.2. The Turing machine $\mathcal{X}$ from the proof of Theorem 4.14.

Now, consider what happens when we give $\mathcal{X}$ an encoding of *itself* as input. Suppose we give $\mathcal{X}$ the input $\langle \mathcal{X} \rangle$, so that $\mathcal{X}$ runs $\mathcal{M}_{\mathrm{A}}$ on the input $\langle \mathcal{X}, \langle \mathcal{X} \rangle \rangle$. Tracing through the behaviour of $\mathcal{M}_{\mathrm{A}}$ followed by $\mathcal{X}$, we can see that:

- $\mathcal{X}$ accepts $\langle \mathcal{X} \rangle$ if $\mathcal{X}$ rejects $\langle \mathcal{X} \rangle$; and
- $\mathcal{X}$ rejects $\langle \mathcal{X} \rangle$ if $\mathcal{X}$ accepts $\langle \mathcal{X} \rangle$.

In either case, the actual computation of $\mathcal{X}$ on the input $\langle \mathcal{X} \rangle$ must return the opposite answer as the simulated computation of $\mathcal{X}$ on the input $\langle \mathcal{X} \rangle$, which is of course impossible! As a result, we obtain a contradiction. Since the only assumption in our proof is about the existence of $\mathcal{M}_{\mathrm{A}}$, we conclude that no such machine $\mathcal{M}_{\mathrm{A}}$ can exist to decide A_{TM}. ■

Diagonalization. The underlying concept that makes the proof of Theorem 4.14 work is *Cantor's diagonal argument*, named for the mathematician Georg Cantor, who used the argument to demonstrate the existence of uncountably infinite sets [1891]. An *uncountably infinite* set is one whose elements cannot be placed into a bijection with the *countably infinite* set of natural numbers $\mathbb{N}$.

The crux of the diagonal argument is as follows: if we take a set T of all infinite-length binary words and assume that this set is countably infinite, then we can enumerate (that is, *count*) all of the elements of T, and we can denote this by writing something like $T = \{t_0, t_1, t_2, \dots\}$. However, using the set T itself, we can construct a new word s that doesn't belong to T by taking the ith symbol of s to be the complement of the ith symbol in

$$
\begin{array}{lcccccccccccc}
t_0 & = & \underline{0} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots \\
t_1 & = & 1 & \underline{1} & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \cdots \\
t_2 & = & 0 & 0 & \underline{1} & 0 & 1 & 1 & 0 & 1 & 1 & 1 & \cdots \\
t_3 & = & 0 & 1 & 1 & \underline{0} & 1 & 0 & 0 & 1 & 1 & 0 & \cdots \\
t_4 & = & 1 & 0 & 1 & 0 & \underline{1} & 0 & 1 & 0 & 1 & 0 & \cdots \\
t_5 & = & 1 & 1 & 0 & 0 & 1 & \underline{1} & 0 & 0 & 1 & 1 & \cdots \\
t_6 & = & 0 & 0 & 0 & 1 & 1 & 1 & \underline{0} & 0 & 0 & 1 & \cdots \\
t_7 & = & 1 & 0 & 0 & 1 & 0 & 1 & 1 & \underline{0} & 0 & 1 & \cdots \\
t_8 & = & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & \underline{0} & 1 & \cdots \\
t_9 & = & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & \underline{0} & \cdots \\
 & \vdots & & & & & & & & & & & \\
\hline
s & = & \underline{1} & \underline{0} & \underline{0} & \underline{1} & \underline{0} & \underline{0} & \underline{1} & \underline{1} & \underline{1} & \underline{1} & \cdots
\end{array}
$$

Figure 4.3. Constructing a new word s that doesn't belong to the set T using Cantor's diagonal argument.

the word t_i. This means that if the ith symbol in the word t_i is `0`, then we set the ith symbol of s to be `1`, and vice versa. This construction process is illustrated in Figure 4.3.

In the process of constructing this new word s, we can see that it differs from every other word in T in at least one position; namely, s differs from t_i at least in position i. (For example, compare s to t_9 in Figure 4.3.) Therefore, it's impossible for s to have already been included in T. Since s is not an element of T, we could not have accounted for s in our enumeration of T. Moreover, since we can create an infinite number of new words in this way, and none of these new words is accounted for in our enumeration, T must not be countably infinite.

Cantor's diagonal argument is popularly used to prove that the set of real numbers $\mathbb{R}$ is uncountably infinite, but the same argument can be applied to Turing machines in order to prove that A_{TM} is undecidable. If we could enumerate all Turing machines, then we could construct a table where each row corresponds to a Turing machine $\mathcal{M}_i$ and each column corresponds to an encoding of a Turing machine $\langle \mathcal{M}_j \rangle$. Then, the entries of the table could be taken to be the result of running our supposed machine $\mathcal{M}_{\text{A}}$ on the input $\langle \mathcal{M}_i, \langle \mathcal{M}_j \rangle \rangle$, where each entry is either "accept" or "reject" depending on the answer that $\mathcal{M}_{\text{A}}$ produced.

	$\langle \mathcal{M}_0 \rangle$	$\langle \mathcal{M}_1 \rangle$	$\langle \mathcal{M}_2 \rangle$	$\langle \mathcal{M}_3 \rangle$	$\cdots$
$\mathcal{M}_0$	accept	accept	reject	reject	
$\mathcal{M}_1$	accept	reject	reject	accept	
$\mathcal{M}_2$	reject	reject	accept	accept	
$\mathcal{M}_3$	accept	accept	accept	reject	
$\vdots$					$\ddots$

By the fact that $\mathcal{X}$ itself is a Turing machine, we know that $\mathcal{X}$ must appear somewhere in our enumerated list, and thus in our table. We also know each entry in the row corresponding to $\mathcal{X}$: since $\mathcal{X}$ produces the opposite answer to whatever $\mathcal{M}_\text{A}$ produces on the input $\langle \mathcal{M}_i, \langle \mathcal{M}_i \rangle \rangle$, the entry in column i must be the opposite of the entry at position (i, i) of the table. But again, therein lies the problem: what is the table entry at the position corresponding to $\langle \mathcal{X}, \langle \mathcal{X} \rangle \rangle$?

	$\langle \mathcal{M}_0 \rangle$	$\langle \mathcal{M}_1 \rangle$	$\langle \mathcal{M}_2 \rangle$	$\langle \mathcal{M}_3 \rangle$	$\cdots$	$\langle \mathcal{X} \rangle$	$\cdots$
$\mathcal{M}_0$	accept	accept	reject	reject		accept	
$\mathcal{M}_1$	accept	reject	reject	accept		reject	
$\mathcal{M}_2$	reject	reject	accept	accept		reject	
$\mathcal{M}_3$	accept	accept	accept	reject		accept	
$\vdots$					$\ddots$		
$\mathcal{X}$	reject	accept	reject	accept		?	
$\vdots$							$\ddots$

The entry at this position of the table somehow needs to be the opposite of itself, and so we arrive at the same contradiction as before. The Turing machine $\mathcal{X}$ cannot exist, which means that the Turing machine $\mathcal{M}_\text{A}$ cannot exist, which means that A_TM is undecidable.

4.4. A NON-SEMIDECIDABLE PROBLEM FOR TURING MACHINES

Now that we know the membership problem A_TM is semidecidable but not decidable, we can be fairly sure that obtaining a positive decidability result for other Turing machine decision problems is hopeless. After all, if we can't even decide the membership of a word in a Turing machine's language, what other questions about the language can we hope to decide?

In order to crush our spirits even further, let's prove another fact about languages and their relation to Turing machines. Earlier, we showed that the class of decidable languages was a proper subset of the class of semidecidable languages by proving that A_TM is semidecidable, but not decidable. The semidecidable languages are the "largest" class we've studied thus far in this book. However, semidecidability is not the upper limit when it comes to language classes! We can in fact show via a familiar argument that there exist languages that lie even outside of the class of semidecidable languages.

Theorem 4.15

There exist languages that are not semidecidable.

Proof. In order for a language to be (at least) semidecidable, it must be recognized by some Turing machine. Recall that we can describe

any Turing machine $\mathcal{M}$ with a unique encoding $\langle \mathcal{M} \rangle$. Although there is an infinite number of Turing machines, each of these encodings has a finite length, so we can enumerate them (say, lexicographically). By associating each encoding with a natural number according to that encoding's position in our enumeration, we obtain a bijection between encodings and the set of natural numbers $\mathbb{N}$, which means that the set of all Turing machines is countably infinite.

Now, consider the set of all words over some alphabet Σ. Again, even though this set is infinite, each of these words has a finite length, so we can enumerate them (say, lexicographically); denote the enumerated set by $\{w_0, w_1, w_2, \dots\}$. From this, we can define languages in terms of the words w_i that belong to the language: we represent each language as an infinite-length binary word where the ith symbol of the word is `1` if $w_i \in L$, and `0` otherwise.

However, the set of all languages cannot be countably infinite: by Cantor's diagonal argument, if we could enumerate all of the infinite-length binary words representing every language, then it would be possible to construct a new infinite-length word representing a language not included in our enumeration. Therefore, the set of all languages must be uncountably infinite, and so there must exist languages that are not recognized by any Turing machine. ■

Looking a bit more closely at our proof, we find ourselves confronted by a perhaps-uncomfortable fact: not only do non-semidecidable languages exist, but *uncountably many* non-semidecidable languages exist. Thus, if we were to choose a language uniformly at random from the set of all languages, that language will—in the probabilistic sense—almost never be decidable by any Turing machine. In other words, the probability that a computer can solve the problem corresponding to our randomly chosen language is *zero*!

Theorem 4.15 is an example of a theorem having a *non-constructive* proof: the proof tells us that there exist languages that are not semidecidable, but it doesn't actually give us an example of such a language. We will, of course, see such an example momentarily, but first we must come up with one more definition.

Co-Semidecidability. The property of semidecidability, at its core, is non-symmetric. Recall the decision problem A_{TM}: we are guaranteed to get an answer in the positive case, if the Turing machine accepts its input, but in the negative case if the Turing machine loops forever, we will never end up with an answer.

What if we wanted to guarantee answers in the negative case, though? Consider how we might define a "looping" decision problem, where we want

to decide whether a Turing machine *will* loop forever on some input. In the positive case, if the Turing machine does in fact loop forever, we will again never end up with an answer. But in the negative case, if the Turing machine halts, we are guaranteed to get an answer! We can't reasonably call this new decision problem semidecidable, since its behaviour doesn't line up with our definition of semidecidability. We need some new terminology.

Let $\overline{L}$ denote the complement of a language L. If we know something about this language L—namely, that it is semidecidable—then we can draw a straightforward conclusion about the complement of this language.

Definition 4.16 (Co-semidecidable language)

If a language L is semidecidable, then we say that the complement of this language, $\overline{L}$, is co-semidecidable.

Alternatively, given a Turing machine $\mathcal{M}$, we say that the language $L(\mathcal{M})$ is co-semidecidable if,

- whenever $w \in L(\mathcal{M})$, then either $\mathcal{M}$ accepts w or $\mathcal{M}$ enters an infinite loop; and
- whenever $w \notin L(\mathcal{M})$, then $\mathcal{M}$ rejects w.

Observe that the definition of co-semidecidability takes the non-symmetric definition of semidecidability and flips it around: now, we are guaranteed an answer in the negative case when $w \notin L(\mathcal{M})$, whereas semidecidability guaranteed us an answer in the positive case when $w \in L(\mathcal{M})$.

Saying that $\overline{L}$ is co-semidecidable is another way of saying that we can *semidecide* the *co*mplement of $\overline{L}$; that is, we can semidecide $\overline{\overline{L}} = L$.

One might wonder, if we have a Turing machine that semidecides some language L, whether we need a separate Turing machine to co-semidecide $\overline{L}$? In fact, we don't always need a new machine: the original language L and the language of all words *not* in the complement language $\overline{L}$ are one and the same. (Think about why this is the case.) Thus, if we want to, we can use the existing Turing machine $\mathcal{M}$ that semidecides L also to co-semidecide $\overline{L}$: if $\mathcal{M}$ accepts some input, then we know that input cannot belong to $\overline{L}$, and so we can treat it as if it were rejected.

Going a step further, if we know something about the co-semidecidability of a language, does this say anything about the semidecidability of the same language? By Definition 4.16, we know that $\overline{L}$ is co-semidecidable whenever L is semidecidable, but this in itself does not suggest or imply that $\overline{L}$ is semidecidable. If we could further show that $\overline{L}$ were also semidecidable, then L would be co-semidecidable, again by our definition. In this situation

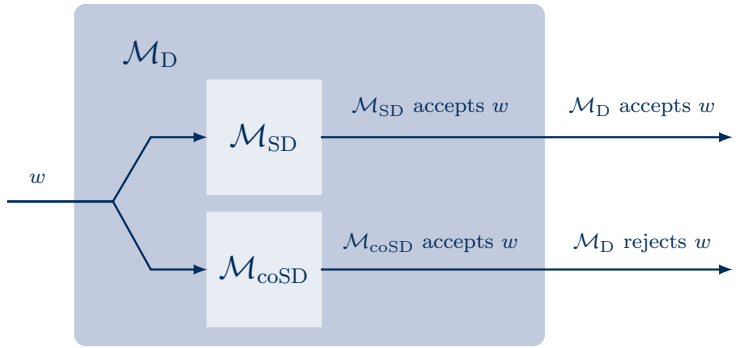


Figure 4.4. The Turing machine $\mathcal{M}_{\mathrm{D}}$ from the proof of Theorem 4.17.

where L is both semidecidable and co-semidecidable, we can "upgrade" the class to which L belongs.

Theorem 4.17

A language L is decidable if and only if L is both semidecidable and co-semidecidable.

Proof. ($\Rightarrow$): Suppose L is decidable. Since all decidable languages are also semidecidable, we have that L is semidecidable. If we take the Turing machine deciding L and exchange the accepting and rejecting outputs, then we get a machine that decides $\overline{L}$. Thus, $\overline{L}$ is semidecidable, and so L is co-semidecidable.

($\Leftarrow$): Suppose L is both semidecidable and co-semidecidable. Then there exists a Turing machine $\mathcal{M}_{\mathrm{SD}}$ semideciding L and a Turing machine $\mathcal{M}_{\mathrm{coSD}}$ co-semideciding L. Using these two Turing machines, we can construct a Turing machine $\mathcal{M}_{\mathrm{D}}$ that takes as input a word w and performs the following steps:

$\mathcal{M}_{\mathrm{D}}$

1. Run both $\mathcal{M}_{\mathrm{SD}}$ and $\mathcal{M}_{\mathrm{coSD}}$ on the input word w in parallel.
2. If $\mathcal{M}_{\mathrm{SD}}$ accepts, then accept. If $\mathcal{M}_{\mathrm{coSD}}$ accepts, then reject.

Figure 4.4 depicts the behaviour of $\mathcal{M}_{\mathrm{D}}$ visually.

Since every word w must belong to either L or $\overline{L}$, either $\mathcal{M}_{\mathrm{SD}}$ or $\mathcal{M}_{\mathrm{coSD}}$ must accept w. Moreover, since $\mathcal{M}_{\mathrm{D}}$ halts whenever either $\mathcal{M}_{\mathrm{SD}}$ or $\mathcal{M}_{\mathrm{coSD}}$ accepts, we have that $\mathcal{M}_{\mathrm{D}}$ decides L, and so L is decidable. ■

The Non-membership Problem. To motivate our study of co-semidecidability, we considered a "looping" decision problem, but our formulation of this problem was rather ad hoc and imprecise. Since co-semidecidability is fundamentally a property of the complement of a language, let's further motivate our study by taking an existing decision problem that we have already formally defined and studying its complement. Namely, consider the Turing machine "non-membership" problem:

$$\overline{A_{\mathsf{TM}}} = \{\langle \mathcal{M}, w\rangle \mid \mathcal{M} \text{ is a Turing machine that does } \textit{not} \text{ accept input word } w\}.$$

Taking all that we know about decidability, we can say that a decidable problem is one where we can always obtain an answer for both the problem itself and for the complement of the problem. Because both A_{DFA} and A_{CFG} are decidable, their Turing machines always halt and either accept or reject any input word given to them. This means that the corresponding "non-membership" problems $\overline{A_{\mathsf{DFA}}}$ and $\overline{A_{\mathsf{CFG}}}$ are also decidable, since we can use the same Turing machines to decide these problems by exchanging accept and reject outputs, and so studying these complementary decision problems isn't too interesting. The Turing machine "non-membership" problem $\overline{A_{\mathsf{TM}}}$, on the other hand, is different, since A_{TM} is only semidecidable. A semidecidable problem is one where we can always obtain an answer for the problem itself, but not necessarily for the complement of the problem. Thus, it makes sense for us to study $\overline{A_{\mathsf{TM}}}$ independently of A_{TM}.

So, what is the decidability status of $\overline{A_{\mathsf{TM}}}$? Instead of approaching a proof from first principles, as we did in showing that A_{TM} was undecidable, we will use a different (and easier!) approach with $\overline{A_{\mathsf{TM}}}$ that focuses on what we already know about A_{TM}.

Since Theorem 4.13 tells us that A_{TM} is semidecidable, we know that $\overline{A_{\mathsf{TM}}}$ is co-semidecidable by Definition 4.16. However, if $\overline{A_{\mathsf{TM}}}$ were also semidecidable, then Definition 4.16 would likewise tell us that A_{TM} is co-semidecidable, and this would present quite a problem! Therefore, we can conclude the following.

Theorem 4.18

$\overline{A_{\mathsf{TM}}}$ is not semidecidable.

Proof. Assume by way of contradiction that $\overline{A_{\mathsf{TM}}}$ is semidecidable. Then A_{TM} would be co-semidecidable, and by Theorem 4.17, A_{TM} would be decidable, which contradicts Theorem 4.14. ■

This theorem reveals a remarkable fact: $\overline{A_{\mathsf{TM}}}$ is not only undecidable as we might expect, but it's also non-semidecidable, and so it lies in an entirely different part of our language hierarchy!

As we observed at the beginning of the section, uncountably many non-semidecidable languages exist, and so we shouldn't be too surprised by the fact that we found an example of such a language. In the next chapter, we'll see a number of other examples of languages that are neither decidable nor semidecidable; in some cases, these languages are co-semidecidable, while in other cases, not even co-semidecidability holds.

CHAPTER NOTES

4.1. The decidability of the membership problem for the class of regular languages is practically folklore, in that the result follows immediately from simply giving an input word to a finite automaton. The decidability of the (non-)emptiness problem was established by Moore [1956], who used an argument different from ours that essentially resembles the pumping lemma for regular languages. As we observed, the decidability of the universality, equivalence, and inclusion problems all follow from the positive answer for the emptiness problem.

4.2. Bar-Hillel, Perles, and Shamir [1961] showed that both the membership problem (or "the property of sentencehood" in their terminology) and the emptiness problem are decidable for the class of context-free languages.

4.3. It can be said that the undecidability of A_{TM} was established by Turing [1936], although he didn't prove this exact result. Turing instead proved the undecidability of the *printing problem*, which takes a machine $\mathcal{M}$ and asks "whether $\mathcal{M}$ ever prints a given symbol (0 say)". We can, however, easily draw a connection between the printing problem and the membership problem by, for instance, modifying a given machine to write some special symbol "a" to its tape before reaching q_{accept}.

Although Cantor's diagonal argument appeared in 1891, it was not Cantor's first proof pertaining to uncountability. Two decades earlier, Cantor published a paper [1874] demonstrating, among other things, the uncountability of the set of real numbers using an approach inspired by Liouville's argument for the existence of transcendental numbers. For further details about Cantor's proofs, see the article by Gray [1994].

4.4. The argument we employed in the proof of Theorem 4.15 is essentially identical to that used by Cantor [1891], but applied to languages instead of general sets.

Theorem 4.17, on the relationship between decidability, semidecidability, and co-semidecidability, is due to Post [1944].

CHAPTER FIVE

PROVING UNDECIDABILITY

Thus far, we only have a couple of examples of decision problems that are undecidable, but these examples are big ones: the membership and non-membership problems for Turing machines. These particular problems being undecidable presents a huge issue for us, since the simple matter of testing membership of an input word in a Turing machine's language is fundamental to answering nearly any other question about the capabilities of a Turing machine.

We know that other undecidable languages exist—in fact, as we observed earlier, there are uncountably many such languages! And, as we might expect, many decision problems for Turing machines fall into this category. So how do we prove such problems truly are undecidable?

In the previous chapter, we combined the techniques of *arithmetization* and *diagonalization* to obtain our first undecidability result. Arithmetization is a fancy word that describes the process of encoding the behaviour of a Turing machine as a string of symbols; say, taking a `1` as the ith symbol if some Turing machine accepts the description of another Turing machine $\langle \mathcal{M}_i \rangle$ and `0` if it rejects the description. We then know from our discussion of diagonalization how these strings of symbols can be manipulated to produce a new string that leads to some absurd and undefined behaviour. But admittedly, undecidability proofs via diagonalization are lengthy and tedious. Is there some other (hopefully easier) way to prove that a decision problem is undecidable?

Enter the notion of *reducibility*. A reduction is a way for us to relate the difficulty of one decision problem to the difficulty of another. If we know how to turn one problem into another, then we can use an existing Turing machine or algorithm for the second problem also to decide the first problem. On the other hand, if we know how to turn one problem into another and we know that the first problem cannot be decided by any Turing machine or algorithm, then we can conclude that it's hopeless for us to decide the second problem. Therein lies the other (easier) way for us to prove undecidability: if we can turn either of our known-undecidable problems A_{TM} or $\overline{A_{\mathsf{TM}}}$ into some new problem, then we have our proof!

If a reduction seems a bit abstract based on what you've read so far, consider that you encounter examples of reductions in your daily life without even realizing it. For example, if you've recently gone to the library to sign out a book you want to read, the problem of locating that book in the stacks is no more difficult than the problem of searching for that book in the library's catalog. If you know how to use the library's catalog, then you know how to acquire the classification number of the book and where to find it on the shelf.

In this chapter, we will use reductions to prove that many decision problems for Turing machines are undecidable, including one of the most famous problems in all of computer science. Each of our proofs of undecidability will have a very similar structure, highlighting the versatility of this approach over our more tedious and ad hoc diagonalization approach. We will then put a spin on the idea of reducibility in order to go one step further and prove that certain decision problems pertaining to context-free languages are likewise undecidable.

5.1. MANY-ONE REDUCTIONS

In our proof showing that A_{TM} was undecidable, we constructed a Turing machine $\mathcal{X}$ that took as input $\langle \mathcal{M} \rangle$, the encoding of a Turing machine $\mathcal{M}$, converted that input to the form $\langle \mathcal{M}, \langle \mathcal{M} \rangle \rangle$, and gave that converted input to another Turing machine $\mathcal{M}_{\mathrm{A}}$. The machine $\mathcal{X}$ then used the output of $\mathcal{M}_{\mathrm{A}}$ to determine what its own output should be. This was the ad hoc diagonalization approach: what we wish to avoid going forward.

Generalizing this notion—that is, the notion of a Turing machine taking an input corresponding to some decision problem X, converting it into an input corresponding to some other decision problem Y, and then giving that converted input to another Turing machine for Y—gives us the foundation for proving undecidability using *many-one reductions* or, more generally, just *reductions*.

> *Remark.* There exist other kinds of reductions that we will study later in this book. In this chapter, though, we will only consider many-one reductions, so we will often use the word "reduction" here as a shorthand to refer to many-one reductions.

Computationally speaking, a many-one reduction is the middle step in our aforementioned generalization that converts the instance of the problem X into an instance of the other problem Y. The reason why this step is called a "many-one" reduction is because the conversion process is performed by a computable function, to which we were introduced in Section 3.1.3. If a computable function exists that gets us from X to Y, then we say that X is *many-one reducible* (or just *reducible*) to Y.

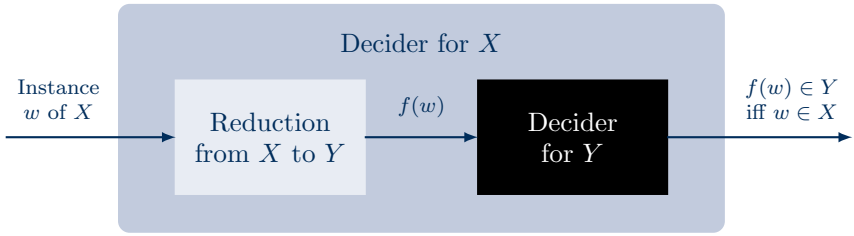


Figure 5.1. A Turing machine for X using the reduction $X \leq_{\mathrm{m}} Y$.

Definition 5.1 (Many-one reduction)

Given two decision problems X and Y over alphabets Σ and Γ, respectively, problem X is many-one reducible to problem Y if there exists a computable function $f\colon \Sigma^* \to \Gamma^*$ where, for all $w \in \Sigma^*$, $w \in X$ if and only if $f(w) \in Y$.

Having a reduction tells us not only that we can transform every instance w of X into an instance $f(w)$ of Y, but also that we *cannot* transform non-instances of X into instances of Y or vice versa. Moreover, since $w \in X$ if and only if $f(w) \in Y$, we know that the transformed instance will produce the same output as the original instance—since we're dealing with decision problems, this means that both w and $f(w)$ will give us either a "yes" answer or a "no" answer. This observation reveals something very useful indeed: since we'll get the same answer for the instance of Y as we would for the instance of X, we can use a reduction along with a decision algorithm for problem Y to decide the original problem X.

We denote a reduction from X to Y by the notation $X \leq_{\mathrm{m}} Y$. Figure 5.1 illustrates how such a reduction works, while Figure 5.2 shows how the computable function f maps elements from one set to another.

Reductions are in general not reversible, so the direction of a reduction is important: if $X \leq_{\mathrm{m}} Y$, then we say that we reduce *from* X *to* Y. Mixing up the direction of a reduction is a very common mistake made by the novice and the expert alike, so don't feel discouraged if it happens to you at least once.

If we have a reduction from X to Y, then we can make some claims about the relative difficulty of X based on what we know about Y, or vice versa. The existence of a reduction from X to Y implies that finding an answer to X is no more difficult than finding an answer to Y, or equivalently, finding an answer to Y is at least as difficult as finding an answer to X. This is because, as Figure 5.1 illustrates, we must use the decider for Y as an intermediate step in the overall decider for X. Thus, we have the

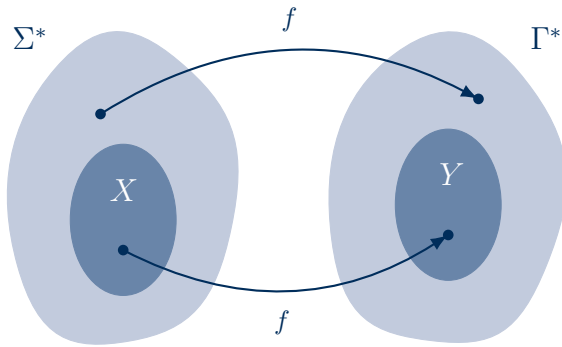


Figure 5.2. How a reduction $X \leq_{\mathrm{m}} Y$ maps elements. All elements in X are mapped by f to some element in Y, while all other elements in $\Sigma^* \setminus X$ are mapped by f to some other element in $\Gamma^* \setminus Y$.

following rules of thumb:

- if X reduces to Y and Y is "easy", then we know that X must similarly be "easy"; and
- if X reduces to Y and X is "hard", then we know that Y must similarly be "hard".

We can intuit about both of these rules of thumb as follows. If Y is "easy", then we already have a decider for Y, and we need only build the decider for X around the decider for Y. On the other hand, if X is "hard", then everything inside a supposed decider for X—including the decider for Y—must be "hard" for us to solve.

It's worth emphasizing once more that directionality is crucial with reductions. It wouldn't make sense for us to reduce from an "easy" problem to a "hard" problem, since our Turing machine could just decide the "easy" problem directly while skipping over the supposed decider for the "hard" problem: we don't need the reduction at all in this case.

> *If you are taller than someone who is tall, then you must be tall.*
> *But if someone tall is taller than you, you might be*
> *short or tall—we wouldn't yet know.*
>
> — JOEL DAVID HAMKINS

For now, we write "easy" and "hard" in quotation marks, since these notions are still informal. Soon, we will introduce complexity classes and define more precise notions of easiness and hardness for decision problems.

5.1.1. Properties of Reductions

Let's now establish some basic facts about the many-one reduction relation itself. We've already observed that reductions are in general not reversible, so we can say that the many-one reduction relation is not symmetric: if $X \leq_m Y$, then it is not always the case that $Y \leq_m X$ as well. However, we can prove two other nice properties.

Lemma 5.2

The many-one reduction relation $\leq_m$ is reflexive and transitive.

Proof. Let X, Y, and Z be arbitrary decision problems.

To show that $\leq_m$ is reflexive, take $f(x) = x$ as our computable function. Then $X \leq_m X$ for all decision problems X.

To show that $\leq_m$ is transitive, suppose that $X \leq_m Y$ by way of some computable function f and $Y \leq_m Z$ by way of some other computable function g. Then $X \leq_m Z$ by taking $h(x) = g(f(x))$ as our computable function. ■

Another fact about reductions that will come in handy for us later is that the many-one reduction relation is closed under complement.

Lemma 5.3

Given two decision problems X and Y, $X \leq_m Y$ if and only if $\overline{X} \leq_m \overline{Y}$.

Proof. Since $X \leq_m Y$, we know by Definition 5.1 that there exists some computable function f where, for all w, $w \in X$ if and only if $f(w) \in Y$.

If $w \in \overline{X}$, then $w \notin X$, so $f(w) \notin Y$ and thus $f(w) \in \overline{Y}$. Similarly, if $w \notin \overline{X}$, then $w \in X$, so $f(w) \in Y$ and thus $f(w) \notin \overline{Y}$. Therefore, $w \in \overline{X}$ if and only if $f(w) \in \overline{Y}$, and the same computable function f gives us the reduction $\overline{X} \leq_m \overline{Y}$. ■

5.1.2. Reductions, Decidability, and Semidecidability

We can combine the notion of a decidable problem with that of a reduction to allow us to characterize an unknown problem in terms of another known problem.

Theorem 5.4

If Y is decidable and $X \leq_m Y$, then X is decidable.

Proof. Since Y is decidable, there exists a Turing machine $\mathcal{M}_Y$ that de-

cides instances of Y. Moreover, since $X \leq_{\mathrm{m}} Y$, there exists a computable function f that reduces instances of X to instances of Y.

We construct a Turing machine $\mathcal{M}_X$ that takes as input a word w and performs the following steps:

$\mathcal{M}_X$

1. Compute $f(w)$ using the reduction $X \leq_{\mathrm{m}} Y$.
2. Run $\mathcal{M}_Y$ on the result $f(w)$.
3. If $\mathcal{M}_Y$ accepts, then accept. If $\mathcal{M}_Y$ rejects, then reject.

The computable function f—that is, our reduction from X to Y—tells us that if $f(w) \in Y$, then it must be the case that $w \in X$. On the other hand, if $f(w) \notin Y$, then $w \notin X$. In either case, we can use the answer produced by the Turing machine $\mathcal{M}_Y$ to obtain an answer for our original decision problem X and its original input w. ■

The main benefit of Theorem 5.4 is that, as long as we know two things—namely, that Y is decidable and that there exists a reduction $X \leq_{\mathrm{m}} Y$—we don't need to construct an entirely new Turing machine just to decide X. We can just sit back and let the existing Turing machine $\mathcal{M}_Y$ do all the work on the reduced instance of X!

Of course, the main focus of this chapter is proving undecidability, and so it makes sense for us to connect reductions to this notion as well. By taking the contrapositive of Theorem 5.4, we get the following important result that we will use frequently in future proofs.

Corollary 5.5

If X is undecidable and $X \leq_{\mathrm{m}} Y$, then Y is undecidable.

Indeed, if we have an undecidable problem X and a reduction $X \leq_{\mathrm{m}} Y$ to some other decision problem Y we wish to prove undecidable, Corollary 5.5 gives us a general template for such a proof:

1. Assume by way of contradiction that Y is decidable by some Turing machine $\mathcal{M}_Y$.
2. Construct the following Turing machine $\mathcal{M}_X$ that takes as input a word w and supposedly decides X using the machine $\mathcal{M}_Y$:

$\mathcal{M}_X$

1. **Compute $f(w)$ using the reduction $X \leq_{\mathrm{m}} Y$.**
2. Run $\mathcal{M}_Y$ on the result $f(w)$.
3. If $\mathcal{M}_Y$ accepts, then accept. If $\mathcal{M}_Y$ rejects, then reject.

3. Since X is undecidable, conclude that such machines $\mathcal{M}_X$ and $\mathcal{M}_Y$ cannot exist, and so Y cannot be decidable either.

The key part of any proof of undecidability that uses reductions comes in the first step of the description of $\mathcal{M}_X$, which we have emphasized in **boldface**. We must develop the reduction $X \leq_{\mathrm{m}} Y$ specifically for the decision problems X and Y under consideration in the proof, and so this step is the one that most often requires creative thinking and customization. Thankfully, the rest of the proof is mostly boilerplate and can be reused.


Another warning about the directionality of reductions: take careful note of the wordings of Theorem 5.4 and Corollary 5.5. If X is *decidable* and $X \leq_{\mathrm{m}} Y$, then we can't make any conclusions about the decidability of Y. It's possible that Y may be undecidable even if X is decidable.

We can make similar claims about semidecidability instead of decidability by using essentially the same proof as in Theorem 5.4. The difference here, of course, is that we no longer have the guarantee that our Turing machine $\mathcal{M}_Y$ will always halt.

Theorem 5.6

If Y is semidecidable and $X \leq_{\mathrm{m}} Y$, then X is semidecidable.

Again, taking the contrapositive gives us another important result that will come in handy later.

Corollary 5.7

If X is not semidecidable and $X \leq_{\mathrm{m}} Y$, then Y is not semidecidable.

5.2. THE HALTING PROBLEM

Most, if not all, of the code we write in our daily lives has the desirable behaviour that it eventually ends its computation and produces an output. For example, the following code can easily be seen to stop after some time:

```
i ← 1
for 2 ≤ j ≤ 10 do
    i ← i × j
return i
```

On the other hand, the following code will never stop, ceaselessly churning on and on until either the user intervenes or the computer shuts off:

```
i ← 0
while true do
    i ← i + 1
```

Does it happen to be the case that any useful code we write exhibits the behaviour of eventually stopping and producing something for us? Let's consider one more block of code, which has both a "while true" infinite loop and a "return" statement within that loop giving us a possible avenue to stop computing:

```
i ← 4
while true do
    if i is not the sum of two prime numbers then
        return false
    else
        print the two prime numbers that sum to i
        i ← i + 2
```

The utility of this code might be up for debate if you're not in any pressing need to know whether a number can be represented as a sum of two primes, but mathematicians would argue that this code is very useful: it will reach that internal "return" statement if and only if the Goldbach conjecture is false! So, will this code ever stop and return false? The Goldbach conjecture has been verified to hold for all even numbers up to 4×10^{18}—an incredibly large value—but no general proof or disproof is yet known. Thus, if we could determine whether this code will ever come to a halt (and assuming it doesn't do so due to a computer bug), it would be very big news in the mathematical world.

The question of whether code halts has deep connections and implications across computer science, from the most abstract theory to the most applied software engineering. Some infinite-looping behaviour is desirable in code—say, in software that polls the status of a computer—but in other code, it is very important for us to know whether or not we'll eventually get the

answer we desire. This has led to the *halting problem* becoming one of the most famous problems in all of computer science.

Although we have framed our discussion so far in terms of code, the halting problem formally asks whether the computation of a Turing machine halts on some given input word. Of course, thanks to the Church–Turing thesis in Section 3.6, we know that code and Turing machines are "the same" in the sense that code implements algorithms and algorithms can be computed on Turing machines. Thus, we can formulate the halting problem precisely as follows:

$$H_{\mathsf{TM}} = \{\langle \mathcal{M}, w \rangle \mid \mathcal{M} \text{ is a Turing machine that halts on input } w\}.$$

Observe that the formulation of H_{TM} looks very similar to that of A_{TM}. However, there exists a subtle difference between the two problems: A_{TM} asks not only whether a given Turing machine *halts*, but also whether it *accepts* a given input word. By contrast, H_{TM} only cares about whether the machine halts.

We can at least say that H_{TM} is semidecidable, since we can construct a Turing machine that takes as input a description of a Turing machine $\mathcal{M}$ along with an input word w and accepts if and only if the simulated computation of $\mathcal{M}$ on w halts.

Theorem 5.8

H_{TM} is semidecidable.

Proof. Construct a Turing machine $\mathcal{M}_{\text{HTM}}$ that takes as input $\langle \mathcal{M}, w \rangle$, where $\mathcal{M}$ is a Turing machine and w is a word, and performs the following steps:

$\mathcal{M}_{\text{HTM}}$

1. Run $\mathcal{M}$ on the input word w.
2. If $\mathcal{M}$ halts, then accept.

This Turing machine accepts its input $\langle \mathcal{M}, w \rangle$ if and only if the Turing machine $\mathcal{M}$ halts when given its input w. If $\mathcal{M}$ enters an infinite loop on the input w, then $\mathcal{M}_{\text{HTM}}$ will likewise loop forever. Thus, $\mathcal{M}_{\text{HTM}}$ semidecides the halting problem. ■

Now, we will prove the undecidability of H_{TM} by way of reduction from our known-undecidable problem A_{TM}. Note that reducing *from* A_{TM} *to* H_{TM} means that we can turn instances of A_{TM} into instances of H_{TM}, and since we know that A_{TM} is undecidable, this must mean that H_{TM} is similarly undecidable by Corollary 5.5.

Theorem 5.9

H_{TM} is undecidable.

Proof. Assume by way of contradiction that H_{TM} is decidable, and suppose that $\mathcal{M}_{\mathrm{H}}$ is a Turing machine that decides H_{TM}.

We construct a new Turing machine $\mathcal{M}_{\mathrm{A}}$ for the membership problem A_{TM} that uses the reduction $A_{\mathsf{TM}} \leq_{\mathrm{m}} H_{\mathsf{TM}}$. The machine $\mathcal{M}_{\mathrm{A}}$ takes as input $\langle \mathcal{M}, w \rangle$, where $\mathcal{M}$ is a Turing machine and w is an input word, and performs the following steps:

$\mathcal{M}_{\mathrm{A}}$

1. Define a computable function f that takes as input $\langle \mathcal{M}, w \rangle$ and produces as output $\langle \mathcal{M}', w \rangle$, where $\mathcal{M}'$ behaves as follows:

   $\mathcal{M}'$

   1. Run $\mathcal{M}$ on the input word given to $\mathcal{M}'$.
   2. If $\mathcal{M}$ accepts, then accept. If $\mathcal{M}$ rejects, then loop forever.

2. Run $\mathcal{M}_{\mathrm{H}}$ on the result $\langle \mathcal{M}', w \rangle$ produced by f.
3. If $\mathcal{M}_{\mathrm{H}}$ accepts, then accept. If $\mathcal{M}_{\mathrm{H}}$ rejects, then reject.

Observe that if $\langle \mathcal{M}, w \rangle \in A_{\mathsf{TM}}$, then $\mathcal{M}$ accepts w and $\mathcal{M}'$ likewise accepts and therefore halts, meaning that $\langle \mathcal{M}', w \rangle \in H_{\mathsf{TM}}$. Otherwise, if $\langle \mathcal{M}, w \rangle \notin A_{\mathsf{TM}}$, then $\mathcal{M}$ either rejects w or loops forever, and in either case $\mathcal{M}'$ loops forever, meaning that $\langle \mathcal{M}', w \rangle \notin H_{\mathsf{TM}}$.

If the machine $\mathcal{M}_{\mathrm{H}}$ existed to decide H_{TM}, then we could decide A_{TM} using the machine $\mathcal{M}_{\mathrm{A}}$. However, we know that A_{TM} is undecidable. Thus, $\mathcal{M}_{\mathrm{H}}$ must not exist, and so H_{TM} must be undecidable. ■

In our proof that H_{TM} is undecidable, we constructed a Turing machine $\mathcal{M}_{\mathrm{A}}$ that ostensibly decides A_{TM} and is composed of two parts: a reduction that computes a function f turning instances of A_{TM} into instances of H_{TM}, and a black-box Turing machine $\mathcal{M}_{\mathrm{H}}$ that decides H_{TM}. As before, the Turing machine using this reduction from A_{TM} to H_{TM} is illustrated in Figure 5.3.

We don't know how the black-box Turing machine $\mathcal{M}_{\mathrm{H}}$ decides H_{TM}; we only assume that it exists. However, we *do* know how the reduction works—this is step 1 in our procedure! The function f takes the description of the input Turing machine $\mathcal{M}$ and converts it into a new Turing machine $\mathcal{M}'$ that halts on its input word if and only if $\mathcal{M}$ accepts the same input

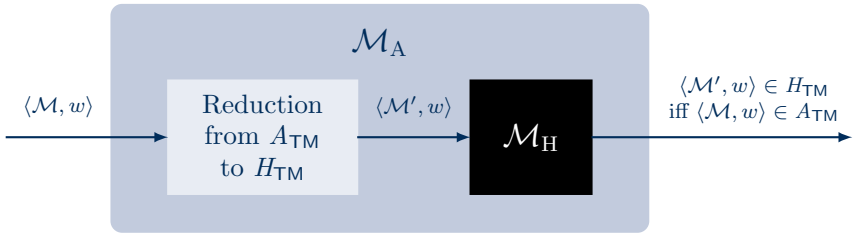


Figure 5.3. A Turing machine for A_{TM} using the reduction $A_{\mathsf{TM}} \leq_{\text{m}} H_{\mathsf{TM}}$.

word.

At this point, you may ask yourself: doesn't this reduction implicitly decide A_{TM}, since it has to figure out whether $\mathcal{M}$ accepts or rejects its input word? This is a reasonable question, since if the reduction did work in this way, we would find ourselves trapped in a snare of circular logic. Fortunately for us, we avoid such a trap, since the reduction does no deciding on its own: it only modifies the description of $\mathcal{M}$. Namely,

- if $\langle \mathcal{M} \rangle$ has a transition leading to q_{accept}, then the reduction leaves this transition as is; and
- if $\langle \mathcal{M} \rangle$ has a transition leading to q_{reject}, then the reduction modifies this transition to instead enter a new "infinite loop" state and render q_{reject} unreachable.

Thus, the reduction only changes how certain transitions of $\mathcal{M}$ behave, and it doesn't consider the output of $\mathcal{M}$ on any particular input word.

Ultimately, our construction establishes that $\mathcal{M}$ accepts w if and only if $\mathcal{M}'$ halts on w (i.e., $\langle \mathcal{M}, w \rangle \in A_{\mathsf{TM}}$ if and only if $\langle \mathcal{M}', w \rangle \in H_{\mathsf{TM}}$). At the same time, $\mathcal{M}$ does not accept w or $\mathcal{M}$ loops forever on w if and only if $\mathcal{M}'$ loops forever on w (i.e., $\langle \mathcal{M}, w \rangle \notin A_{\mathsf{TM}}$ if and only if $\langle \mathcal{M}', w \rangle \notin H_{\mathsf{TM}}$).

You may be wondering whether we needed such a complicated proof to show that the halting problem is undecidable, and strictly speaking, we didn't. In fact, we can return to the code-oriented approach with which we started this section! Strachey [1965] showed that there can exist no function that behaves like our Turing machine $\mathcal{M}_H$ using just a few lines of code. In his approach, Strachey takes $T[R]$ to be a Boolean function receiving an arbitrary program R as input, where R has no free variables as arguments. (In other words, such a program R is "fixed" in the sense that it will always exhibit the same behaviour when run.) The function T behaves in the following way: for all programs R, $T[R]$ returns true if R terminates when run, and $T[R]$ returns false otherwise. Then, Strachey

writes the following program P:

```
rec routine P
    § L : if T[P] go to L
        return §
```

Unless you're familiar with the programming language CPL, this code may look a little obscure to you, but we can look beyond the syntax to tease out the general idea of what the code is doing. As we noted, $T[P]$ checks whether or not P terminates when run, and it returns either true or false. If $T[P]$ returns true, then we will enter the body of the if statement and jump to the line labelled L, which puts us back at the beginning of the if statement; that is, P loops forever. On the other hand, if $T[P]$ returns false, then we will skip over the if statement entirely and return an empty output; that is, P halts. In either case, P behaves in a manner opposite to what $T[P]$ indicated, and so the function T cannot exist!

I have never actually seen a proof of this in print, and though Alan Turing once gave me a verbal proof (in a railway carriage on the way to a Conference at the NPL in 1953), I unfortunately and promptly forgot the details.
— CHRISTOPHER STRACHEY

5.3. MORE UNDECIDABLE PROBLEMS FOR TURING MACHINES

AT THIS POINT, our collection of undecidable problems is slowly growing: we know now that each of A_{TM}, $\overline{A_{\mathsf{TM}}}$, and H_{TM} is undecidable. Using these facts together with the power of reducibility, we can prove many other problems for Turing machines are undecidable.

Emptiness Problem

Let's start by considering the familiar emptiness problem for Turing machines, E_{TM}. In the previous section, we showed that H_{TM} is undecidable by reducing from A_{TM}, since both of these decision problems rely in some way on halting: H_{TM} just focuses on a Turing machine halting on some input word, while for A_{TM}, we know that a Turing machine must necessarily halt in the process of accepting some input word.

The decision problem E_{TM} is different, though: if a Turing machine's language is empty, then it must *not* accept any input words we give to it. Therefore, it doesn't make much sense for us to involve A_{TM} in our proof of undecidability, since this decision problem relies on accepting! Instead, since we're focused on not accepting, let's go with the complementary decision problem $\overline{A_{\mathsf{TM}}}$.

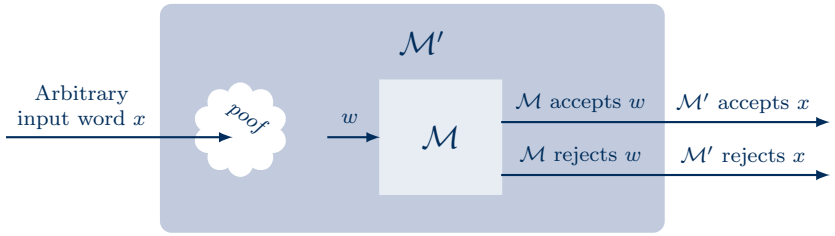


Figure 5.4. The Turing machine $\mathcal{M}'$ from the proof of Theorem 5.10.

Recall that E_{TM} expects to receive a description of a Turing machine as input, while $\overline{A_{\mathsf{TM}}}$ expects to receive as input both a description of a Turing machine and an input word to give to that machine. Our reduction therefore must have the property that, for any Turing machine $\mathcal{M}$ and input word w, $\langle \mathcal{M}, w \rangle \in \overline{A_{\mathsf{TM}}}$ if and only if $\langle \mathcal{M}' \rangle \in E_{\mathsf{TM}}$ for some Turing machine $\mathcal{M}'$.

The big question is: what is $\mathcal{M}'$, and how can we ensure $\mathcal{M}'$ doesn't accept any input words if and only if $\mathcal{M}$ doesn't accept w? Our approach for this proof will be to construct $\mathcal{M}'$ in such a way that, no matter what input word it receives, it completely ignores that word and only checks the behaviour of $\mathcal{M}$ on the specific input word w. Then, whatever answer $\mathcal{M}$ gives for w will become the answer given by $\mathcal{M}'$. (See Figure 5.4 for an illustration of what $\mathcal{M}'$ is doing.) In essence, $\mathcal{M}'$ generalizes the answer given by $\mathcal{M}$: if $\mathcal{M}$ does not accept w, then $\mathcal{M}'$ will accept *nothing*, while if $\mathcal{M}$ accepts w, then $\mathcal{M}'$ will accept *everything*.

At this point, we can start to see how the proof will come together. Our reduction will turn the encoding $\langle \mathcal{M}, w \rangle$ into an encoding $\langle \mathcal{M}' \rangle$, where the behaviour of $\mathcal{M}'$ depends solely on the behaviour of $\mathcal{M}$ given w. Then, supposing we can test the emptiness of the language $L(\mathcal{M}')$, we can also determine whether $w \notin L(\mathcal{M})$.

Theorem 5.10

E_{TM} is undecidable.

Proof. Assume by way of contradiction that E_{TM} is decidable, and suppose that $\mathcal{M}_{\mathrm{E}}$ is a Turing machine that decides E_{TM}.

We construct a new Turing machine $\mathcal{M}_{\overline{\mathrm{A}}}$ for the non-membership problem $\overline{A_{\mathsf{TM}}}$ that uses the reduction $\overline{A_{\mathsf{TM}}} \leq_{\mathrm{m}} E_{\mathsf{TM}}$. The machine $\mathcal{M}_{\overline{\mathrm{A}}}$ takes as input $\langle \mathcal{M}, w \rangle$, where $\mathcal{M}$ is a Turing machine and w is an input word, and performs the following steps:

$\mathcal{M}_{\overline{\text{A}}}$

1. Define a computable function f that takes as input $\langle \mathcal{M}, w \rangle$ and produces as output $\langle \mathcal{M}' \rangle$, where $\mathcal{M}'$ behaves as follows:

   $\mathcal{M}'$

   1. Run $\mathcal{M}$ on the input word w.
   2. If $\mathcal{M}$ accepts w, then accept. If $\mathcal{M}$ rejects w, then reject.

2. Run $\mathcal{M}_{\text{E}}$ on the result $\langle \mathcal{M}' \rangle$ produced by f.
3. If $\mathcal{M}_{\text{E}}$ accepts, then accept. If $\mathcal{M}_{\text{E}}$ rejects, then reject.

Observe that if $\langle \mathcal{M}, w \rangle \in \overline{A_{\mathsf{TM}}}$, then $\mathcal{M}$ does not accept w and $\mathcal{M}'$ will not accept any input word given to it, meaning that $\langle \mathcal{M}' \rangle \in E_{\mathsf{TM}}$. Otherwise, if $\langle \mathcal{M}, w \rangle \notin \overline{A_{\mathsf{TM}}}$, then $\mathcal{M}$ accepts w and $\mathcal{M}'$ will accept every input word given to it, meaning that $\langle \mathcal{M}' \rangle \notin E_{\mathsf{TM}}$.

If the machine $\mathcal{M}_{\text{E}}$ existed to decide E_{TM}, then we could decide $\overline{A_{\mathsf{TM}}}$ using the machine $\mathcal{M}_{\overline{\text{A}}}$. However, we know that $\overline{A_{\mathsf{TM}}}$ is undecidable. Thus, $\mathcal{M}_{\text{E}}$ must not exist, and so E_{TM} must be undecidable. ■

The careful reader might have noticed that the construction we used to prove Theorem 5.10 in fact goes one step further in telling us about the properties of E_{TM}. Since we know that $\overline{A_{\mathsf{TM}}}$ is non-semidecidable by Theorem 4.18, the reduction $\overline{A_{\mathsf{TM}}} \leq_{\text{m}} E_{\mathsf{TM}}$ combined with Corollary 5.7 allows us to establish the following even stronger fact about E_{TM}.

Corollary 5.11

E_{TM} is not semidecidable.

On the other hand, we at least have a glimmer of hope when it comes to handling instances of the Turing machine "non-emptiness" problem $\overline{E_{\mathsf{TM}}}$.

Theorem 5.12

$\overline{E_{\mathsf{TM}}}$ is semidecidable.

Proof. Construct a Turing machine $\mathcal{M}_{\overline{\text{ETM}}}$ that takes as input $\langle \mathcal{M} \rangle$, where $\mathcal{M}$ is a Turing machine, and performs the following steps:

$\mathcal{M}_{\overline{\text{ETM}}}$

1. Enumerate all words over Σ^*, producing a list $\{w_1, w_2, w_3, \dots\}$.
2. For all $i \in \{1, 2, 3, \dots\}$, simulate i steps of the computation of $\mathcal{M}$ on the first i words w_1 to w_i.
3. If $\mathcal{M}$ ever enters its accepting state q_{accept} on any of the words, then accept.

Observe first that since Σ^* is a countably infinite union of finite sets, it is itself countably infinite, and so we can perform the enumeration specified in Step 1 of $\mathcal{M}_{\overline{\text{ETM}}}$.

This Turing machine accepts its input $\langle \mathcal{M} \rangle$ if and only if the Turing machine $\mathcal{M}$ accepts at least one input word w_j where $j \in \{1, 2, 3, \dots\}$, and if no word is accepted by $\mathcal{M}$, then the Turing machine loops forever. Thus, $\mathcal{M}_{\overline{\text{ETM}}}$ semidecides the non-emptiness problem for Turing machines. ■

As a consequence of Theorem 5.12, we can also say that E_{TM} is co-semidecidable.

Unfortunately, semidecidability is all that we can hope for with $\overline{E_{\mathsf{TM}}}$: since reductions are closed under complement by Lemma 5.3, knowing that $\overline{A_{\mathsf{TM}}} \leq_{\text{m}} E_{\mathsf{TM}}$ also tells us that $A_{\mathsf{TM}} \leq_{\text{m}} \overline{E_{\mathsf{TM}}}$, and since A_{TM} is undecidable, we conclude that $\overline{E_{\mathsf{TM}}}$ must also be undecidable.

Before we continue, let's revisit our choice to use the decision problem $\overline{A_{\mathsf{TM}}}$ in our reduction showing that E_{TM} is undecidable. Even though we saw an earlier justification for why it didn't make much sense for us to use A_{TM} in this situation, would it nevertheless have been possible for us to reduce A_{TM} to E_{TM} as an alternative proof? Interestingly, no! Using many-one reductions, there is in fact no way for us to reduce A_{TM} to E_{TM}, and the proof follows directly from what we already know about both problems.

Theorem 5.13

$A_{\mathsf{TM}} \not\leq_{\text{m}} E_{\mathsf{TM}}$.

Proof. Assume by way of contradiction that $A_{\mathsf{TM}} \leq_{\text{m}} E_{\mathsf{TM}}$, and let f be the computable function performing such a reduction. Since reductions are closed under complement by Lemma 5.3, we also have that $\overline{A_{\mathsf{TM}}} \leq_{\text{m}} \overline{E_{\mathsf{TM}}}$ by the same computable function f.

We know by Theorem 5.12 that $\overline{E_{\mathsf{TM}}}$ is semidecidable, so by Theorem 5.6, this implies that $\overline{A_{\mathsf{TM}}}$ is also semidecidable. However, this contradicts Theorem 4.18, so no such reduction can exist. ■

Theorem 5.13 raises a fascinating point about many-one reductions: sometimes, a many-one reduction between two decision problems simply cannot exist! This, in turn, highlights the importance of creatively constructing our reductions: if we wanted to prove directly that E_{TM} was undecidable, and we focused on using the classic undecidable problem A_{TM} in our proof, then we would've quickly hit a brick wall. Introducing the complementary problems $\overline{E_{\mathsf{TM}}}$ and $\overline{A_{\mathsf{TM}}}$ to our suite of undecidable problems gives us more flexibility.

> *Remark.* Confusingly, some textbooks and learning materials purport to give a "reduction" from A_{TM} to E_{TM} in their proof of the undecidability of E_{TM}. These proofs typically use an informal notion of reducibility that disguises the fact that the reduction is actually from A_{TM} to $\overline{E_{\mathsf{TM}}}$.

Universality Problem

Let's now move on to the universality problem for Turing machines, U_{TM}. Since this decision problem asks whether the language of a Turing machine contains all words, one might think (as we did with finite automata) that U_{TM} is just the opposite of E_{TM}, and so we can immediately draw a connection to our complementary decision problem $\overline{E_{\mathsf{TM}}}$. Unfortunately, things aren't so easy when it comes to Turing machines, as the class of semidecidable languages isn't closed under complement! A positive answer to an instance of $\overline{E_{\mathsf{TM}}}$ only tells us that the Turing machine in question accepts *at least one* word, not that it accepts *all* words.

This doesn't mean we need to start from scratch, though: we can in fact reuse some of the ideas we had in showing that E_{TM} is undecidable to prove that U_{TM} is undecidable. If a Turing machine's language is universal, then it must accept *every* word we give to it. This statement sounds very similar to A_{TM}, which asks whether a Turing machine accepts whatever specific word we gave to it. Thus, just like we reduced $\overline{A_{\mathsf{TM}}}$ to E_{TM} in the previous section, it seems promising here for us to reduce A_{TM} to U_{TM}.

Recall that U_{TM} expects to receive a description of a Turing machine as input, while A_{TM} expects to receive as input both a description of a Turing machine and an input word to give to that machine. Thus, our reduction must have the property that, for any Turing machine $\mathcal{M}$ and input word w, $\langle \mathcal{M}, w \rangle \in A_{\mathsf{TM}}$ if and only if $\langle \mathcal{M}' \rangle \in U_{\mathsf{TM}}$ for some Turing machine $\mathcal{M}'$. As before, we must construct a Turing machine $\mathcal{M}'$ that accepts all input words if and only if $\mathcal{M}$ accepts w.

It's no coincidence that we denoted this Turing machine by $\mathcal{M}'$ again: it turns out that we can use the exact same approach that we used with the Turing machine emptiness problem! Again, we will ensure that no matter what input word $\mathcal{M}'$ receives, it will ignore that word and instead simulate the computation of $\mathcal{M}$ on its input word w. Then, whatever answer $\mathcal{M}$ gives

for w will become the answer given by $\mathcal{M}'$. We're following the same idea by having $\mathcal{M}'$ generalize the answer given by $\mathcal{M}$: if $\mathcal{M}$ accepts w, then $\mathcal{M}'$ will accept *everything*, while if $\mathcal{M}$ rejects w, then $\mathcal{M}'$ will accept *nothing*.

Our reduction will therefore behave identically to our previous reduction for E_{TM}: we will turn the encoding $\langle \mathcal{M}, w \rangle$ into an encoding $\langle \mathcal{M}' \rangle$, where the behaviour of $\mathcal{M}'$ depends solely on the behaviour of $\mathcal{M}$ given w. Then, supposing we can test the universality of the language $L(\mathcal{M}')$, we can also determine whether $w \in L(\mathcal{M})$.

Theorem 5.14

U_{TM} is undecidable.

Proof. Assume by way of contradiction that U_{TM} is decidable, and suppose that $\mathcal{M}_{\mathrm{U}}$ is a Turing machine that decides U_{TM}.

We construct a new Turing machine $\mathcal{M}_{\mathrm{A}}$ for the membership problem A_{TM} that uses the reduction $A_{\mathsf{TM}} \leq_{\mathrm{m}} U_{\mathsf{TM}}$. The machine $\mathcal{M}_{\mathrm{A}}$ takes as input $\langle \mathcal{M}, w \rangle$, where $\mathcal{M}$ is a Turing machine and w is an input word, and performs the following steps:

$\mathcal{M}_{\mathrm{A}}$

1. Define a computable function f that takes as input $\langle \mathcal{M}, w \rangle$ and produces as output $\langle \mathcal{M}' \rangle$, where $\mathcal{M}'$ behaves as follows:

   $\mathcal{M}'$

   1. Run $\mathcal{M}$ on the input word w.
   2. If $\mathcal{M}$ accepts w, then accept. If $\mathcal{M}$ rejects w, then reject.

2. Run $\mathcal{M}_{\mathrm{U}}$ on the result $\langle \mathcal{M}' \rangle$ produced by f.
3. If $\mathcal{M}_{\mathrm{U}}$ accepts, then accept. If $\mathcal{M}_{\mathrm{U}}$ rejects, then reject.

Observe that if $\langle \mathcal{M}, w \rangle \in A_{\mathsf{TM}}$, then $\mathcal{M}$ accepts w and $\mathcal{M}'$ will accept every input word given to it, meaning that $\langle \mathcal{M}' \rangle \in U_{\mathsf{TM}}$. Otherwise, if $\langle \mathcal{M}, w \rangle \notin A_{\mathsf{TM}}$, then $\mathcal{M}$ does not accept w and $\mathcal{M}'$ will not accept any input word given to it, meaning that $\langle \mathcal{M}' \rangle \notin U_{\mathsf{TM}}$.

If the machine $\mathcal{M}_{\mathrm{U}}$ existed to decide U_{TM}, then we could decide A_{TM} using the machine $\mathcal{M}_{\mathrm{A}}$. However, we know that A_{TM} is undecidable. Thus, $\mathcal{M}_{\mathrm{U}}$ must not exist, and so U_{TM} must be undecidable. ■

Now, even if we have a negative decidability result, we know that A_{TM} is semidecidable, so does the reduction $A_{\mathsf{TM}} \leq_{\mathrm{m}} U_{\mathsf{TM}}$ give us any clue as to the status of semidecidability for U_{TM}? No—and be careful to remember

the wording of Theorem 5.6! If we have a reduction $X \leq_m Y$ and we know that Y is semidecidable, then we can conclude that X is semidecidable, but we can't conclude anything about Y in the case where X is semidecidable.

To tackle the question of semidecidability for U_{TM}, let us introduce another decision problem known as the *totality problem* for Turing machines:

$$T_{\mathsf{TM}} = \{\langle \mathcal{M} \rangle \mid \mathcal{M} \text{ is a Turing machine that halts on all input words}\}.$$

By analogy, if H_{TM} is the "halting only" version of A_{TM}, then T_{TM} is the "halting only" version of U_{TM}. Naturally, since both H_{TM} and T_{TM} talk about halting, we can reduce one to the other in a rather straightforward way, and our reduction again uses our tried-and-true Turing machine $\mathcal{M}'$.

Lemma 5.15

$H_{\mathsf{TM}} \leq_m T_{\mathsf{TM}}$.

Proof. Define a computable function f that takes as input $\langle \mathcal{M}, w \rangle$ and produces as output $\langle \mathcal{M}' \rangle$, where $\mathcal{M}'$ behaves as follows:

$\mathcal{M}'$

1. Run $\mathcal{M}$ on the input word w.
2. If $\mathcal{M}$ accepts w, then accept. If $\mathcal{M}$ rejects w, then reject.

Observe that if $\langle \mathcal{M}, w \rangle \in H_{\mathsf{TM}}$, then $\mathcal{M}$ halts on w and $\mathcal{M}'$ will halt on any input word given to it, meaning that $\langle \mathcal{M}' \rangle \in T_{\mathsf{TM}}$. Otherwise, if $\langle \mathcal{M}, w \rangle \notin H_{\mathsf{TM}}$, then $\mathcal{M}$ does not halt on w and $\mathcal{M}'$ will likewise loop forever, meaning that $\langle \mathcal{M}' \rangle \notin T_{\mathsf{TM}}$. ■

Since we know that reductions are closed under complement, Lemma 5.15 tells us that $\overline{H_{\mathsf{TM}}} \leq_m \overline{T_{\mathsf{TM}}}$ as well. At the same time, we can also reduce the complement of the halting problem, $\overline{H_{\mathsf{TM}}}$, directly to T_{TM}. Of course, since we're attempting to test *non*-halting with this reduction, we must be careful not to fall into the trap of looping forever by bounding the length of our Turing machine's computation.

Lemma 5.16

$\overline{H_{\mathsf{TM}}} \leq_m T_{\mathsf{TM}}$.

Proof. Define a computable function f that takes as input $\langle \mathcal{M}, w \rangle$ and produces as output $\langle \mathcal{M}'' \rangle$, where $\mathcal{M}''$ behaves as follows:

$\mathcal{M}''$

1. Run $\mathcal{M}$ on the input word w for $|x|$ computation steps, where $|x|$ is the length of the input word x given to $\mathcal{M}''$.
2. If $\mathcal{M}$ halts on w within those $|x|$ computation steps, then loop forever. Otherwise, accept.

Observe that if $\langle \mathcal{M}, w \rangle \in \overline{H_{\mathsf{TM}}}$, then $\mathcal{M}$ will not halt on w and $\mathcal{M}''$ will halt on any input word given to it, meaning that $\langle \mathcal{M}'' \rangle \in T_{\mathsf{TM}}$. Otherwise, if $\langle \mathcal{M}, w \rangle \notin \overline{H_{\mathsf{TM}}}$, then $\mathcal{M}$ will halt on w in some number of computation steps s and $\mathcal{M}''$ will not halt on any input word of length at most s, meaning that $\langle \mathcal{M}'' \rangle \notin T_{\mathsf{TM}}$. ■

You might be wondering at this point why we've introduced the complement of the halting problem, $\overline{H_{\mathsf{TM}}}$, and what this has to do with the totality problem for Turing machines. We haven't forgotten about our question of semidecidability for U_{TM}; all of this is just building up to the answer.

Recall from Theorem 5.9 that there exists a reduction $A_{\mathsf{TM}} \leq_{\mathrm{m}} H_{\mathsf{TM}}$. Since reductions are (as we well know by now) closed under complement, this means that there also exists a reduction $\overline{A_{\mathsf{TM}}} \leq_{\mathrm{m}} \overline{H_{\mathsf{TM}}}$. But Theorem 4.18 tells us that $\overline{A_{\mathsf{TM}}}$ is not semidecidable, and therefore $\overline{H_{\mathsf{TM}}}$ must not be semidecidable either. Now, everything begins to fall into place: since $\overline{H_{\mathsf{TM}}} \leq_{\mathrm{m}} \overline{T_{\mathsf{TM}}}$ and $\overline{H_{\mathsf{TM}}} \leq_{\mathrm{m}} T_{\mathsf{TM}}$, it must be the case that neither T_{TM} nor $\overline{T_{\mathsf{TM}}}$ is semidecidable.

Completing this line of reasoning by connecting T_{TM} to U_{TM} gives us the final remarkable result: U_{TM} is not just undecidable, but it is also neither semidecidable nor co-semidecidable, and so it falls completely outside of our language hierarchy!

Theorem 5.17

U_{TM} is neither semidecidable nor co-semidecidable.

Proof. We begin by demonstrating the existence of a reduction $T_{\mathsf{TM}} \leq_{\mathrm{m}} U_{\mathsf{TM}}$. Define a computable function f that takes as input $\langle \mathcal{M} \rangle$ and produces as output $\langle \mathcal{N} \rangle$, where $\mathcal{N}$ behaves as follows:

$\mathcal{N}$

1. Behave exactly as $\mathcal{M}$ behaves, except redirect all transitions to q_{reject} to instead go to q_{accept}.

Observe that if $\langle \mathcal{M} \rangle \in T_{\mathsf{TM}}$, then $\mathcal{M}$ halts on every input word

and either accepts or rejects. Thus, $\mathcal{N}$ will also halt on every input word, but it will always accept, meaning that $\langle \mathcal{N} \rangle \in U_{\mathsf{TM}}$. Otherwise, if $\langle \mathcal{M} \rangle \notin T_{\mathsf{TM}}$, then $\mathcal{M}$ must not halt on at least one input word, and so $\mathcal{N}$ will likewise never accept that input word, meaning that $\langle \mathcal{N} \rangle \notin U_{\mathsf{TM}}$.

From the reduction $T_{\mathsf{TM}} \leq_{\mathrm{m}} U_{\mathsf{TM}}$, we know also that $\overline{T_{\mathsf{TM}}} \leq_{\mathrm{m}} \overline{U_{\mathsf{TM}}}$. However, since neither T_{TM} nor $\overline{T_{\mathsf{TM}}}$ is semidecidable, it must be the case that neither U_{TM} nor $\overline{U_{\mathsf{TM}}}$ is semidecidable. Saying that $\overline{U_{\mathsf{TM}}}$ is not semidecidable is equivalent to saying that U_{TM} is not co-semidecidable. ■

Equivalence Problem

Recall that the equivalence problem for Turing machines, EQ_{TM}, asks whether the languages of two Turing machines are equivalent; that is, whether no word belongs to one language but not the other.

As we might reasonably expect by this point, EQ_{TM} is an undecidable problem, just like all of the other problems we've studied thus far for Turing machines. But in each of our previous undecidability proofs, the underlying decision problems focused only on a single Turing machine: A_{TM} and H_{TM} both took as input an encoding of the form $\langle \mathcal{M}, w \rangle$, while each of E_{TM}, U_{TM}, and T_{TM} took as input an encoding of the form $\langle \mathcal{M} \rangle$. The problem EQ_{TM}, by contrast, takes as input $\langle \mathcal{M}, \mathcal{N} \rangle$, where both $\mathcal{M}$ and $\mathcal{N}$ are Turing machines.

How, then, can we prove that EQ_{TM} is undecidable via some many-one reduction? We just need to use a little cleverness in our construction. Take, for example, the problem U_{TM}. Going back to the definition of this problem, we have that $\langle \mathcal{M} \rangle \in U_{\mathsf{TM}}$ if and only if $L(\mathcal{M}) = \Sigma^*$. Look at what we just used in that definition: an equals sign! If we construct a new Turing machine specially designed to accept all input words, then we can compare the language of our original Turing machine $\mathcal{M}$ to the language of this new Turing machine, and therein lies the foundation for our reduction.

Theorem 5.18

EQ_{TM} is undecidable.

Proof. Assume by way of contradiction that EQ_{TM} is decidable, and suppose that $\mathcal{M}_{\mathrm{EQ}}$ is a Turing machine that decides EQ_{TM}.

We construct a new Turing machine $\mathcal{M}_{\mathrm{U}}$ for the universality problem U_{TM} that uses the reduction $U_{\mathsf{TM}} \leq_{\mathrm{m}} EQ_{\mathsf{TM}}$. The machine $\mathcal{M}_{\mathrm{U}}$ takes as input $\langle \mathcal{M} \rangle$, where $\mathcal{M}$ is a Turing machine, and performs the following steps:

$\mathcal{M}_{\text{U}}$

1. Define a computable function f that takes as input $\langle \mathcal{M} \rangle$ and produces as output $\langle \mathcal{M}, \mathcal{M}_{\Sigma^*} \rangle$, where $\mathcal{M}_{\Sigma^*}$ behaves as follows:

   $\mathcal{M}_{\Sigma^*}$

   1. Accept.

2. Run $\mathcal{M}_{\text{EQ}}$ on the result $\langle \mathcal{M}, \mathcal{M}_{\Sigma^*} \rangle$ produced by f.
3. If $\mathcal{M}_{\text{EQ}}$ accepts, then accept. If $\mathcal{M}_{\text{EQ}}$ rejects, then reject.

Observe that if $\langle \mathcal{M} \rangle \in U_{\mathsf{TM}}$, then $\mathcal{M}$ accepts every input word given to it. Therefore, $L(\mathcal{M})$ is equal to $L(\mathcal{M}_{\Sigma^*})$, meaning that $\langle \mathcal{M}, \mathcal{M}_{\Sigma^*} \rangle \in EQ_{\mathsf{TM}}$. Otherwise, if $\langle \mathcal{M} \rangle \notin U_{\mathsf{TM}}$, then $\mathcal{M}$ does not accept at least one input word, and therefore its language cannot be equal to $L(\mathcal{M}_{\Sigma^*})$, meaning that $\langle \mathcal{M}, \mathcal{M}_{\Sigma^*} \rangle \notin EQ_{\mathsf{TM}}$.

If the machine $\mathcal{M}_{\text{EQ}}$ existed to decide EQ_{TM}, then we could decide U_{TM} using the machine $\mathcal{M}_{\text{U}}$. However, we know that U_{TM} is undecidable. Thus, $\mathcal{M}_{\text{EQ}}$ must not exist, and so EQ_{TM} must be undecidable. ■

Keen readers might have noticed that we could alternatively prove Theorem 5.18 via the reduction $E_{\mathsf{TM}} \leq_{\text{m}} EQ_{\mathsf{TM}}$. In this case, the computable function f would construct a Turing machine $\mathcal{M}_{\emptyset}$ that rejects all input words, but the rest of the proof would be otherwise identical.

There's a good reason why we chose to reduce from U_{TM}, though: this reduction provides an immediate proof that the equivalence problem for Turing machines is neither semidecidable nor co-semidecidable, giving us a second decision problem that falls completely outside of our language hierarchy.

Theorem 5.19

EQ_{TM} is neither semidecidable nor co-semidecidable.

Proof. We demonstrated the existence of a reduction $U_{\mathsf{TM}} \leq_{\text{m}} EQ_{\mathsf{TM}}$ in the proof of Theorem 5.18, and since U_{TM} is not semidecidable by Theorem 5.17, EQ_{TM} is likewise not semidecidable.

From the reduction $U_{\mathsf{TM}} \leq_{\text{m}} EQ_{\mathsf{TM}}$, we know also that $\overline{U_{\mathsf{TM}}} \leq_{\text{m}} \overline{EQ_{\mathsf{TM}}}$, and since $\overline{U_{\mathsf{TM}}}$ is not semidecidable by Theorem 5.17, $\overline{EQ_{\mathsf{TM}}}$ is likewise not semidecidable. Saying that $\overline{EQ_{\mathsf{TM}}}$ is not semidecidable is equivalent to saying that EQ_{TM} is not co-semidecidable. ■

Inclusion Problem

As we might expect, the inclusion problem is also undecidable for Turing machines, and this is a fact that will soon be proven in this section.

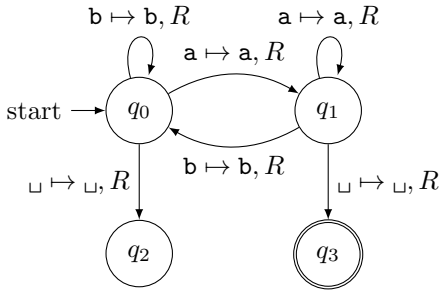
5.4. REDUCING FROM TURING MACHINE COMPUTATIONS

At this point, we're nearly done with our exploration of decision problems applied to our common models of computation. We've seen that all of the common decision problems about finite automata and other regular models are decidable, while all of the common decision problems about Turing machines are undecidable—and, in some cases, not even semidecidable. However, there remain three omissions from our landscape of decidability results: we haven't yet taken a closer look at the universality, equivalence, or inclusion problems for context-free languages.

As we will see, the class of context-free languages is interesting in that it presents a sort-of middle ground where not every problem is decidable, but where we can still decide certain problems. We observed this in the previous chapter by proving that both A_{CFG} and E_{CFG} (as well as their pushdown automaton analogues) are decidable. Unlike the class of regular languages, where the decidability of U_{DFA} followed via a combination of the decidability of E_{DFA} and closure of the regular languages under complement, the question of decidability for U_{CFG} isn't as straightforward. This is primarily because, as we know, the class of context-free languages isn't closed under complement. This may suggest to us that U_{CFG} is undecidable, but at the same time, how can we prove the undecidability of U_{CFG} if the only tool we have at our disposal is a many-one reduction from an undecidable problem for a completely different model?

While pondering this same question, Juris Hartmanis made a fascinating discovery that relates context-free languages to Turing machines [1967]. If we take all of the configurations of a Turing machine as it processes some input word, we can combine these configurations into something called a *computation history*, which is effectively a complete record of everything the Turing machine did over the course of its computation. Hartmanis' key discovery was that the computation history of any Turing machine is itself a context-free language!

What this means for us is that, if we wish to prove that a decision problem for context-free languages is undecidable, we no longer need to rely on coming up with a much-too-powerful many-one reduction from an undecidable problem for Turing machines. Instead, we can reframe the undecidable Turing machine problem in terms of the computation histories of some Turing machine, and if we could construct a context-free grammar that generates these computation histories, then that grammar would allow



(a) A Turing machine $\mathcal{M}$.

$C_1 : q_0\,\mathtt{a}\,\mathtt{b}\,\mathtt{a}\,\mathtt{a}\,\sqcup$
$C_2 : \mathtt{a}\,q_1\,\mathtt{b}\,\mathtt{a}\,\mathtt{a}\,\sqcup$
$C_3 : \mathtt{a}\,\mathtt{b}\,q_0\,\mathtt{a}\,\mathtt{a}\,\sqcup$
$C_4 : \mathtt{a}\,\mathtt{b}\,\mathtt{a}\,q_1\,\mathtt{a}\,\sqcup$
$C_5 : \mathtt{a}\,\mathtt{b}\,\mathtt{a}\,\mathtt{a}\,q_1\,\sqcup$
$C_6 : \mathtt{a}\,\mathtt{b}\,\mathtt{a}\,\mathtt{a}\,\sqcup\,q_3$

(b) The computation history of $\mathcal{M}$ on $w = \mathtt{abaa}$.

$$q_0\mathtt{abaa}\sqcup\#\sqcup\mathtt{aab}q_1\mathtt{a}\#\mathtt{ab}q_0\mathtt{aa}\sqcup\#\sqcup\mathtt{a}q_1\mathtt{aba}\#\mathtt{abaa}q_1\sqcup\#q_3\sqcup\mathtt{aaba}$$

(c) The computation history of $\mathcal{M}$ on $w = \mathtt{abaa}$, represented as a string. Observe that every second configuration is reversed.

Figure 5.5. An example of a Turing machine and its computation history on an input word.

us to answer the undecidable problem. Essentially, we're using a more informal notion of a "reduction" from Turing machine computation histories to context-free grammars.

Before we continue along this line of thought, let's reacquaint ourselves with some Turing machine definitions. Recall from Section 3.1.1 that we took the configuration of a Turing machine to be a representation of the current state, tape contents, and input head position of that Turing machine at some point in its computation. In other terms, a configuration is a "snapshot" of the Turing machine mid-computation. Depending on the current state of the machine, a configuration may be a start configuration if the current state is q_0, an accepting configuration if the current state is q_{accept}, or a rejecting configuration if the current state is q_{reject}.

Taking a sequence of configurations together, we get the aforementioned computation history, which we may represent either as a set of individual configurations or as a single string of concatenated configurations; examples of each are depicted in Figure 5.5.

Definition 5.20 (Computation history)

Given a Turing machine $\mathcal{M}$ and an input word w, a computation history for $\mathcal{M}$ on w is a string of the form

$$C_1 \,\#\, (C_2)^{\mathrm{R}} \,\#\, C_3 \,\#\, (C_4)^{\mathrm{R}} \,\#\, \ldots,$$

where $\# \notin \Gamma$ is a special boundary marker, C_i is the ith configuration of

$\mathcal{M}$, and $(C_i)^{\mathrm{R}}$ denotes the reversal of configuration C_i.

Note that a computation history for a Turing machine $\mathcal{M}$ only exists when $\mathcal{M}$ halts on its input word w. As a result, the sequence of configurations $C_1, C_2, \ldots, C_n$ always has a finite number of elements. Deterministic computations have exactly one computation history per input word, while nondeterministic computations may have multiple computation histories for the same input word.

Just like how code may or may not contain syntax errors, we can have either *valid* or *invalid* computation histories depending on the configurations themselves and the order in which the configurations are sequenced. A valid computation history is one whose configurations satisfy four criteria.

Definition 5.21 (Valid computation history)

Given a Turing machine $\mathcal{M}$ and an input word w, a valid computation history for $\mathcal{M}$ on w is a computation history where all of the following are true:

1. For all $1 \leq i \leq n$, each configuration C_i is of the form $\Gamma^* q \Gamma^*$, where $q \in Q$ is a state of $\mathcal{M}$;
2. The configuration C_1 is a start configuration of the form $q_0 w$, where q_0 is the initial state of $\mathcal{M}$ and $w \in \Sigma^*$;
3. The configuration C_n is either an accepting configuration or a rejecting configuration of the form $\Gamma^* q_f \Gamma^*$, where $q_f \in \{q_{\text{accept}}, q_{\text{reject}}\}$; and
4. For all $1 \leq i \leq (n-1)$, $C_i \vdash C_{i+1}$.

Naturally, then, an invalid computation history belongs to the complement of the language of valid computation histories; to be precise, an invalid computation history is one whose configurations *do not* satisfy at least one of the four criteria provided in Definition 5.21.

Definition 5.22 (Invalid computation history)

Given a Turing machine $\mathcal{M}$ and an input word w, an invalid computation history for $\mathcal{M}$ on w is a computation history where at least one of the following is true:

1. For some $1 \leq i \leq n$, configuration C_i is not of the form $\Gamma^* q \Gamma^*$, where $q \in Q$ is a state of $\mathcal{M}$; or

2. The configuration C_1 is not a start configuration; or
3. The configuration C_n is neither an accepting configuration nor a rejecting configuration; or
4. For some $1 \leq i \leq (n-1)$, $C_i \nvdash C_{i+1}$.

We put a particular emphasis on invalid computation histories here since they will be the key to obtaining our desired undecidability results for context-free languages. Indeed, to draw the same connection between Turing machines and context-free languages as Hartmanis did, we can prove that each of the four violated Turing machine configuration conditions is recognized by some context-free model of computation.

Lemma 5.23

Given a Turing machine $\mathcal{M}$ and an input word w, the set of invalid computation histories of $\mathcal{M}$ on w is a context-free language.

Proof. We will prove this statement by showing that each of the four individual violated criteria corresponds to some context-free language.

1. Where some configuration C_i is not of the form $\Gamma^* q \Gamma^*$, we can construct a finite automaton that checks whether a nondeterministically selected configuration does not consist of a (possibly empty) prefix of symbols from Γ, followed by an encoding of a state in $\mathcal{M}$, and finally a (possibly empty) suffix of symbols from Γ. Since all regular languages are context-free, the set of configurations violating the first criterion is context-free.
2. Where the configuration C_1 is not a start configuration, we can again construct a finite automaton that checks whether this configuration does not consist of the encoding of the state q_0 followed by a (possibly empty) suffix of symbols from Σ. For the same reason as before, the set of configurations violating the second criterion is regular, and therefore context-free.
3. Where the configuration C_n is neither an accepting nor a rejecting configuration, we can once more construct a finite automaton that functions much like in the first case, except it checks specifically for encodings of either of the states q_{accept} or q_{reject}. Thus, the set of configurations violating the third criterion is regular, and therefore context-free.

4. To check whether $C_i \not\vdash C_{i+1}$ for some i, due to the way the computation history string is formatted, we must consider two possible subcases based on whether configuration C_i is odd-indexed or even-indexed:

   (a) Where $C_i \not\vdash C_{i+1}$ for some odd i, we can construct a pushdown automaton that receives as input both the encoding of $\mathcal{M}$ as well as the computation history of $\mathcal{M}$ on w. This pushdown automaton then nondeterministically reads an even number of boundary markers # from the computation history before beginning to read the configuration C_i. As the pushdown automaton reads C_i, it consults the encoding of $\mathcal{M}$ to determine some configuration C' such that $C_i \vdash C'$, and pushes the symbols of C' to its stack. Once the pushdown automaton reads another boundary marker # from the computation history, it compares the configuration $(C_{i+1})^{\mathrm{R}}$ to C' symbol-by-symbol, and if there is any mismatch in symbols, it accepts.

   (b) Where $C_i \not\vdash C_{i+1}$ for some even i, we use essentially the same pushdown automaton construction as in the first subcase, making the appropriate modifications to handle $(C_i)^{\mathrm{R}}$.

   In either case, as a consequence of the existence of these pushdown automata, the set of configurations violating the fourth criterion is context-free.

Since the class of context-free languages is closed under union, taking the union of these individual context-free languages gives us our result. ■

5.5. UNDECIDABLE PROBLEMS FOR CONTEXT-FREE LANGUAGES

Having established what it means for a computation history to be invalid, and having showed that there exists a connection between the invalid computation histories of a Turing machine and our context-free models of computation, we have all we need to establish the undecidability of our remaining decision problems for context-free languages.

Universality Problem

When we established the decidability of the universality problem for regular languages, we relied on Theorem 1.30, which showed that the class of regular languages was closed under complement. Taking the complement of a given finite automaton's language allowed us to use our existing decision procedure

for E_{DFA} to decide U_{DFA}, as shown in Theorem 4.5.

For context-free languages, we know by Theorem 4.11 that E_{CFG} is decidable, so one may be tempted to try a similar approach to establish the supposed decidability of U_{CFG}. However, Theorem 2.25 tells us that the class of context-free languages is *not* closed under complement, and so this approach is a non-starter.

Instead, using a straightforward argument that relies on the connection between invalid computation histories and context-free languages, we can show that U_{CFG} is in fact *undecidable*—the first such result we have for a model weaker than Turing machines!

Theorem 5.24

U_{CFG} is undecidable.

Proof. Suppose that $\mathcal{M}$ is an arbitrary Turing machine. By Lemma 5.23, we can construct a context-free grammar G having the property that $L(G) = \Sigma^*$ if and only if $L(\mathcal{M}) = \emptyset$. This is because if $\mathcal{M}$ accepts no input words, then every input word given to $\mathcal{M}$ would produce an invalid computation history, and the aforementioned lemma tells us that the set of invalid computation histories of $\mathcal{M}$ is a context-free language.

However, if it were possible for us to check whether $L(G) = \Sigma^*$, then it would also become possible for us to decide whether $L(\mathcal{M}) = \emptyset$, and we know by Theorem 5.10 that E_{TM} is undecidable. Therefore, U_{CFG} must also be undecidable. ■

At this point, take a moment to reflect on the argument we just used to establish the undecidability of U_{CFG}. Unlike our previous undecidability proofs using many-one reductions, we didn't need to construct any Turing machines here. Rather, we simply relied on a property that is true of any Turing machine: the set of invalid computation histories is a context-free language, and so we can construct a grammar for this language.

Note that nowhere did we say the language of our grammar G *is* Σ^*; all we said is that $L(G) = \Sigma^*$ *if and only if* $L(\mathcal{M}) = \emptyset$. The language of G depends on whatever arbitrary Turing machine $\mathcal{M}$ we are given, but if we had a method of deciding whether the language of G was universal, then that same method could decide whether the language of this Turing machine $\mathcal{M}$ is empty.

Naturally, since context-free grammars and pushdown automata are equivalent, this negative decidability result transfers over to our other context-free model of computation.

Corollary 5.25

U_{PDA} is undecidable.

Equivalence Problem

Recall that we can test the equivalence of two languages L_1 and L_2 by testing the emptiness of their symmetric difference, where the symmetric difference operation is defined in terms of union, intersection, and complement:

$$L_1 \triangle L_2 = \left(L_1 \cap \overline{L_2}\right) \cup \left(\overline{L_1} \cap L_2\right).$$

Now, again, we know by Theorem 4.11 that we can test the emptiness of a context-free grammar, so we can test emptiness for two individual, separate context-free grammars just as easily. However, the class of context-free languages is not closed under intersection (by Theorem 2.24), nor is it closed under complement (by Theorem 2.25). Thus, we have no way of testing the emptiness of the symmetric difference of two context-free grammars, and so it is hopeless for us to expect that we can decide the equivalence problem for context-free grammars.

But how do we formally prove the undecidability of EQ_{CFG}? We need only fix one of our two context-free grammars to be "special", which will then allow us to use what we know about U_{CFG} to arrive at our desired outcome.

Theorem 5.26

EQ_{CFG} is undecidable.

Proof. Let G_1 be an arbitrary context-free grammar, and let G_2 be a context-free grammar that generates all words over its alphabet Σ. If it were possible for us to check whether $L(G_1) = L(G_2)$, then it would also become possible for us to decide whether $L(G_1) = \Sigma^*$, and we know by Theorem 5.24 that U_{CFG} is undecidable. Therefore, EQ_{CFG} must also be undecidable. ■

In this proof, we have reduced in the sense that we made an instance of the decision problem EQ_{CFG} appear like an instance of the decision problem U_{CFG}, and since the latter problem is undecidable, so too must the former problem be undecidable.

As we would expect, the same outcome holds for pushdown automata.

Corollary 5.27

EQ_{PDA} is undecidable.

Inclusion Problem

Yes, the inclusion problem is also undecidable for context-free languages. This fact will be proved in due time.

5.6. POST'S CORRESPONDENCE PROBLEM ◇

I plan to write a short section on Post's correspondence problem [1946], its undecidability, and its applicability to showing other problems are undecidable.

5.7. RICE'S THEOREM ◇

To close this chapter, I'll discuss Rice's theorem [1953]: a generalization of the halting problem that reveals every nontrivial semantic property of Turing machines is undecidable! Hopefully, by this point, the reader's hope will not have been totally wiped out before getting to later chapters.

CHAPTER NOTES

The body of work pertaining to undecidability is vast, and the papers therein are often dense with mathematical and logical notation. Davis [1965] collects many of the major papers in his anthology and provides brief commentaries for each paper.

5.1. Many-one reductions were first studied by Post [1944]. As we noted, the name "many-one" reflects the fact that a reduction is a function, but Post chose this particular name to set this notion of reducibility apart from his more restrictive notion of *one-one reducibility*, where the function is constrained to be injective. Shapiro [1956] later studied ideas similar to those of Post, but referred to many-one reducibility as *strong reducibility*.

Other authors refer to many-one reductions under a different name; for instance, Sipser [2013] calls them *mapping reductions*.

The result stated in Lemma 5.2 follows more or less directly from the definition of a many-one reduction, but Shapiro [1956] explicitly lists these same properties in his paper.

5.2. The Goldbach conjecture was first formulated in a letter from Christian Goldbach to Leonhard Euler written in 1742. In our discussion, we noted that the conjecture was verified for all even numbers up to

4×10^{18}. This verification was performed by Oliveira e Silva, Herzog, and Pardi [2014].

The earliest mention of a decision problem that asks about "a machine which [...] eventually stops" appears in Kleene's book [1952, chapter XIII, section 71]; there, Kleene further asserts that the problem is undecidable. The term "halting problem" was introduced by Davis [1958, chapter 5, section 2], who again proved that the problem is undecidable. For more information, Lucas [2021] has written a detailed survey on the history of the halting problem.

Although a number of authors claim that Alan Turing originally studied the halting problem in his 1936 paper, this is not the case! While Turing did prove the undecidability of a *satisfactoriness problem* in his paper, and while this problem does ask whether a given machine $\mathcal{M}$ is "circular" or "circle-free", this problem is not equivalent to the halting problem. Indeed, using the notion of *degrees of unsolvability* [Post, 1944], we note that while the halting problem belongs to the class of recursively enumerable sets of numbers (i.e., H_{TM} is of degree $\mathbf{0}'$), the satisfactoriness problem belongs to a strictly larger class (i.e., satisfactoriness is of degree $\mathbf{0}''$). See the aforementioned survey by Lucas [2021] for a proof of this fact.

An alternative, more Seussical proof of the undecidability of the halting problem was published by Pullum [2000].

No program can say what another will do.
Now, I won't just assert that, I'll prove it to you:
I will prove that although you might work till you drop,
you can't predict whether a program will stop.
— GEOFFREY PULLUM

5.3. There are many (many) more undecidable problems than those we have discussed in this section. The book chapter by Davis [1977] provides a nice overview of a number of undecidable problems in computer science and fields beyond, including group theory, combinatorics, and number theory. Harkleroad [1996] gives a gentler introduction to some of the same undecidability results.

5.4. As we noted, Hartmanis [1967] was the first to draw a connection between context-free languages and the computations of Turing machines. In tandem with the result that all invalid computation histories of a Turing machine are context-free, Hartmanis further showed that all valid computation histories of a Turing machine can be represented as the intersection of two context-free languages.

5.5. Both the equivalence problem and the inclusion problem were shown to be undecidable by Bar-Hillel, Perles, and Shamir [1961], whose proofs relied on a reduction from the Post correspondence problem. The undecidability of the universality problem follows from its relationship to the equivalence problem. Hartmanis [1967] gave alternative proofs of these same facts by reducing from the halting problem rather than the Post correspondence problem.

5.6. *Chapter notes will be added when this section is written.*

5.7. *Chapter notes will be added when this section is written.*

APPENDIX A

MATHEMATICAL BACKGROUND

To say that computer science and mathematics are closely related would be an understatement. Indeed, some of the pioneering work in computer science was done by career mathematicians in a time before "computer scientists" even existed. The bond between these two subjects is perhaps strongest when it comes to the theory of computing: flip to any page in this book and you'll find some kind of mathematical terminology or notation.

Comfort with mathematics is necessary for success in computer science. In this appendix, we will review some of the important notions one should know in order to learn and understand the theory of computing.

A.1. SETS AND SEQUENCES

The concepts underpinning almost everything we cover in this book are some of the most elementary in all of mathematics: sets and sequences.

Sets, Operations, and Properties

Let us begin with a very simple definition: that of a *set*.

Definition A.1 (Set)

A set is a collection of elements.

If an element a belongs to a set S, then we write $a \in S$. Otherwise, we write $a \notin S$. Elements of a set are unique; for example, two indistinguishable copies of an element a cannot belong to the same set S. If we require that some set contain more than one indistinguishable copy of some element, then we say that set is a *multiset*.

We may describe sets either by listing their elements explicitly, or by describing some property or properties possessed by each element in the set. Of course, if the set is infinite, we typically prefer to use the latter method.

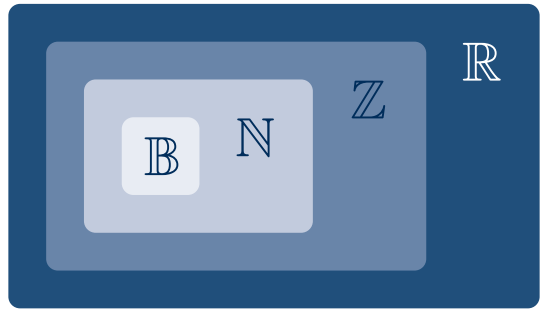


Figure A.1. Some common sets and their relationships.

Example A.2

Let's come up with some basic examples of sets:

- The set of all Canadian host cities of the Olympics is

$$C_{\text{Olympics}} = \{\text{Montréal, Calgary, Vancouver}\}.$$

- The set of all years in the 20th century in which the summer Olympics were held is

$$\begin{aligned} O_{1900\text{s}} = \{&1904, 1908, 1912, 1920, 1924, 1928, 1932,\\ &1936, 1948, 1952, 1956, 1960, 1964, 1968,\\ &1972, 1976, 1980, 1984, 1988, 1992, 1996\}. \end{aligned}$$

- The set of all prime numbers less than 20 is

$$P_{<20} = \{2, 3, 5, 7, 11, 13, 17, 19\}.$$

- The set of all positive odd integers is

$$S_{\text{odd}} = \{n \mid n = 2k + 1, k \geq 0\}.$$

There are some sets that we use frequently enough to warrant their own notation. The set of *natural numbers* is $\mathbb{N} = \{0, 1, 2, \dots\}$, and the set of *integers* is $\mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$. Occasionally, we also refer to the set of *real numbers*, denoted $\mathbb{R}$, although their definition is better left for a book on real analysis. Computer scientists in particular are often interested in the set of *binary digits* or *bits*, sometimes denoted $\mathbb{B} = \{0, 1\}$. These common sets and their relationships are depicted in Figure A.1.

The *cardinality* or *size* of a set S, denoted $|S|$, is equal to the number of elements in S. If $|S| = n$ for some finite $n \geq 0$, then we say that S is a *finite set*. Otherwise, we say that S is an *infinite set*. We say that an infinite set S is *countably infinite* if we can associate each element of S to exactly one natural number in $\mathbb{N}$, and each natural number is associated with exactly one element of S; if no such association is possible, then we say that S is *uncountably infinite*.

The unique set with cardinality zero is called the *empty set*, and we denote it by $\emptyset$. At the other extreme, the set of all elements under consideration is the *universal set* or just the *universe*, and is occasionally written $\mathcal{U}$.

There are a number of elementary and useful operations we can apply to sets. The *union* of two sets S and T, denoted $S \cup T$, is the set containing all elements that are in S or in T (or in both). The *intersection* of S and T, denoted $S \cap T$, is the set containing all elements that are in both S and T. The *complement* of a set S, denoted $\overline{S}$, is the set containing all elements from our universe that are not in S. Lastly, the *difference* of S and T, denoted $S \setminus T$, is the set of all elements of S that are not also in T.

Example A.3

Suppose $\mathcal{U} = \mathbb{N}$. The set of all positive even integers is

$$S_{\text{even}} = \overline{S_{\text{odd}}} = \mathbb{N} \setminus S_{\text{odd}}.$$

We can see that $S_{\text{odd}} \cup S_{\text{even}} = \mathcal{U}$, while $S_{\text{odd}} \cap S_{\text{even}} = \emptyset$.

From our example, we see that we can define our complement operation in terms of our difference operation by taking $\overline{S} = \mathcal{U} \setminus S$. As a consequence, we have that both $\overline{\mathcal{U}} = \emptyset$ and $\overline{\emptyset} = \mathcal{U}$.

Given two sets S and T, if every element of S is also an element of T, we say that S is a *subset* of T and we write $S \subseteq T$. If, additionally, T contains at least one element that S does not contain, then we say that S is a *proper subset* of T and we write $S \subset T$. If no element of S is an element of T and vice versa, then we say that S and T are *disjoint*.

Example A.4

Each of the following relationships between sets holds:

- The set S_{odd} is a proper subset of both $\mathbb{N}$ and $\mathbb{Z}$, while the sets S_{odd} and S_{even} are disjoint.
- It is possible to have chains of subsets, such as $\mathbb{B} \subset \mathbb{N} \subset \mathbb{Z} \subset \mathbb{R}$.
- For all sets S, $\emptyset \subseteq S \subseteq \mathcal{U}$.

> *Remark.* The notation for indicating subset and proper subset relationships is, annoyingly, inconsistent across the literature. Here, we denote a subset relationship by the symbol $\subseteq$, analogous to how the "less than or equal to" symbol $\leq$ indicates that the object on the right-hand side may be the same as the object on the left-hand side. Likewise, we denote a proper subset relationship by the symbol $\subset$, analogous to how the "less than" symbol $<$ indicates that the object on the right-hand side is strictly larger than the object on the left-hand side.

We can go one step further by defining *set equality* in terms of subsets: we say that two sets S and T are *equal* if both $S \subseteq T$ and $T \subseteq S$.

Lastly, the *power set* of a set S, denoted $\mathcal{P}(S)$, is the set of all subsets of S. For all sets S, it is the case that $|S| < |\mathcal{P}(S)|$; specifically, if S is a finite set, then $|\mathcal{P}(S)| = 2^{|S|}$. The power set of an infinite set always has infinite cardinality, but the power set of a countably infinite set has an uncountably infinite cardinality.

Example A.5

Let $S = \{1, 2, 3\}$. Then

$$\mathcal{P}(S) = \{\{\emptyset\}, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\},$$

and $|\mathcal{P}(S)| = 2^3 = 8$.

Note that, for every set S, the power set $\mathcal{P}(S)$ always contains the elements $\{\emptyset\}$ and S itself. It is also worth noting that $\emptyset$ and $\{\emptyset\}$ are *not* the same: $\emptyset$ is the unique set with zero elements, while $\{\emptyset\}$ is a set with cardinality 1 containing the element $\emptyset$.

Sequences and Operations

In a set, ordering does not matter: as long as two sets contain the same elements, they are equal. For example, the sets $\{1, 2, 4, 8\}$ and $\{2, 8, 4, 1\}$ are equal, despite their elements not appearing in the same order. If we need to maintain or preserve some ordering in a collection of elements, we must instead use a *sequence*.

Definition A.6 (Sequence)

A sequence is an ordered set of elements.

We distinguish notationally between sets and sequences in the following way: sets are surrounded by {braces}, while sequences are surrounded by (parentheses). We further distinguish sets from sequences by appending to

the sequence's label a subscripted n; thus, while we may denote a set by something like S, we will denote a sequence by something like A_n. Given a sequence $A_n = (a_0, a_1, a_2, \dots)$, we say that the element a_i is the ith *term* of the sequence.

We occasionally refer to a sequence having a finite length as a *tuple*, or as a *k-tuple* if we know the sequence contains k terms. A sequence having a length of two is called an *ordered pair*.

Unlike sets, a sequence may contain non-unique or repeated terms.

Example A.7

Let's come up with some basic examples of sequences:

- The sequence L_n of the digits in the decimal representation of the speed of light c, in metres per second, is
$$L_n = (2, 9, 9, 7, 9, 2, 4, 5, 8).$$
- The Fibonacci sequence F_n is defined as follows: fix $F_0 = F_1 = 1$. Then, for each $n \geq 2$, $F_n = F_{n-1} + F_{n-2}$. The first few terms of the Fibonacci sequence are
$$F_n = (1, 1, 2, 3, 5, 8, 13, 21, 34, \dots).$$
- The Thue–Morse sequence T_n is defined as follows: for all $n \geq 0$, $T_n = 0$ if the number of ones in the binary representation of n is even, and $T_n = 1$ if the number of ones is odd. The first few terms of the Thue–Morse sequence are
$$T_n = (0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, \dots).$$

Remark. The study of integer sequences is of keen interest to some mathematicians and computer scientists. The On-line Encyclopedia of Integer Sequences, accessible at `https://oeis.org`, contains over 375 000 examples of sequences, many of which include relationships to other sequences, code snippets to generate terms of the sequence, and citations to the literature.

Using the notion of sequences, we can define another set operation: the *Cartesian product*. Given two sets S and T, their Cartesian product, denoted $S \times T$, is the set of all ordered pairs where the first element of the pair comes from S and the second element comes from T. Formally speaking, $S \times T = \{(a, b) \mid a \in S \text{ and } b \in T\}$.

Example A.8

Let $S = \{1, 2, 3\}$ and $T = \{2, 4, 6\}$. Then

$$S \times T = \{(1, 2), (1, 4), (1, 6), (2, 2), (2, 4), (2, 6), (3, 2), (3, 4), (3, 6)\}.$$

We can, of course, take the Cartesian product of a set S with itself: we denote this by $S \times S = S^2$. We can further take $S^2 \times S = S \times S^2 = S^3$, and so on. In general, taking the Cartesian product of a set S with itself k times is denoted S^k.

Example A.9

Taking $\mathbb{Z} \times \mathbb{Z} = \mathbb{Z}^2$ yields the set of all ordered pairs of integers, which gives us the Cartesian coordinate system (otherwise known as the "xy-plane").

A.2. RELATIONS AND FUNCTIONS

By taking the idea of the Cartesian product further, we can define relations and functions, which associate elements of one set S to elements of another set T. The set S is called the *domain*, while the set T is called the *codomain*. Not all elements of T need be associated to some element of S; elements that are associated constitute the *range* of the relation or function.

Relations

A *relation* consists of ordered pairs from the Cartesian product $S \times T$. If $a \in S$ and $b \in T$, then the ordered pair (a, b) relates a and b.

Definition A.10 (Relation)

A relation R from a set S to a set T is a subset of $S \times T$.

Example A.11

There are many classic examples of relations:

- The equality relation is taken to be
$$R_= = \{(a, b) \in \mathbb{R} \times \mathbb{R} \mid a = b\}.$$
- The less-than relation is taken to be
$$R_< = \{(a, b) \in \mathbb{R} \times \mathbb{R} \mid a < b\}.$$

- The divides relation is taken to be

$$R_{\text{div}} = \{(a, b) \in \mathbb{Z} \times \mathbb{Z} \mid b = ac \text{ for some } c \in \mathbb{Z}\}.$$

- The rock-paper-scissors relation is taken to be

$$R_{\text{rps}} = \{(a, b) \in \{\text{r}, \text{p}, \text{s}\} \times \{\text{r}, \text{p}, \text{s}\} \mid a \text{ beats or ties } b\}.$$

We have assumed in our examples that relations are between two sets; that is, they are *binary* relations. We can generalize from binary relations to k-ary relations in the usual way.

Example A.12

Let's come up with some examples of relations having an arity greater than two:

- Let P be the set of professors, let C be the set of courses, and let $T = \{\text{fall}, \text{winter}\}$ be the set of academic terms. Define a relation $R_{\text{sched}} \subseteq P \times C \times T$ where a tuple $(p, c, t) \in R_{\text{sched}}$ indicates that professor p is teaching course c in term t.

  (Finding the complete set of tuples in R_{sched} is left as an exercise for the Registrar's Office.)

- Let N be the set of names of people, let O be the set of origin airports, let D be the set of destination airports, and let F be the set of flight numbers. Define a relation $R_{\text{flight}} \subseteq N \times O \times D \times F$ where a tuple $(n, o, d, f) \in R_{\text{flight}}$ indicates that person n flew from airport o to airport d via the flight f.

We can define a number of properties of a relation R depending on which ordered pairs belong to the relation. Let a, b, and c each be elements. Then

- R is *reflexive* if, for all a, $(a, a) \in R$;
- R is *symmetric* if $(a, b) \in R$ implies $(b, a) \in R$;
- R is *antisymmetric* if $(a, b) \in R$ and $(b, a) \in R$ implies $a = b$; and
- R is *transitive* if $(a, b) \in R$ and $(b, c) \in R$ implies $(a, c) \in R$.

Despite their names, the properties of symmetry and antisymmetry are not mutually exclusive. A relation may be both symmetric and antisymmetric.

Example A.13

Returning to our classic relations from Example A.11, we see that:

- the equality relation $R_=$ is
  - reflexive since $a = a$ for all a,
  - symmetric since $a = b$ implies $b = a$ for all a and b,
  - antisymmetric since both $a = b$ and $b = a$ implies that a and b are the same element, and
  - transitive since $a = b$ and $b = c$ implies that $a = c$ for all a, b, and c;
- the less-than relation $R_<$ is
  - not reflexive since $a \not< a$ for any a,
  - not symmetric since $a < b$ does not imply that $b < a$ for any a or b,
  - not antisymmetric since it is impossible to have both $a < b$ and $b < a$, and
  - transitive since $a < b$ and $b < c$ implies that $a < c$ for all a, b, and c;
- the divides relation R_{div} is
  - reflexive since a divides a for all a,
  - not symmetric since a dividing b does not imply that b divides a for all a and b,
  - antisymmetric since both a dividing b and b dividing a implies that a and b are the same element, and
  - transitive since a dividing b and b dividing c implies that $c = kb = k(\ell a) = (k\ell)a$ for some $k, \ell \in \mathbb{Z}$; and
- the rock-paper-scissors relation R_{rps} is
  - reflexive since a ties a for all a,
  - not symmetric since a beating or tying b does not imply that b beats or ties a for all a and b,
  - antisymmetric since both a tying b and b tying a implies that a and b are the same element, and
  - not transitive since (for example) rock beating scissors and scissors beating paper does not imply that rock beats paper.

Functions

A *function* from a set S to a set T is a special kind of relation where each element of S is mapped to exactly one element of T. All functions are relations, but not all relations are functions.

Definition A.14 (Function)

A function f from a set S to a set T is a subset of $S \times T$ such that, for each $a \in S$, there exists exactly one $b \in T$ such that $(a, b) \in f$.

If f is a function from a set S to a set T, then we often denote this by the shorthand notation $f\colon S \to T$, where the colon is taken to mean "from" and the arrow is taken to mean "to". If $(a, b) \in f$, then we write $f(a) = b$.

Example A.15

There are many classic examples of functions $f\colon \mathbb{N} \to \mathbb{N}$:

- The constant function, $f(x) = c$ for some $c \in \mathbb{N}$, where $f(0) = c$, $f(1) = c$, $f(2) = c$, $f(3) = c$, $f(4) = c$, $f(5) = c$, ...
- The integer division function, $f(x) = \lfloor x/2 \rfloor$, where $f(0) = 0$, $f(1) = 0$, $f(2) = 1$, $f(3) = 1$, $f(4) = 2$, $f(5) = 2$, ...
- The linear function, $f(x) = x$, where $f(0) = 0$, $f(1) = 1$, $f(2) = 2$, $f(3) = 3$, $f(4) = 4$, $f(5) = 5$, ...
- The quadratic function, $f(x) = x^2$, where $f(0) = 0$, $f(1) = 1$, $f(2) = 4$, $f(3) = 9$, $f(4) = 16$, $f(5) = 25$, ...
- The exponential function, $f(x) = 2^x$, where $f(0) = 1$, $f(1) = 2$, $f(2) = 4$, $f(3) = 8$, $f(4) = 16$, $f(5) = 32$, ...

Functions are sometimes expressed visually by way of a *bubble diagram*, wherein the two sets S and T are represented by two large ellipses and elements within each set are represented by smaller circles. If $f(a) = b$ for some elements $a \in S$ and $b \in T$, then this is depicted by an arrow in the bubble diagram. An example of a bubble diagram is shown in Figure A.2.

Similar to relations, we can define a number of properties of a function f from a set S to a set T.

- f is *injective* (or *one-to-one*) if, for all $a_1, a_2 \in S$ where $a_1 \neq a_2$, $f(a_1) \neq f(a_2)$;

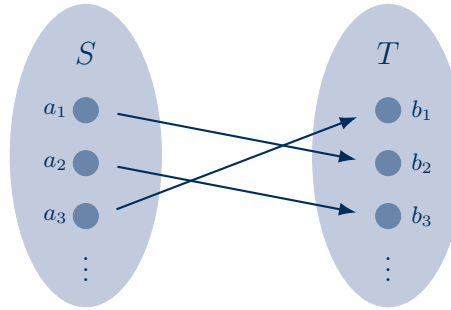


Figure A.2. An example of a bubble diagram.

- f is *surjective* (or *onto*) if, for all $b \in T$, there exists $a \in S$ such that $f(a) = b$; and
- f is *bijective* if it is both injective and surjective.

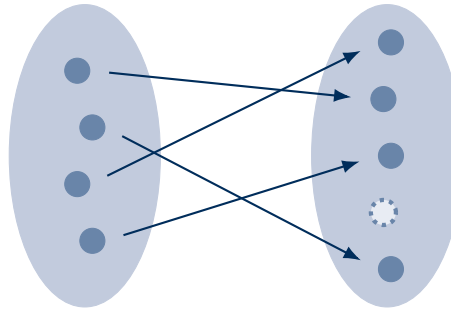
Examples of each type of function are illustrated in Figure A.3.

Example A.16

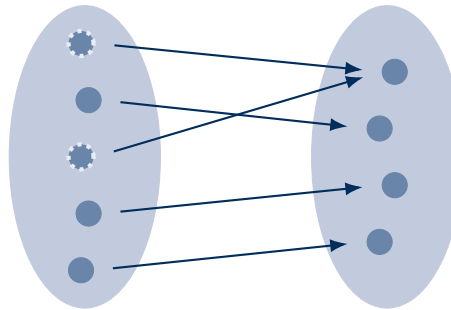
Returning to our classic functions from Example A.15, we see that:

- the constant function, $f(x) = c$ for some $c \in \mathbb{N}$, is not injective and not surjective;
- the integer division function, $f(x) = \lfloor x/2 \rfloor$, is surjective but not injective;
- the linear function, $f(x) = x$, is both injective and surjective (and thus, bijective);
- the quadratic function, $f(x) = x^2$, is injective but not surjective; and
- the exponential function, $f(x) = 2^x$, is also injective but not surjective.

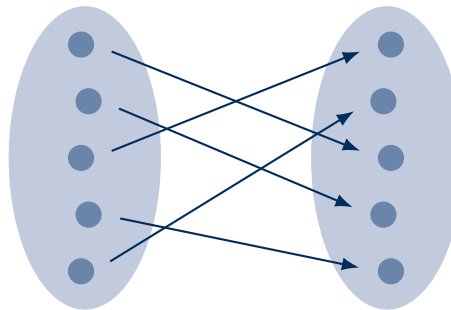
Remark. If you need a handy little mnemonic to remember how to associate injective/surjective with one-to-one/onto, just remember that "sur" is the French word for "on", so a surjective function is onto.



(a) A function that is injective, but not surjective.



(b) A function that is surjective, but not injective.



(c) A function that is both injective and surjective; that is, bijective.

Figure A.3. Examples of different kinds of functions.

A.3. GRAPHS

Many ideas in mathematics and computer science have a natural visual representation in the form of a *graph*. A graph in this context is not a plot or a chart (as in, say, "the graph of a function"); rather, it is a structure that consists of two components: *vertices*—also called *nodes*—that are connected to one another by *edges*. Indeed, we may define a graph solely in terms of its vertex set and edge set.

Definition A.17 (Graph)

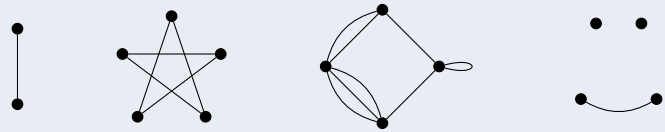
A graph $G = (V, E)$ consists of a set of vertices V and a set of edges E, where each element $e \in E$ is a pair $\{u, v\}$ of vertices $u, v \in V$.

If there exists an edge $e = \{u, v\}$ between two vertices u and v, then we say that these two vertices are *adjacent* to one another. At the same time, we say that both u and v are *incident* to the edge e.

Definition A.17 is worded in a very general way on purpose, since we want graphs to be able to model potentially many different ideas. For example, our definition allows for constructions such as multiple edges between the same pair of vertices, or edges that join a vertex to itself (called *loops*).

Example A.18

Each of the following is a graph:



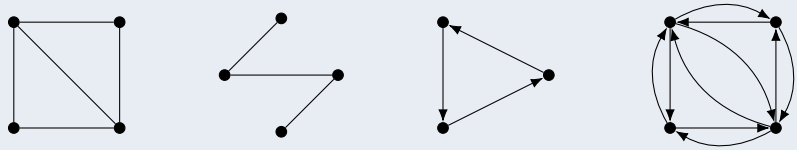
Note that Definition A.17 also supposes that the set of edges E consists of *unordered* pairs. By our definition, if an edge exists between vertices u and v, then this means that another edge implicitly exists between v and u.

If we instead require that E consists of *ordered* pairs, then the existence of an edge between vertices u and v does not necessarily imply the existence of an edge between v and u. We say that such graphs are *directed*, because the direction of each edge in the set E matters when we move between vertices.

Example A.19

Of the following graphs, the two leftmost graphs are undirected, while the two rightmost graphs are directed. In fact, the rightmost graph is

the directed equivalent of the leftmost graph.

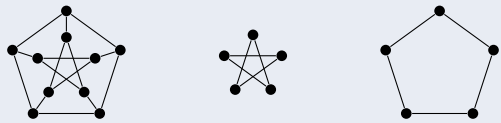


If a graph, whether undirected or directed, has the property that we can reach any vertex from any other vertex by way of following some sequence of edges, then we say that graph is *connected*. Determining whether a directed graph is connected is somewhat more difficult than in the undirected case, since we may not be able to follow the same sequence of directed edges between vertices v and u as we could between u and v.

If we have two graphs $G = (V, E)$ and $H = (V', E')$ where $V' \subseteq V$ and $E' \subseteq E$, then we say that H is a *subgraph* of G. In other words, H is a copy of G with vertices or edges (or both) removed. Note that if we remove some vertex, then we must also remove any edges that were incident to the removed vertex.

Example A.20

The leftmost graph contains the two rightmost graphs as subgraphs. Observe that the five edges connecting the inner subgraph to the outer subgraph were removed and do not appear in either subgraph.



Given a graph $G = (V, E)$, two vertices $u, v \in V$, and a number $n \in \mathbb{N}$, a *path* of length n from vertex u to vertex v is a sequence of edges $(e_1, \ldots, e_n)$ such that $e_1 = \{u, w_1\}$, $e_2 = \{w_1, w_2\}$, $\ldots$, and $e_n = \{w_{n-1}, v\}$, where $w_1, \ldots, w_{n-1} \in V$. We can define paths in directed graphs by making the appropriate changes to the individual edges.

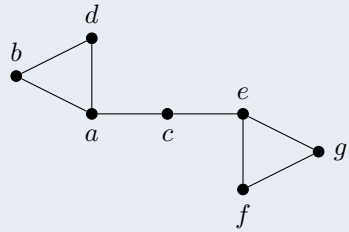
We say that a path of length n from a vertex u to a vertex v is called a *circuit* (or a *cycle*) if $u = v$ and $n > 0$.

If no edge in the path or circuit is included more than once, then we say that the path or circuit is *simple*.

Example A.21

In the following graph, we see that (among many others) there exist

- paths a–b, a–c–e–g, c–e–c–a, and f–g–f;
- simple paths b–a–c–e, d–a–b, and f–e–c;
- circuits a–c–e–c–a and a–d–b–a–c–a; and
- simple circuits a–b–d–a and e–f–g–e.



APPENDIX B

THE GREEK ALPHABET

If some of the symbols in this book are Greek to you, then the following table may help. Below, you will find the uppercase and lowercase forms of all 24 Greek letters, together with selected examples of how these letters are used in the theory of computing.

Table B.1
THE GREEK ALPHABET

Alpha	A	α	
Beta	B	β	
Gamma	Γ	γ	A tape alphabet, or a generic second alphabet
Delta	Δ	δ	A transition function
Epsilon	E	ϵ	The empty word
Zeta	Z	ζ	
Eta	H	η	
Theta	Θ	θ	Asymptotic tight bound
Iota	I	ι	
Kappa	K	κ	
Lambda	Λ	λ	
Mu	M	μ	
Nu	N	ν	
Xi	Ξ	ξ	
Omicron	O	o	Asymptotic upper bound (alt. using Latin O/o)
Pi	Π	π	
Rho	P	ρ	
Sigma	Σ	σ	An input alphabet, or a generic alphabet
Tau	T	τ	
Upsilon	Υ	υ	
Phi	Φ	ϕ	
Chi	X	χ	
Psi	Ψ	ψ	
Omega	Ω	ω	Asymptotic lower bound

BIBLIOGRAPHY

Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, Cambridge, 2009.

Christopher Bader and Arnaldo Moura. A generalization of Ogden's lemma. *Journal of the ACM*, 29(2):404–407, 1982.

Yehoshua Bar-Hillel, Micha Perles, and Eliahu Shamir. On formal properties of simple phrase structure grammars. *Zeitschrift für Phonetik, Sprachwissenschaft und Kommunikationsforschung*, 14(2):143–172, 1961.

Bruce H. Barnes. A programmer's view of automata. *ACM Computing Surveys*, 4(4):221–239, 1972.

Elwyn R. Berlekamp, John H. Conway, and Richard K. Guy. *Winning Ways for Your Mathematical Plays*, volume 4. A K Peters, Wellesley, second edition, 2004.

Leonard Bloomfield. *Language*. Henry Holt, New York, 1933.

Luc Boasson and Sándor Horváth. On languages satisfying Ogden's lemma. *RAIRO Informatique théorique/Theoretical Computer Science*, 12(3):201–202, 1978.

Janusz A. Brzozowski. A survey of regular expressions and their applications. *IRE Transactions on Electronic Computers*, 11(3):324–335, 1962.

Janusz A. Brzozowski and Edward J. McCluskey, Jr. Signal flow graph techniques for sequential circuit state diagrams. *IEEE Transactions on Electronic Computers*, 12(2):67–76, 1963.

Arthur W. Burks, Don W. Warren, and Jesse B. Wright. An analysis of a logical machine using parenthesis-free notation. *Mathematical Tables and Other Aids to Computation*, 8(46):53–57, 1954.

Bytejacker. Minecraft - Notch Interview! https://www.youtube.com/watch?v=rqUDam_KJno, 2010.

Cezar Câmpeanu, Kai Salomaa, and Sheng Yu. A formal study of practical regular expressions. *International Journal of Foundations of Computer Science*, 14(6):1007–1018, 2003.

Georg Cantor. Ueber eine Eigenschaft des Inbegriffes aller reellen algebraischen Zahlen [On a property of the set of real algebraic numbers]. *Journal für die reine und angewandte Mathematik*, 77:258–262, 1874. Translation: [Ewald, 1996, pages 839–843].

Georg Cantor. Ueber eine elementare Frage der Mannigfaltigkeitslehre [On an elementary question in the theory of manifolds]. *Jahresbericht der Deutschen Mathematiker-Vereinigung*, 1:75–78, 1891. Translation: [Ewald, 1996, pages 920–922].

Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124, 1956.

Noam Chomsky. Some properties of phrase structure grammars. Research Laboratory of Electronics Quarterly Progress Report 49:108–111, Massachusetts Institute of Technology, 1958.

Noam Chomsky. On certain formal properties of grammars. *Information and Control*, 2(2):137–167, 1959a.

Noam Chomsky. A note on phrase structure grammars. *Information and Control*, 2(4):393–395, 1959b.

Noam Chomsky. Context-free grammars and pushdown storage. Research Laboratory of Electronics Quarterly Progress Report 65:187–194, Massachusetts Institute of Technology, 1962.

Noam Chomsky and George A. Miller. Finite state languages. *Information and Control*, 1(2):91–112, 1958.

Noam Chomsky and Marcel-Paul Schützenberger. The algebraic theory of context-free languages. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, volume 26 of *Studies in Logic and the Foundations of Mathematics*, pages 118–161. North-Holland Publishing Co., Amsterdam, 1963.

Alonzo Church. A note on the Entscheidungsproblem. *Journal of Symbolic Logic*, 1(1):40–41, 1936a.

Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, 1936b.

Alonzo Church. Review of "On computable numbers, with an application to the Entscheidungsproblem" by A. M. Turing. *Journal of Symbolic Logic*, 2(1):42–43, 1937.

Alex Churchill, Stella Biderman, and Austin Herrick. Magic: The Gathering is Turing complete. In M. Farach-Colton, G. Prencipe, and R. Uehara, editors, *Proceedings of the 10th International Conference on Fun with Algorithms (FUN 2021)*, volume 157 of *Leibniz International Proceedings in Informatics*, pages 9:1–9:19, Sicily, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

Matthew Cook. Universality in elementary cellular automata. *Complex Systems*, 15:1–40, 2004.

B. Jack Copeland, editor. *The Essential Turing: Seminal Writings in Computing, Logic, Philosophy, Artificial Intelligence, and Artificial Life, plus The Secrets of Enigma.* Clarendon Press, Oxford, 2004.

Irving M. Copi, Calvin C. Elgot, and Jesse B. Wright. Realization of events by logical nets. *Journal of the ACM*, 5(2):181–196, 1958.

Russ Cox. Regular expression matching can be simple and fast (but is slow in Java, Perl, PHP, Python, Ruby, ...). `https://swtch.com/~rsc/regexp/regexp1.html`, 2007.

Martin Davis. *Computability and Unsolvability.* McGraw-Hill, New York, 1958.

Martin Davis, editor. *The Undecidable: Basic Papers on Undecidable Propositions, Unsolvable Problems and Computable Functions.* Raven Press, Hewlett, 1965.

Martin Davis. Unsolvable problems. In J. Barwise, editor, *Handbook of Mathematical Logic*, volume 90 of *Studies in Logic and the Foundations of Mathematics*, pages 567–594. North-Holland Publishing Co., Amsterdam, 1977.

Martin Davis. *The Universal Computer: The Road from Leibniz to Turing.* W. W. Norton, New York, 2000.

Stephen Dolan. `mov` is Turing-complete. `https://drwho.virtadpt.net/files/mov.pdf`, 2013.

Andrzej Ehrenfeucht, Rohit J. Parikh, and Grzegorz Rozenberg. Pumping lemmas for regular sets. *SIAM Journal on Computing*, 10(3):536–541, 1981.

Herbert B. Enderton. Elements of recursion theory. In J. Barwise, editor, *Handbook of Mathematical Logic*, volume 90 of *Studies in Logic and the Foundations of Mathematics*, pages 527–566. North-Holland Publishing Co., Amsterdam, 1977.

Jeff Erickson. *Algorithms*. Self-published open educational resource, first edition, 2019. http://algorithms.wtf.

R. James Evey. Application of pushdown-store machines. In J. D. Tupac, editor, *Proceedings of the Fall Joint Computer Conference*, volume 24 of *AFIPS Conference Proceedings*, pages 215–227, Las Vegas, 1963a. American Federation of Information Processing Societies.

R. James Evey. *The Theory and Applications of Pushdown Store Machines*. Doctoral thesis, Harvard University, 1963b.

William Ewald. *From Kant to Hilbert: A Source Book in the Foundations of Mathematics*, volume 2. Clarendon Press, Oxford, 1996.

Jeffrey E. F. Friedl. *Mastering Regular Expressions*. O'Reilly Media, Sebastopol, third edition, 2006.

Yuan Gao, Nelma Moreira, Rogério Reis, and Sheng Yu. A survey on operational state complexity. *Journal of Automata, Languages and Combinatorics*, 21(4):251–310, 2016.

Seymour Ginsburg. *The Mathematical Theory of Context Free Languages*. McGraw-Hill, New York, 1966.

Seymour Ginsburg and Sheila Greibach. Deterministic context free languages. *Information and Control*, 9(6):620–648, 1966.

Seymour Ginsburg and Henry G. Rice. Two families of languages related to ALGOL. *Journal of the ACM*, 9(3):350–371, 1962.

Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I [On formally undecidable propositions of Principia Mathematica and related systems I]. *Monatschefte für Mathematik und Physik*, pages 173–198, 1931. Translation: [Van Heijenoort, 1967, pages 596–616].

Kurt Gödel. On undecidable propositions of formal mathematical systems. In M. Davis, editor, *The Undecidable: Basic Papers on Undecidable Propositions, Unsolvable Problems and Computable Functions*, pages 39–74. Raven Press, Hewlett, 1965.

Andy Gordon and Simon Peyton Jones. LAMBDA: The ultimate Excel worksheet function. https://www.microsoft.com/en-us/research/blog/lambda-the-ultimatae-excel-worksheet-function/, 2021.

James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, Boston, third edition, 2005.

Robert Gray. Georg Cantor and transcendental numbers. *American Mathematical Monthly*, 101(9):819–832, 1994.

Sheila A. Greibach. A new normal-form theorem for context-free phrase structure grammars. *Journal of the ACM*, 12(1):42–52, 1965.

Sheila A. Greibach. Formal languages: Origins and directions. *Annals of the History of Computing*, 3(1):14–41, 1981.

Leon Harkleroad. How mathematicians know what computers can't do. *College Mathematics Journal*, 27(1):37–42, 1996.

Zellig S. Harris. From morpheme to utterance. *Language*, 22(3):161–183, 1946.

Juris Hartmanis. Context-free languages and Turing machine computations. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 42–51. American Mathematical Society, Providence, 1967.

Juris Hartmanis and Herbert Shank. On the recognition of primes by automata. *Journal of the ACM*, 15(3):382–389, 1968.

Juris Hartmanis and Richard E. Stearns. On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 117: 285–306, 1965.

Frederick C. Hennie and Richard E. Stearns. Two-tape simulation of multitape Turing machines. *Journal of the ACM*, 13(4):533–546, 1966.

David Hilbert. Mathematische Probleme [Mathematical problems]. *Nachrichten von der Königlichen Gesellschaft der Wissenschaften zu Göttingen*, pages 253–297, 1900. Reprint: [Hilbert, 1901]. Translation: [Hilbert, 1902].

David Hilbert. Mathematische Probleme [Mathematical problems]. *Archiv der Mathematik und Physik*, s3-1:44–63 and 213–237, 1901.

David Hilbert. Mathematical problems. *Bulletin of the American Mathematical Society*, 8(10):437–479, 1902. Translated by Mary Winston Newson.

David Hilbert and Wilhelm Ackermann. *Grundzüge der theoretischen Logik*. Springer-Verlag, Berlin, 1928.

Andrew Hodges. *Alan Turing: The Enigma*. Simon & Schuster, New York, 1983.

Markus Holzer and Martin Kutrib. Descriptional and computational complexity of finite automata — a survey. *Information and Computation*, 209 (3):456–470, 2011.

John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, 1979.

Sándor Horváth. The family of languages satisfying Bar-Hillel's lemma. *RAIRO Informatique théorique/Theoretical Computer Science*, 12(3):193–199, 1978.

David A. Huffman. The synthesis of sequential switching circuits. *Journal of the Franklin Institute*, 257(3):161–190, 1954a.

David A. Huffman. The synthesis of sequential switching circuits, part II. *Journal of the Franklin Institute*, 257(4):275–303, 1954b.

Seiiti Huzino and Ryoko Shibata. An elementary proof of R.W. Ritchie's theorem on the set of squares. *Memoirs of the Faculty of Science, Kyushu University, Series A*, 31(1):9–14, 1977.

Jeffrey Jaffe. A necessary and sufficient pumping lemma for regular languages. *ACM SIGACT News*, 10(2):48–49, 1978.

Gerhard Jäger and James Rogers. Formal language theory: refining the Chomsky hierarchy. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 367:1956–1970, 2012.

Richard Johnsonbaugh and David P. Miller. Converses of pumping lemmas. *ACM SIGCSE Bulletin*, 22(1):27–30, 1990.

Jong89. DF map archive: Razorlength - 1036 Early Winter. `https://mkv25.net/dfma/map-8269`, 2009.

Philip E. B. Jourdain. The logical work of Leibniz. *The Monist*, 26(4): 504–523, 1916.

Richard Kaye. Infinite versions of Minesweeper are Turing complete. School of Mathematics preprint 2000/15, University of Birmingham, 2000.

Stephen C. Kleene. General recursive functions of natural numbers. *Mathematische Annalen*, 112(1):727–742, 1936a.

Stephen C. Kleene. λ-definability and recursiveness. *Duke Mathematical Journal*, 2(2):340–353, 1936b.

Stephen C. Kleene. Recursive predicates and quantifiers. *Transactions of the American Mathematical Society*, 53(1):41–73, 1943.

Stephen C. Kleene. Representation of events in nerve nets and finite automata. Research memorandum RM-704, RAND Corporation, 1951. Reprint: [Kleene, 1956].

Stephen C. Kleene. *Introduction to Metamathematics*. D. Van Nostrand, Princeton, 1952.

Stephen C. Kleene. Representation of events in nerve nets and finite automata. In C. E. Shannon and J. McCarthy, editors, *Automata Studies*, volume 34 of *Annals of Mathematics Studies*, pages 3–42. Princeton University Press, Princeton, 1956.

Dexter C. Kozen. *Automata and Computability*. Undergraduate Texts in Computer Science. Springer-Verlag, New York, 1997.

Dexter C. Kozen. *Theory of Computation*. Texts in Computer Science. Springer-Verlag, London, 2006.

Marcus Kracht. Too many languages satisfy Ogden's lemma. In S. Arunachalam, M. Baranowski, U. Horesh, I. Ross, T. Sanchez, T. Scheffler, S. Sundaresan, and A. Williams, editors, *Proceedings of the 27th Annual Penn Linguistics Colloquium*, volume 10 of *University of Pennsylvania Working Papers in Linguistics*, pages 115–121, Philadelphia, 2004. University of Pennsylvania.

Martin Lange and Hans Leiß. To CNF or not to CNF? An efficient yet presentable version of the CYK algorithm. *Informatica Didactica*, 8, 2009.

Chester Y. Lee. Automata and finite automata. *Bell System Technical Journal*, 39(5):1267–1295, 1960.

Wolfgang Lenzen. Leibniz and the calculus ratiocinator. In S. O. Hansson, editor, *Technology and Mathematics: Philosophical and Historical Investigations*, volume 30 of *Philosophy of Engineering and Technology*, pages 47–78. Springer-Verlag, Cham, 2018.

Salvador Lucas. The origins of the halting problem. *Journal of Logical and Algebraic Methods in Programming*, 121:100687, 2021.

Jan Łukasiewicz. O znaczeniu i potrzebach logiki matematycznej [On the meaning and needs of mathematical logic]. *Nauka Polska*, 10:604–620, 1929.

Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5(4): 115–133, 1943.

Robert McNaughton. The theory of automata, a survey. *Advances in Computers*, 2:379–421, 1961.

Robert McNaughton and Hisao Yamada. Regular expressions and state graphs for automata. *IRE Transactions on Electronic Computers*, 9(1): 39–47, 1960.

George H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955.

Marvin L. Minsky. Recursive unsolvability of Post's problem of "tag" and other topics in theory of Turing machines. *Annals of Mathematics*, 74(3): 437–455, 1961.

Marvin L. Minsky. Size and structure of universal Turing machines using tag systems. In J. C. E. Dekker, editor, *Recursive Function Theory*, volume 5 of *Proceedings of Symposia in Pure Mathematics*, pages 229–238. American Mathematical Society, Providence, 1962.

Marvin L. Minsky and Seymour Papert. Unrecognizable sets of numbers. *Journal of the ACM*, 13(2):281–286, 1966.

Edward F. Moore. Gedanken-experiments on sequential machines. In C. E. Shannon and J. McCarthy, editors, *Automata Studies*, volume 34 of *Annals of Mathematics Studies*, pages 129–153. Princeton University Press, Princeton, 1956.

Frank R. Moore. On the bounds for state-set size in the proofs of equivalence between deterministic, nondeterministic, and two-way finite automata. *IEEE Transactions on Computers*, 20(10):1211–1214, 1971.

Trygve Nagell, Atle Selberg, Sigmund Selberg, and Knut Thalberg, editors. *Selected Mathematical Papers of Axel Thue*. Universitetsforlaget, Oslo, 1977.

Allen Newell and John C. Shaw. Programming the logic theory machine. In J. L. Barnes, editor, *Proceedings of the Western Joint Computer Conference*, pages 230–240, Los Angeles, 1957. Institute of Radio Engineers.

Anthony G. Oettinger. Automatic syntactic analysis and the pushdown store. In R. Jakobson, editor, *Structure of Language and its Mathematical Aspects*, volume 12 of *Proceedings of Symposia in Applied Mathematics*, pages 104–129. American Mathematical Society, Providence, 1961.

William Ogden. A helpful result for proving inherent ambiguity. *Mathematical Systems Theory*, 2(3):191–194, 1968.

Tomás Oliveira e Silva, Siegfried Herzog, and Silvio Pardi. Empirical verification of the even Goldbach conjecture and computation of prime gaps up to $4 \cdot 10^{18}$. *Mathematics of Computation*, 83(288):2033–2060, 2014.

Gene Ott and Neil H. Feinstein. Design of sequential machines from their regular expressions. *Journal of the ACM*, 8(4):585–600, 1961.

Pāṇini. Aṣṭādhyāyī [The book of eight chapters], c. 500 BCE. Translation: [Sharma, 2002].

Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, Reading, 1994.

Rohit J. Parikh. Language-generating devices. Research Laboratory of Electronics Quarterly Progress Report 60:199–212, Massachusetts Institute of Technology, 1961. Reprint: [Parikh, 1966].

Rohit J. Parikh. On context-free languages. *Journal of the ACM*, 13(4): 570–581, 1966.

Dominique Perrin. Les débuts de la théorie des automates [The beginnings of automata theory]. *Technique et Science Informatique*, 14:409–433, 1995.

Charles Petzold. *The Annotated Turing: A Guided Tour through Alan Turing's Historic Paper on Computability and the Turing Machine*. Wiley Publishing, Indianapolis, 2008.

Emil L. Post. Finite combinatory processes—formulation 1. *Journal of Symbolic Logic*, 1(3):103–105, 1936.

Emil L. Post. Formal reductions of the general combinatorial decision problem. *American Journal of Mathematics*, 65(2):197–215, 1943.

Emil L. Post. Recursively enumerable sets of positive integers and their decision problems. *Bulletin of the American Mathematical Society*, 50(5): 284–316, 1944.

Emil L. Post. A variant of a recursively unsolvable problem. *Bulletin of the American Mathematical Society*, 52(4):264–269, 1946.

James F. Power. Thue's 1914 paper: a translation. `https://arxiv.org/abs/1308.5858`, 2013.

Geoffrey K. Pullum. Scooping the loop snooper. *Mathematics Magazine*, 73 (4):319–320, 2000. Corrigendum: [Pullum, 2022].

Geoffrey K. Pullum. Scooping the loop snooper. http://www.lel.ed.ac.uk/~gpullum/loopsnoop.html, 2022.

Michael O. Rabin and Dana Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2):114–125, 1959.

Henry G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2): 358–366, 1953.

Elaine Rich. *Automata, Computability, and Complexity: Theory and Applications*. Pearson Prentice Hall, Upper Saddle River, 2008.

Robert W. Ritchie. Finite automata and the set of squares. *Journal of the ACM*, 10(4):528–531, 1963.

Yurii Rogozhin. Small universal Turing machines. *Theoretical Computer Science*, 168(2):215–240, 1996.

Simon Scarle. Implications of the Turing completeness of reaction-diffusion models, informed by GPGPU simulations on an XBox 360: Cardiac arrhythmias, re-entry and the Halting problem. *Computational Biology and Chemistry*, 33(4):253–260, 2009.

Stephen Scheinberg. Note on the boolean properties of context free languages. *Information and Control*, 3(4):372–375, 1960.

Marcel-Paul Schützenberger. Certain elementary families of automata. In J. Fox, editor, *Proceedings of the Symposium on Mathematical Theory of Automata*, volume 12 of *Microwave Research Institute Symposia Series*, pages 139–153, New York, 1962. Polytechnic Institute of Brooklyn.

Marcel-Paul Schützenberger. On context-free languages and push-down automata. *Information and Control*, 6(3):246–264, 1963.

Marcel-Paul Schützenberger. A remark on acceptable sets of numbers. *Journal of the ACM*, 15(2):300–303, 1968.

Pieter A. M. Seuren. The Chomsky hierarchy in perspective. In *From Whorf to Montague: Explorations in the Theory of Language*, chapter 6, pages 205–238. Oxford University Press, Oxford, 2013.

Claude E. Shannon. A universal Turing machine with two internal states. In C. E. Shannon and J. McCarthy, editors, *Automata Studies*, volume 34 of *Annals of Mathematics Studies*, pages 157–165. Princeton University Press, Princeton, 1956.

Claude E. Shannon. Von Neumann's contributions to automata theory. *Bulletin of the American Mathematical Society*, 64(3):123–129, 1958.

Ehud Shapiro. A mechanical Turing machine: Blueprint for a biomolecular computer. *Interface Focus*, 2:497–503, 2012.

Norman Shapiro. Degrees of computability. *Transactions of the American Mathematical Society*, 82(2):281–299, 1956.

Rama Nath Sharma. *The Aṣṭādhyāyī of Pāṇini*. Munshiram Manoharlal Publishers, New Delhi, second edition, 2002. Six volumes.

Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, Boston, third edition, 2013.

Alex Smith. Universality of Wolfram's 2, 3 Turing machine. *Complex Systems*, 29(1):1–44, 2020.

Robert I. Soare. Computability and recursion. *Bulletin of Symbolic Logic*, 2 (3):284–321, 1996.

Christopher Strachey. An impossible program. *The Computer Journal*, 7 (4):313, 1965.

Ken Thompson. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968.

Axel Thue. Probleme über Veränderungen von Zeichenreihen nach gegebenen Regeln [Problems concerning the transformation of symbol sequences according to given rules]. *Skrifter udgivne af Videnskabs-Selskabet i Christiania, I. Mathematisk-naturvidenskabelig Klasse*, 10:1–34, 1914. Reprint: [Nagell, Selberg, Selberg, and Thalberg, 1977, pages 493–524]. Translation: [Power, 2013].

Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1936. Corrigendum: [Turing, 1937].

Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. A correction. *Proceedings of the London Mathematical Society*, s2-43(1):544–546, 1937.

Dermot Turing. *Prof: Alan Turing Decoded*. The History Press, Gloucestershire, 2015.

Sara Turing. *Alan M. Turing*. W. Heffer & Sons, Cambridge, 1959. Reprint: [Turing, 2012].

Sara Turing. *Alan M. Turing: Centenary Edition*. Cambridge University Press, Cambridge, 2012.

Jean van Heijenoort, editor. *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931*. Harvard University Press, Cambridge, 1967.

John von Neumann. The general and logical theory of automata. In L. A. Jeffress, editor, *Cerebral Mechanisms in Behavior: The Hixon Symposium*, pages 1–31. John Wiley & Sons, New York, 1951.

Rulon S. Wells. Immediate constituents. *Language*, 23(2):81–117, 1947.

Tom Wildenhain. On the Turing completeness of MS PowerPoint. In *Proceedings of the 11th Annual Intercalary Robot Dance Party in Celebration of Workshop on Symposium about 2^6th Birthdays; in particular, that of Harry Q. Bovik (SIGBOVIK 2017)*, pages 102–106, Pittsburgh, 2017. Association for Computational Heresy.

David S. Wise. A strong pumping lemma for context-free languages. *Theoretical Computer Science*, 3(3):359–369, 1976.

Stephen Wolfram. *A New Kind of Science*. Wolfram Media, Champaign, 2002.

Damien Woods and Turlough Neary. The complexity of small universal Turing machines: A survey. *Theoretical Computer Science*, 410(4–5): 443–450, 2009.

Sheng Yu. Regular languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 1, pages 41–110. Springer-Verlag, Berlin, 1997.

Richard Zach. Hilbert's program then and now. In D. Jacquette, editor, *Philosophy of Logic*, volume 5 of *Handbook of the Philosophy of Science*, pages 411–447. North-Holland Publishing Co., Amsterdam, 2007.

THEORY OF COMPUTING: AN OPEN INTRODUCTION
Taylor J. Smith

What *is* the theory of computing? Computers are machines made by humans, so surely we should know all about how they work. However, all the day-to-day work we do with our computers belies the reality of computation itself and all of its intricacies. For even if we were to remove all of the physical components of a computer, producing an abstract machine that has unlimited processing power and unlimited memory, we would still encounter problems that are impossible for even this machine to solve in general.

In this α pre-publication edition, we learn what it means for a computer to compute by starting with one of the simplest models of computation: the finite automaton. We then augment this model with forms of storage—namely, a stack and a tape, which give us the pushdown automaton and the famous Turing machine, respectively. After studying the fundamental properties of these three models, we shift our focus to what these models are capable of solving, and we eventually reach the edges of computability itself by investigating undecidable problems.

This book is suitable for courses on the theory of computing at both the undergraduate and graduate levels, and for self-study. It is published under a Creative Commons BY-SA license and is available for free, forever.

Taylor J. Smith is an assistant professor in the Department of Computer Science and the director of the Formal Languages and Automata Research Lab at St. Francis Xavier University. His research and teaching specialties are in formal languages and automata theory. He earned his PhD in 2021 from Queen's University.

creativecommons.org