MORGAN & CLAYPOOL PUBLISHERS

Finite-State Text Processing

Kyle Gorman
Richard Sproat

Synthesis Lectures on Human Language Technologies

Graeme Hirst, *Series Editor*

Finite-State Text Processing

Synthesis Lectures on Human Language Technologies

Editor
Graeme Hirst, ***University of Toronto***

Synthesis Lectures on Human Language Technologies is edited by Graeme Hirst of the University of Toronto. The series consists of 50- to 150-page monographs on topics relating to natural language processing, computational linguistics, information retrieval, and spoken language understanding. Emphasis is on important new techniques, on new applications, and on topics that combine two or more HLT subfields.

Finite-State Text Processing
Kyle Gorman and Richard Sproat
2021

Embeddings in Natural Language Processing: Theory and Advances in Vector Representations of Meaning
Mohammad Taher Pilehvar and Jose Camacho-Collados
2020

Conversational AI: Dialogue Systems, Conversational Agents, and Chatbots
Michael McTear
2020

Natural Language Processing for Social Media, Third Edition
Anna Atefeh Farzindar and Diana Inkpen
2020

Statistical Significance Testing for Natural Language Processing
Rotem Dror, Lotem Peled, Segev Shlomov, and Roi Reichart
2020

Deep Learning Approaches to Text Production
Shashi Narayan and Claire Gardent
2020

Linguistic Fundamentals for Natural Language Processing II: 100 Essentials from Semantics and Pragmatics
Emily M. Bender and Alex Lascarides
2019

Cross-Lingual Word Embeddings
Anders Søgaard, Ivan Vulić, Sebastian Ruder, Manaal Faruqui
2019

Bayesian Analysis in Natural Language Processing, Second Edition
Shay Cohen
2019

Argumentation Mining
Manfred Stede and Jodi Schneider
2018

Quality Estimation for Machine Translation
Lucia Specia, Carolina Scarton, and Gustavo Henrique Paetzold
2018

Natural Language Processing for Social Media, Second Edition
Atefeh Farzindar and Diana Inkpen
2017

Automatic Text Simplification
Horacio Saggion
2017

Neural Network Methods for Natural Language Processing
Yoav Goldberg
2017

Syntax-based Statistical Machine Translation
Philip Williams, Rico Sennrich, Matt Post, and Philipp Koehn
2016

Domain-Sensitive Temporal Tagging
Jannik Strötgen and Michael Gertz
2016

Linked Lexical Knowledge Bases: Foundations and Applications
Iryna Gurevych, Judith Eckle-Kohler, and Michael Matuschek
2016

Bayesian Analysis in Natural Language Processing
Shay Cohen
2016

Metaphor: A Computational Perspective
Tony Veale, Ekaterina Shutova, and Beata Beigman Klebanov
2016

Grammatical Inference for Computational Linguistics
Jeffrey Heinz, Colin de la Higuera, and Menno van Zaanen
2015

Automatic Detection of Verbal Deception
Eileen Fitzpatrick, Joan Bachenko, and Tommaso Fornaciari
2015

Natural Language Processing for Social Media
Atefeh Farzindar and Diana Inkpen
2015

Semantic Similarity from Natural Language and Ontology Analysis
Sébastien Harispe, Sylvie Ranwez, Stefan Janaqi, and Jacky Montmain
2015

Learning to Rank for Information Retrieval and Natural Language Processing, Second Edition
Hang Li
2014

Ontology-Based Interpretation of Natural Language
Philipp Cimiano, Christina Unger, and John McCrae
2014

Automated Grammatical Error Detection for Language Learners, Second Edition
Claudia Leacock, Martin Chodorow, Michael Gamon, and Joel Tetreault
2014

Web Corpus Construction
Roland Schäfer and Felix Bildhauer
2013

Recognizing Textual Entailment: Models and Applications
Ido Dagan, Dan Roth, Mark Sammons, and Fabio Massimo Zanzotto
2013

Linguistic Fundamentals for Natural Language Processing: 100 Essentials from Morphology and Syntax
Emily M. Bender
2013

Semi-Supervised Learning and Domain Adaptation in Natural Language Processing
Anders Søgaard
2013

Semantic Relations Between Nominals
Vivi Nastase, Preslav Nakov, Diarmuid Ó Séaghdha, and Stan Szpakowicz
2013

Computational Modeling of Narrative
Inderjeet Mani
2012

Natural Language Processing for Historical Texts
Michael Piotrowski
2012

Sentiment Analysis and Opinion Mining
Bing Liu
2012

Discourse Processing
Manfred Stede
2011

Bitext Alignment
Jörg Tiedemann
2011

Linguistic Structure Prediction
Noah A. Smith
2011

Learning to Rank for Information Retrieval and Natural Language Processing
Hang Li
2011

Computational Modeling of Human Language Acquisition
Afra Alishahi
2010

Introduction to Arabic Natural Language Processing
Nizar Y. Habash
2010

Cross-Language Information Retrieval
Jian-Yun Nie
2010

Automated Grammatical Error Detection for Language Learners
Claudia Leacock, Martin Chodorow, Michael Gamon, and Joel Tetreault
2010

Data-Intensive Text Processing with MapReduce
Jimmy Lin and Chris Dyer
2010

Semantic Role Labeling
Martha Palmer, Daniel Gildea, and Nianwen Xue
2010

Spoken Dialogue Systems
Kristiina Jokinen and Michael McTear
2009

Introduction to Chinese Natural Language Processing
Kam-Fai Wong, Wenjie Li, Ruifeng Xu, and Zheng-sheng Zhang
2009

Introduction to Linguistic Annotation and Text Analytics
Graham Wilcock
2009

Dependency Parsing
Sandra Kübler, Ryan McDonald, and Joakim Nivre
2009

Statistical Language Models for Information Retrieval
ChengXiang Zhai
2008

Copyright © 2021 by Morgan & Claypool

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means—electronic, mechanical, photocopy, recording, or any other except for brief quotations in printed reviews, without the prior permission of the publisher.

Finite-State Text Processing

Kyle Gorman and Richard Sproat

www.morganclaypool.com

ISBN: 9781636391137 paperback
ISBN: 9781636391144 ebook
ISBN: 9781636391151 hardcover

DOI 10.2200/S01086ED1V01Y202104HLT050

A Publication in the Morgan & Claypool Publishers series
SYNTHESIS LECTURES ON HUMAN LANGUAGE TECHNOLOGIES

Lecture #50
Series Editor: Graeme Hirst, *University of Toronto*
Series ISSN
Print 1947-4040 Electronic 1947-4059

Finite-State Text Processing

Kyle Gorman
Graduate Center, City University of New York

Richard Sproat
Google LLC

SYNTHESIS LECTURES ON HUMAN LANGUAGE TECHNOLOGIES #50

MORGAN & CLAYPOOL PUBLISHERS

ABSTRACT

Weighted finite-state transducers (WFSTs) are commonly used by engineers and computational linguists for processing and generating speech and text. This book first provides a detailed introduction to this formalism. It then introduces Pynini, a Python library for compiling finite-state grammars and for combining, optimizing, applying, and searching finite-state transducers. This book illustrates this library's conventions and use with a series of case studies. These include the compilation and application of context-dependent rewrite rules, the construction of morphological analyzers and generators, and text generation and processing applications.

KEYWORDS

automata, finite automata, finite-state automata, finite-state transducers, grammar development, language processing, speech processing, state machines, text generation, text processing, Python, Pynini

Contents

Preface

This book is our attempt to provide a "one-stop" reference for engineers and linguists interested in using finite-state technologies for text generation and processing. As such, it begins with formal language and automata theory, topics covered in much greater detail by textbooks such as Hopcroft et al. 2008 and handbook chapters such as Mohri 2009. In our experience, full command of finite-state technologies requires familiarity with a number of matters that have not received much attention in prior literature. Among these topics is the theory of semirings, and algorithms specific to weighted automata such as the shortest-distance and shortest-path algorithms. These formalisms and algorithms are key for finite-state speech recognition. Furthermore, there exist many text processing applications that resemble weighted finite-state-based speech recognition insofar as hypotheses—that is, possible output strings—are represented as paths through a lattice constructed via composition of weighted automata, and inference/decoding involves computing the shortest path.

Users interested in text applications also stand to benefit from lesser-known "tricks of the trade" for finite-state development. These tricks include fuzzy string matching (Figure 7.1), efficient algorithms for optimizing arbitrary weighted finite-state transducers (section 4.1), compiling rewrite rules (section 5.2) and morphological analyzers and generators (chapter 6), and applying these transducers to sets of strings (section 5.3).

At the same time, we wish to go beyond algebraic formalisms and pseudocode. Thus, we illustrate our examples with Pynini, an open-source Python library for weighted finite-state transducers developed at Google. Still, we are skeptical that anything made out of dead trees is an appropriate medium for documenting a rapidly changing software library. So whereas earlier texts like *Finite State Morphology* (Beesley and Karttunen 2003) are in some sense *about* the Xerox finite-state toolkit as it existed at the time, we hope that this is not merely a book about Pynini. It is our hope that this melange of formalisms and algorithms, code and applications, meets the needs of our readers.

Finally, in the current age we would be remiss if we did not stress the importance of ethical use of this—or indeed any—technology. Ten years ago, Sproat (2010a:255) pointed out the potential dangers for society of language technology and its misuse, especially on social media platforms, noting that "language can be abused, and so can the technology that supports it". The recent rise in disinformation on social media has unfortunately made those concerns seem all too prophetic. The ongoing pandemic, aggravated in large part by disinformation, has brought these dangers into even starker relief. It is therefore our profound hope that the technology described in this book only be used for the betterment of humankind. One example of this sort suggests itself: Markov et al. (2021) describe how regular expression matching is used

to determine whether a post on social media mentions COVID-19 so it can be screened for disinformation.

Kyle Gorman and Richard Sproat
April 2021

Acknowledgments

We first owe an enormous debt to the many Google engineers who have contributed over the years to the OpenFst and OpenGrm libraries, particularly Cyril Allauzen, Brian Roark, Michael Riley, and Jeffrey Sorensen. Substantial improvements to the Pynini library have been made by Lawrence Wolf-Sonkin, and this book has greatly benefited from the user community of Google linguists, especially Sandy Ritchie. Thanks to Anssi Yli-Jyrä and an anonymous reviewer for their detailed reviews; to Jeffrey Heinz for detailed feedback on our pre-final draft; to Alëna Aksënova, Hossep Dolatian, Jordan Kodner, Constantine Lignos, Fred Mailhot, and Arya McCarthy, who provided useful comments on early drafts of the book; and to Chandan Narayan for notes on Pāṇini.

Kyle Gorman and Richard Sproat
April 2021

CHAPTER 1

Finite-State Machines

This is a book about **weighted finite-state transducers** (WFSTs) and their use in text generation and processing. The WFST formalism synthesizes decades of research into graphs, automata, and formal languages, including lines of research blossoming long before the era of ubiquitous digital computing.

The history of finite-state technology stretches back almost a century. Some key theorems and algorithms were discovered—and rediscovered—long before computers became powerful enough to exploit them (see chapter 5 for an example) and in some cases decades have elapsed between discovery and software implementation. Some essential algorithms were not generalized until the 1990s or later, as part of efforts—particularly at AT&T Bell Labs, and later at Google—to use WFSTs for scalable automatic speech recognition and text-to-speech synthesis.

A few key notions connect these disparate areas of research and application. The first is that of the **state machine**, a sort of abstract mathematical model of computation of which weighted finite-state transducers are a special case. Such models, first formalized by Turing (1936), are not only the foundation of the theory of computation—quite literally, the study of what it means to compute—but also inspired the creation of ENIAC, the first general-purpose digital computer, a decade later. The second is that of **formal languages**. While the origins of formal language theory can be traced at least as far back as Thue (1914), perhaps the most important contribution is a study by Kleene (1956) first circulated in 1951. Kleene's study springs from an obscure goal: the formal characterization of the expressive capacity of "nerve nets", a primitive form of artificial neural network proposed by McCulloch and Pitts (1943) a few years prior. To do so, Kleene introduces a family of formal languages called the "regular languages" and established strong connections between the algebraic characterizations of formal language theory and the automata (i.e., state machine) characterizations used by Turing and others. This body of work was an enormous inspiration in the development of modern linguistic theory—generative grammar in particular (Chomsky 1963)—and also contributed to the theory of compilers, computer programs which translate other computer programs. This chapter traces these two threads—automata and formal languages—and their relationship.

All of this effort, by some of the greatest scientific minds of the early 20th century, could easily have come to naught had the objects of study—regular languages and finite-state automata—turned out to have limited real-world relevance. But it turns out that these exhibit tantalizing similarities to phenomena found in natural—that is, human—languages, a fact which has only become clearer with time. A few examples should suffice. It is now believed that vir-

tually all patterns that define the phonology—or the grapheme-to-phoneme rules—of natural languages can be expressed as relations between regular languages. The hypothesis space of automatic speech recognizers, consisting of a probabilistic mapping between acoustic observations and word sequences, can also be compactly expressed as a relation between two regular languages. Finally, many text generation and processing problems can be framed as transductions between regular languages. Thanks to Kleene and others, it is known that these types of relations can be encoded by state machines, and subsequent work introduces techniques for combining, applying, optimizing, and searching these machines.

1.1 STATE MACHINES

A **state machine** is hardware or software whose behavior can be described solely in terms of a set of **states** and **arcs**, which represent transitions between those states. In this formalism, states roughly correspond to "memory" and arcs to "operations" or "computations". State machines are examples of what computer scientists call **directed graphs**.[1] These are "directed" in the sense that the existence of an arc from state q to state r does not imply an arc from r to q. A **finite-state machine** is merely a state machine with a finite, predetermined set of states and labeled arcs.

One familiar example of a state machine—encoded in hardware, rather than software—is the old-fashioned gumball machine (Figure 1.1). Such machines can be in any one of three states at a time, and each state is associated with actions such as

- turning the knob,
- inserting a coin, or
- emitting a gumball.

At one state, arbitrarily called state 0, it is possible to turn the knob, but this has no effect on the behavior of the machine. If, on the other hand, one inserts the appropriate coin(s), that transitions the machine to a state 1, at which point a subsequent turn of the knob will cause the machine to emit a gumball and return to state 0. This of course is an idealization of real-world gumball machines, which may experience mechanical failure or run out of gumballs. Without a shop-keeper around to service the machine, model and reality necessarily diverge.

The description of the gumball machine above is given a graphical representation in Figure 1.2. By convention, the bold outline of state 0 indicates that it has been—arbitrarily—chosen as the **start** or **initial state**; the double-struck outline indicates that it is also a **final state**; these notions will be formalized shortly. Valid transitions between states are indicated with arrows. These arcs are labeled with pairs of actions. Here, the inputs are user actions and the outputs are gumballs. The Greek letter ϵ ("epsilon") is used to represent the absence of an input and/or

[1] The primary difference is terminological; what are here called **states** and **arcs** are known in other communities as "vertices" and "edges", respectively.

Figure 1.1: An old-fashioned gumball machine. (Image credit: Dario Lo Presti/Shutterstock.com)

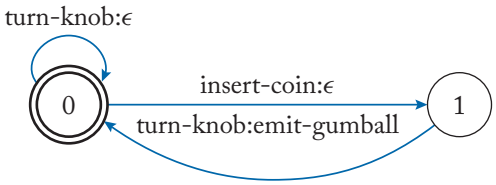

Figure 1.2: An old-fashioned gumball machine schematized as a state machine.

output for a given arc. Because, as mentioned, turning the knob at state 0 produces no output and does not change the state of the machine, there is a self-arc at state 0 labeled `turn-knob:`ϵ. On the other hand, inserting a coin at state 0 produces no observable output, but it transitions the machine to state 1. At this state a knob turn by the user causes the machine to emit a gumball and return to state 0.

We now provide definitions for various types of finite-state machine, after reviewing some formal preliminaries.

1.2 FORMAL PRELIMINARIES

This section provides a brief introduction to set theory and related topics. Those readers already familiar with sets, relations, functions, strings, and languages are welcome to skip to section 1.3.

1.2.1 SETS

Sets are abstract, unordered collections of distinct objects. They are an abstract, purely logical notion, and their definition does not presuppose any particular method of representing them in hardware or software; they are unordered in the sense that there is no natural ordering among the **elements** or **members** of any set. By convention, sets are represented using uppercase Greek or Italic letters, and elements of sets are denoted using lowercase Italic letters. Set membership is indicated using the $\in$ symbol, e.g., $x \in X$ is read "x is a member of X". Non-membership is written using the $\notin$ symbol, e.g., $x \notin X$ is read "x is not a member of X".

Members of a set can be any type of object, including other sets. There are several ways to specify the members of a set. First, for finite sets, one can simply list the elements in the set enclosed in curly braces, a representation called **extensional** or **list notation**. For instance, $\{2, 3, 5, 7\}$ is the finite set of prime numbers less than 10. An alternative notation, and the only one which can be used to denote infinite sets, uses a predicate such that if some element satisfies the predicate, that element is a member of a set; this is known as **intensional**, **predicate**, **set-builder**, or **set-former** notation. For instance, one might indicate the infinite set of prime numbers using the notation $\{x \mid \text{prime}(x)\}$. Finally, special notation is used for the **empty set**, the set with no elements: it is written $\emptyset$. The **cardinality** of a set X, written $|X|$, is the number of distinct elements in the set.

A set X is said to be a **subset** of another set Y if every element in X is also a member of Y. This property is written using the $\subseteq$ operator, e.g., $X \subseteq Y$ is read "X is a subset of Y". X is a **proper subset** of Y $(X \subset Y)$ if and only if X is a subset of Y and $X \neq Y$.

There are various logical operations over sets. Given two sets X and Y, their **intersection** $X \cap Y$ is the set that contains all elements which are members of both X and Y: that is, $X \cap Y = \{z \mid z \in X \wedge z \in Y\}$ where $\wedge$ represents logical AND. Given two sets X and Y, their **union** $X \cup Y$ is the set that contains all elements which are members of X, Y, or both: that is, $X \cup Y = \{z \mid z \in X \vee z \in Y\}$ where $\vee$ represents logical OR. Finally, their **difference** $X - Y$ is the set that contains all elements which are members of X but not of Y: that is, $X - Y = \{z \mid z \in X \wedge z \notin Y\}$.

1.2.2 RELATIONS AND FUNCTIONS

A **pair** or **two-tuple** is a sequence of two elements, e.g., (a, b) is the pair consisting of a then b. This is used to define an operation over sets known as the **cross-product** or **Cartesian product**. Given two sets X and Y, their cross-product $X \times Y$ is the set containing all ordered pairs (x, y) where x is an element of X and Y is an element of Y. That is, $X \times Y = \{(x, y) \mid x \in X \wedge y \in Y\}$.

A **relation**—specifically, a **binary** or **two-way relation**—over sets X and Y is a subset of the cross-product $X \times Y$. In this book, relations are indicated using lowercase Greek letters, and the **domain**—set of inputs—and **range** (or more properly, the **codomain**)—the set of outputs—are usually provided upon first definition. For instance, the expression $\gamma \subseteq X \times Y$ indicate that γ is a relation with domain X and range Y. Relations represent mappings between elements of the domain and elements of the range; for instance, the "less than" relation can be written $\lambda \subseteq \mathbb{R} \times \mathbb{R} = \{(x, y) \mid x < y\}$ where $\mathbb{R}$ is the set of real numbers.

A **function** is a relation for which every element of the domain is associated with exactly one element of the range. The "less than" relation above is not a function because, for example, there are an infinitude of real numbers that are less than any other real number. However, the "successor" relation $\sigma \subseteq \mathbb{N} \times \mathbb{N} = \{(x, x + 1) \mid x \in \mathbb{N}\}$, where $\mathbb{N}$ is the set of natural numbers, is a function, because each natural number has exactly one successor.

Three-, four-, and five-way relations, and so on, are all well-defined, though there is no such generalization for functions, since n-way relations where $n > 2$ lack well-defined domain and range. However, one can redefine any n-way relation into a two-way relation by grouping the various sets into domain and range; for instance, a four-way relation over $A \times B \times C \times D$ can be redefined as a two-way relation (and possibly, a function) with domain $A \times B$ and range $C \times D$. Such a relation might be defined as a subset of $A \times B \to C \times D$, with the arrow used to indicate the partition into domain and range.

The application of an input argument to a relation or function can be indicated using square brackets. For instance, given the successor function σ, then $\sigma[3] = \{4\}$ because $(3, 4) \in \sigma$.

Given a relation $\gamma \subseteq X \times Y$ and $x \in X$, $\gamma[x] \downarrow$ indicates that γ is well defined at x and $\gamma[x] \uparrow$ indicates that γ is undefined at x. A relation or function is said to be **total** if it is defined for all values of the domain. The less-than relation and successor functions, for example, are both total.

1.2.3 STRINGS AND LANGUAGES

Many of the sets defined below contain a type of element known as a string. Let Σ be a set of symbols called the **alphabet**. A **string** is a finite ordered sequence of zero or more elements from the alphabet. By convention, the empty string is indicated by ϵ. Note that ϵ is not a member of Σ.

The **concatenation** of two strings is the string produced by joining the two strings end-to-end. The concatenation of two strings x, y is written xy. Note that ϵ is the concatenative identity, thus $x\epsilon = \epsilon x = x$ for all x.

A set of zero or more strings is known as a **language**.[2] Since languages are sets, operations such as intersection, union, and difference are well defined. In addition, concatenation can also be generalized to languages, i.e., given languages X and Y, $XY = \{xy \mid x \in X \wedge y \in Y\}$. One other operation over languages is closure. First, the notation X^n, where n is a natural number, denotes a language consisting of n self-concatenations of X; e.g., $X^0 = \{\epsilon\}$ and $X^4 = XXXX$. The (**concatenative**) **closure** of a language X is an infinite union of zero or more concatenations of X with itself. It is notated with a superscripted asterisk, e.g., $X^* = \bigcup_{i \geq 0} X^i = \{\epsilon\} \cup X \cup XX \cup XXX \cup \ldots$. One variant of closure, indicated with a superscript plus-sign, excludes the empty string, e.g., $X^+ = \bigcup_{i > 0} X^i = X \cup XX \cup XXX \cup \ldots$, or equivalently, $X^+ = XX^*$. These two variants of closure are sometimes referred to as **Kleene star** and **Kleene plus**, respectively. Finally, a superscripted question mark is used to indicate optionality, e.g., $X^? = \{\epsilon\} \cup X$.

1.3 ACCEPTORS AND REGULAR LANGUAGES

Finite acceptors are the simplest form of finite automata, in some ways simpler than the model of a gumball machine presented above. They represent a family of string sets known as the regular languages.

1.3.1 FINITE-STATE ACCEPTORS

A **finite-state acceptor** (FSA) is a five-tuple consisting of

1. a finite set of states Q,
2. a **start** or **initial state** $s \in Q$,
3. a set of **final** (or **accepting**) **states** $F \subseteq Q$,
4. an **alphabet** Σ, and
5. a **transition relation** $\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$.

Note that as formalized here, there is only one start state but there may be many final states; also note that the start state may itself be a final state.[3]

An FSA is said to accept a string if there exists a path from the initial state to some final state, and the labels of the arcs traversed by that path correspond to the string in question. The set of all strings so accepted by an FSA is called its language. More formally, given two states $q, r \in Q$ and a symbol $z \in \Sigma \cup \{\epsilon\}$, $(q, z, r) \in \delta$ implies that there is an arc from state q to state r with label z. A **path** through a finite acceptor is a pair of

[2] This is not intended to supplant common-sense notions of what a language is; it is merely a term of art.

[3] One could allow for arbitrarily many start states, but given any finite automaton with multiple start states $S \subseteq Q$, it is trivial to construct an equivalent automaton with a single "superinitial" start state. Alternatively, one could limit the formalism to a single "superfinal" final state $f \in Q$.

1. a state sequence $q_1, q_2, \ldots, q_n \in Q^n$ and a
2. a string $z_1, z_2, \ldots, z_n \in (\Sigma \cup \{\epsilon\})^n$,

subject to the constraint that $\forall i \in [1, n] : (q_i, z_i, q_{i+1}) \in \delta$; that is, there exists an arc from q_i to q_{i+1} labeled z_i. A path that visits a state more than one time—i.e., if its state sequence contains the start state s or any repeated states—has a **cycle**. Automata are **cyclic** if any of their paths contain cycles and **acyclic** otherwise.

A path is said to be **complete** if

1. $(s, z_1, q_1) \in \delta$ and
2. $q_n \in F$.

That is, a complete path must also begin with an arc from the initial state s to q_1 labeled z_1 and terminate at a final state. Henceforth, without loss of generality, ϵ-labels are omitted from path strings because ϵ signals the absence of a symbol and therefore can be ignored. Indeed, for every FSA, there is an equivalent **ϵ-free** FSA, i.e., an FSA which accepts the same language but which has no ϵ-arcs, computed with the ϵ-removal algorithm (Mohri 2002a). Then, an FSA **accepts** or **recognizes** a string $z \in \Sigma^*$ if there exists a complete path with string z. The set of strings accepted by an FSA is called its language.

1.3.2 REGULAR LANGUAGES

The family of languages recognized by finite acceptors are the **regular languages**. Kleene (1956) provides an algebraic characterization. Given an alphabet Σ:

1. The empty language $\emptyset$ is a regular language.
2. The empty string language $\{\epsilon\}$ is a regular language.
3. If $s \in \Sigma$, then the singleton language $\{s\}$ is a regular language.
4. If X is a regular language, then its closure X^* is a regular language.
5. If X, Y are regular languages, then:
   - their concatenation XY is a regular language, and
   - their union $X \cup Y$ is a regular language.
6. Languages which cannot be derived as above are not regular languages.

Kleene (ibid.) also shows that every finite acceptor corresponds to a regular language and that every regular language corresponds to a finite acceptor. This result, known as **Kleene's theorem**, implies that operations over languages such as closure, concatenation, and union are defined not only for languages but also for finite acceptors.

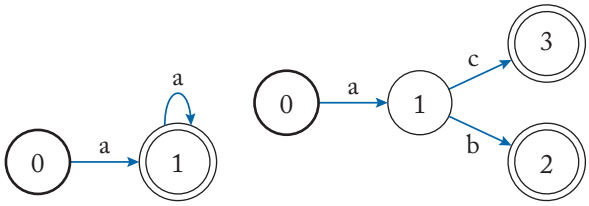

Figure 1.3: Finite acceptors for the languages $\{\mathtt{a}\}^+$ (left) and $\mathtt{a}(\mathtt{b} \cup \mathtt{c})$ (right).

Two examples of FSAs and their corresponding regular languages are shown in Figure 1.3 as **state transition diagrams**. The left pane contains an FSA defined by $Q = \{0, 1\}$, $s = 0$, $F = \{1\}$, $\Sigma = \{\mathtt{a}\}$, and $\delta = \{((0, \mathtt{a}), 1), ((1, \mathtt{a}), 1)\}$, which accepts the infinite language $\{\mathtt{a}\}^+ = \{\mathtt{a}, \mathtt{aa}, \mathtt{aaa}, \ldots\}$. The right pane shows an FSA which accepts the finite language $\mathtt{a}(\mathtt{b} \cup \mathtt{c}) = \{\mathtt{ab}, \mathtt{ac}\}$. The reader is encouraged to study these acceptors and manually trace the generation of a few strings.

1.3.3 REGULAR EXPRESSIONS

Regular expressions are a declarative notational scheme used to characterize the regular languages (Hopcroft et al. 2008: ch. 3). One can convert any finite acceptor to a regular expression, and any regular expression to a finite automaton. However, implementations of regular expressions in many programming languages—for instance, the one implementation used in Python's built-in `re` module—include additional features which cannot be encoded using regular languages or finite-state acceptors.

1.4 TRANSDUCERS AND RATIONAL RELATIONS

Finite transducers are a generalization of finite acceptors. Rather than modeling languages, they model **rational relations** between pairs of languages, and as such they can be used to encode string-to-string transductions.[4]

1.4.1 FINITE-STATE TRANSDUCERS

A **finite-state transducer** (FST) is a six-tuple consisting of

1. a finite set of states Q,
2. a start state $s \in Q$,
3. a set of final states $F \subseteq Q$,

[4] It is possible to generalize rational relations, and finite-state transducers, to relations between sets of more than two languages. This generalization is not discussed here as it is only rarely employed in computational linguistics, but see, e.g., Kay 1987, Kiraz 2001, or Hulden 2017.

4. an **input alphabet** Σ,
5. an **output alphabet** Φ, and
6. a transition relation $\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times (\Phi \cup \{\epsilon\}) \times Q$.

The first three elements are also used in the definition of FSAs; the latter three are novel. The key distinction between FSAs and FSTs is that in the latter case, arcs bear pairs of labels, one drawn from an input alphabet and the other from a (possibly disjoint) output alphabet. A **path** through a finite transducer is a triple consisting of

1. a state sequence $q_1, q_2, \ldots, q_n \in Q^n$,
2. an input string $x_1, x_2, \ldots, x_n \in (\Sigma \cup \{\epsilon\})^n$, and
3. an output string $y_1, y_2, \ldots, y_n \in (\Phi \cup \{\epsilon\})^n$,

subject to the constraint that $\forall i \in [1, n] : (q_i, x_i, y_i, q_{i+1}) \in \delta$. A **complete path** is a path where

1. $(s, x_1, y_1, q_1) \in \delta$ and
2. $q_n \in F$.

That is, a complete path must also begin with a transition from the initial state s to q_i with input label x_i and output label y_i and halt in a final state. Without loss of generality, and once again ignoring the presence of ϵ, the domain Σ^* and range Φ^* of an FST are both themselves regular languages, and the FST itself can be interpreted as a relation, a subset of the cross-product $\Sigma^* \times \Phi^*$. Then, an FST **transduces** or **maps** from $x \in \Sigma^*$ to $y \in \Phi^*$ so long as a complete path with input string x and output string y exists. However, unlike FSAs, not all FSTs have an equivalent ϵ-free form. For example, consider an FST mapping from two-character U.S. state abbreviations (e.g., `OH`) to state names (`Ohio`); a fragment of such an FST is shown in Figure 1.4. Here, arcs with ϵ input labels are necessary to allow input strings which are shorter than the corresponding output strings. Note also that the ϵ-removal algorithm mentioned in subsection 1.3.1 removes ϵ-arcs—those which have ϵ as both input and output labels—not ϵ-labels in general.

1.4.2 RATIONAL RELATIONS

The family of string relations that can be encoded as a finite-state transducer are the **rational relations**. Like regular languages, closure, concatenation, and union are all well defined for rational relations. The rational relations are closed under these operations, meaning that the closure of a rational relation, or the concatenation or union of two or more rational relations, are also rational relations. However, there are other operations, such as difference, under which the regular languages are closed but the rational relations are not.

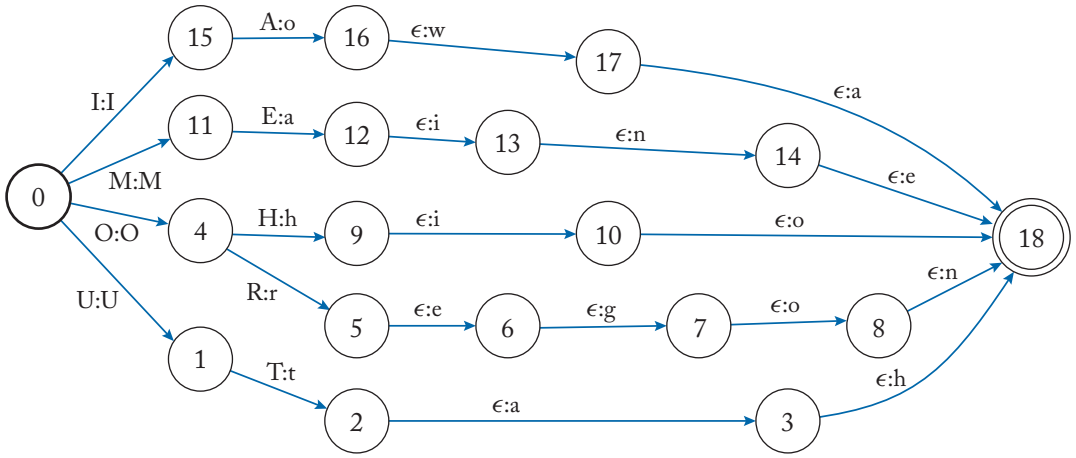

Figure 1.4: Fragment of a FST mapping from state abbreviations to state names.

Rational relations are closely related to, but distinct from, **regular expression substitutions** (e.g., as performed by Python's `re.sub` function).[5] On one dimension, regular expression substitutions are less expressive than rational relations, because the former permit many-to-many (rather than merely one-to-one and many-to-one) transductions, whereas the pattern matched by a `re.sub` is an arbitrary regular language, the substitution must be a single string. Neither finite state transducers nor the rational relations are restricted in this fashion. At the same time, `re.sub` implements other mechanisms that make it more expressive than rational relations.

1.5 WEIGHTED ACCEPTORS AND LANGUAGES

The above formalisms also permit an extension in which acceptors and transducers—and languages and relations—are generalized by attaching weights to states and arcs. These weights can represent virtually any set so long as the set and associated operations obey certain constraints decribed below. **Language models**, probability distributions over strings, can be compactly encoded as weighted acceptors (e.g., Allauzen et al. 2003, 2005, Roark et al. 2012); **hidden Markov models** can be encoded as weighted transducers (Roche and Schabes 1995) as can sequential **linear models** (Wu et al. 2014) and decoder graphs for automatic speech recognition engines (e.g., Mohri 1997, Mohri et al. 2002). Below, semirings are defined and exemplified and then used to generalize earlier definitions of automata, languages, and relations.

1.5.1 MONOIDS AND SEMIRINGS

Weighted automata algorithms are defined with respect to an algebraic system known as a semiring (Kuich and Salomaa 1986). It is first necessary to define a related notion, monoids.

[5] https://docs.python.org/3/library/re.html#re.sub

A **monoid**, is an ordered pair $(\mathbb{K}, \bullet)$ where $\mathbb{K}$ is a set and $\bullet$ is a binary operator over $\mathbb{K}$ with the properties of

1. **closure**: $\forall a, b \in \mathbb{K} : a \bullet b \in \mathbb{K}$,
2. **associativity**: $\forall a, b, c \in \mathbb{K} : (a \bullet b) \bullet c = a \bullet (b \bullet c)$, and
3. **identity**: $\exists e \in \mathbb{K} : e \bullet a = a \bullet e = a$.

A monoid is said to be **commutative** if $\forall a, b \in \mathbb{K} : a \bullet b = b \bullet a$. Then, a **semiring** is then a five-tuple $(\mathbb{K}, \oplus, \otimes, \bar{0}, \bar{1})$ such that

1. the pair $(\mathbb{K}, \oplus)$ form a commutative monoid with identity element $\bar{0}$,
2. the pair $(\mathbb{K}, \otimes)$ form a monoid with identity element $\bar{1}$,
3. $\forall a, b, c \in \mathbb{K} : a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$, and
4. $\forall a \in \mathbb{K} : a \otimes \bar{0} = \bar{0} \otimes a = \bar{0}$.

These constraints require that $\oplus$ is commutative, that $\bar{0}$ is the additive identity, that $\bar{1}$ is the multiplicative identity, that $\otimes$ distributes over $\oplus$, and that $\bar{0}$ is the multiplicative annihilator (i.e., that any weight multiplied with $\bar{0}$ is $\bar{0}$). Some common semirings are shown in Table 1.1. The **Boolean semiring** consists of true (1) and false (0) values and logical OR and AND operators. The **probability semiring** ranges over positive real numbers $\mathbb{R}_+$ and employs the expected $+$ and $\times$ arithmetic operations for $\oplus$ and $\otimes$.[6] The **log semiring** is the projection of the probability semiring onto the log domain.[7] The log semiring uses the logarithmic identity $\ln(xy) = \ln x + \ln y$ to replace multiplication with addition in the log domain; this helps to avoid arithmetic underflow when weight computations are performed with floating-point numbers. The definition of addition in this semiring is somewhat more complex: $\oplus = \oplus_{\log}$ where $a \oplus_{\log} b = -\ln(e^{-a} + e^{-b})$. Finally, the **tropical semiring** is identical to the log semiring except that $\oplus = \min$.[8]

A semiring is said to exhibit the **path property** (or to be a **path semiring**) if for all $a, b \in \mathbb{K} : a \oplus b \in \{a, b\}$. The tropical semiring has this property—the minimum of any two numbers must be one of those two numbers—as does the boolean semiring. Non-path semirings such as the probability semiring and log semirings define $\oplus$ in a way with common-sense arithmetic notions, making them suitable for applications that involve counting. One example of this is the expectation maximization algorithm, commonly used to learn free parameters of ASR models. In contrast, path semirings are used for decoding because the path property is required efficiently compute the shortest path(s) through weighted automata (section 4.3).

[6] For probabilities, only numbers between 0 and 1 inclusive make sense, but numbers in the range $(1, +\infty]$ serve as inverse elements.

[7] The OpenFst library use the natural logarithm, specifically.

[8] The tropical semiring is named in tribute to the late mathematician Imre Simon of the University of São Paulo. We note that São Paulo is just south of the Tropic of Capricorn, so "subtropical" would have been more apt.

Table 1.1: Some commonly used semirings for finite-state applications; $\mathbb{R}$ and $\mathbb{R}_+$ denote the real, and positive real, numbers, respectively.

	$\mathbb{K}$	$\oplus$	$\otimes$	$\bar{0}$	$\bar{1}$
Boolean	$\{0, 1\}$	$\vee$	$\wedge$	0	1
Probability	$\mathbb{R}_+$	$+$	$\times$	0	1
Log	$\mathbb{R} \cup \{\pm\infty\}$	$\oplus_{\log}$	$+$	$+\infty$	0
Tropical	$\mathbb{R} \cup \{\pm\infty\}$	min	$+$	$+\infty$	0

1.5.2 WEIGHTED FINITE ACCEPTORS

A **weighted finite-state acceptor** (WFSA) is an FSA in which weights are associated with arcs and states. It is defined by a six-tuple consisting of

1. a finite set of states Q,
2. a start state $s \in Q$,
3. a **semiring** $(\mathbb{K}, \oplus, \otimes, \bar{0}, \bar{1})$,
4. a **final weight function** $\omega \subseteq Q \times \mathbb{K}$,
5. an alphabet Σ, and
6. a **transition relation** $\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times \mathbb{K} \times Q$.

Three modifications have been made with respect to the earlier definition of FSAs in subsection 1.3.1 above. First, WFSAs are defined with respect to a particular semiring. Second, in place of the finite state set F there is a function ω which gives the **final weight** for each state. By convention, this is assumed to be a total function and a state $q \in Q$ is said to be non-final if $\omega(q) = \bar{0}$.[9] Third, the transition relation δ has been extended to include weights. A **path** through a weighted finite acceptor is a triple of

1. a state sequence $q_1, q_2, \ldots, q_n \in Q^n$,
2. a string $z_1, z_2, \ldots, z_n \in (\Sigma \cup \{\epsilon\})^n$, and
3. a weight sequence $k_1, k_2, \ldots, k_n \in \mathbb{K}^n$

subject to the constraint that $\forall i \in [1, n] : (q_i, z_i, k_i, q_{i+1}) \in \delta$. This constraint holds that there exists an arc from q_i to q_{i+1} that has the label z_i and weight k_i. A path is **complete** if

[9] Alternatively, one could define ω as a partial function in which $\omega[q] \downarrow$ if and only if state q is final.

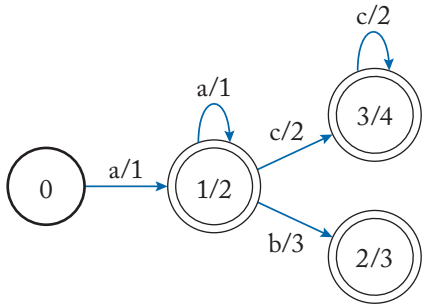

Figure 1.5: Weighted finite acceptor over the language $\{\mathtt{a}\}^+(\{\mathtt{b}\} \cup \{\mathtt{c}\}^*)$.

1. $(s, z_1, k_1, q_1) \in \delta$ and
2. $\omega[q_n] \neq \bar{0}$.

That is, a complete path must also begin with an arc from the initial state s to q_1 with label z_1 and weight k_1 and halt in a final state, i.e., a state with a non-$\bar{0}$ final weight. Once again ignoring ϵ-labels, a WFSA accepts a string $z \in \Sigma^*$ with weight

$$\left(\bigotimes_{i=1}^{n} k_i\right) \otimes \omega[q_n] = k_1 \otimes k_2 \otimes \ldots \otimes k_n \otimes \omega[q_n],$$

if there exists a complete path with string z and weight sequence $k_1, k_2, \ldots, k_n$. Note that the **path weight**, the weight associated with a path, is given by the $\otimes$-product of the weight sequence and the final weight of the final state in the path.

An example WFSA is shown in Figure 1.5; weights are separated from arc and/or state labels by a forward slash. This WFSA accepts the string `aacc`, for example, with weight $1 \otimes 1 \otimes 2 \otimes 2 \otimes 4$, equal to 10 in the log and tropical semirings.

1.5.3 WEIGHTED REGULAR LANGUAGES

There are two roughly equivalent ways to define the **weighted regular languages** expressed by weighted finite acceptors. Under one definition, a weighted language is a partial relation over $\Sigma^* \times \mathbb{K}$; that is, it assigns weights to those strings in its language. However, one can alternatively define weighted languages as a total relation with $\bar{0}$ used as the weight for strings not accepted under the previous definition. This eliminates the distinction between those strings not accepted by the language and those accepted with weight $\bar{0}$.

1.6 WEIGHTED TRANSDUCERS AND RELATIONS

Finite transducers and relations can also be extended to support weights.

1.6.1 WEIGHTED FINITE TRANSDUCERS

The definition of a **weighted finite-state transducer** (WFST) should be obvious from the preceding discussion, but is provided for completeness. A WFST is a seven-tuple consisting of

1. a finite set of states Q,
2. a start state $s \in Q$,
3. a semiring $(\mathbb{K}, \oplus, \otimes, \bar{0}, \bar{1})$,
4. a final weight function $\omega \subseteq Q \times \mathbb{K}$,
5. an input alphabet Σ,
6. an output alphabet Φ, and
7. a transition relation $\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times (\Phi \cup \{\epsilon\}) \times \mathbb{K} \times Q$.

Paths through a WFST are then four-tuples consisting of

1. a state sequence $q_1, q_2, \ldots q_n \in Q^n$,
2. a input string $x_1, x_2, \ldots, x_n \in (\Sigma \cup \{\epsilon\})^n$,
3. a output string $y_1, y_2, \ldots, y_n \in (\Phi \cup \{\epsilon\})^n$, and
4. a weight sequence $k_1, k_1, \ldots, k_n \in \mathbb{K}^n$

subject to the constraint that $\forall i \in [1, n] : (q_1, x_i, y_i, k_i, q_{i+1}) \in \delta$. A **complete path** is a path where

1. $(s, x_1, y_1, k_1, q_1) \in \delta$ and
2. $\omega[q_n] \neq \bar{0}$.

That is, a complete path must also begin with a transition from the initial state s to q_1 with input label x_1, output label y_1, and weight k_1, and halt in a final state. Once again, ignoring the presence of ϵ-labels in the input and output strings, a WFST **transduces** or **maps** from $x \in \Sigma^*$ to $y \in \Phi^*$ with weight $k \in \mathbb{K}$ so long as a complete path with path weight k, input string x, and output string y exists.

1.6.2 WEIGHTED RATIONAL RELATIONS

Each WFST corresponds to a **weighted rational relation**, a three-way partial relation over $\Sigma^* \times \Phi^* \times \mathbb{K}$, but in practice, such relations are often reinterpreted as two-way partial relations over $\Sigma^* \to \Phi^* \times \mathbb{K}$; that is, for a given input string, they yield pairs of an output string and an associated path weight. Weighted relations can alternatively be defined as total relations similarly to the alternative definition of weighted languages given in subsection 1.5.3.

FURTHER READING

Partee et al. (1993: ch. 1–3) give a gentle introduction to sets, pairs, relations, functions, and strings.

Hopcroft et al. (2008: ch. 2) formalizes finite acceptors, though they eschew both transducers and weights.

Comparable formalizations of WFSTs are given by Roark and Sproat (2007: ch. 1) and Mohri (2009).

Hopcroft et al. (2008: ch. 3) formalize connections between finite acceptors, regular languages, and regular expressions. Jurafsky and Martin (2009: ch. 2) and Eisenstein (2019: ch. 9) briefly discuss these connections.

Hopcroft et al. (2008: ch. 5–7) and Allauzen and Riley (2012) present an extension of finite automata known as **pushdown automata**, corresponding to the family of formal languages known as **context-free grammars** (Chomsky 1963).

CHAPTER 2

The Pynini Library

This chapter illustrates finite-state text processing using Pynini (Gorman 2016), an open-source Python library for finite-state text processing. Pynini is one of the two major libraries used for finite-state grammar development at Google. It marries efficient implementations of a wide variety of WFST algorithms with the convenience of the Python programming language. Pynini has seen wide adoption since its release; for example, it has been used to build text normalization grammars (Gorman and Sproat 2016, Ritchie et al. 2019) and grapheme-to-phoneme conversion models (Gorman et al. 2020, Lee et al. 2020) for dozens of languages.

The next section describes the basic design architecture of Pynini and related libraries. Readers less interested in these details are welcome to skip to section 2.2, which describes conventions used by Pynini. A basic familiarity with the Python language is assumed; readers who lack this familiarity should first consult one of the many textbooks on the subject.

2.1 DESIGN

Pynini, like several other finite-state toolkits, builds upon OpenFst (Allauzen et al. 2007), an open-source C++ library also developed at Google. The OpenFst library is an efficient, fast, and comprehensive general-purpose framework for WFST applications, and it has been used at Google and elsewhere to develop **automatic speech recognizers**, **text-to-speech synthesizers**, and **input method engines** (i.e., text entry systems for mobile devices), including those bundled with Android devices. At the lowest level, OpenFst provides classes—representing WFSTs—and functions—representing algorithms over WFSTs—templated on the semiring of the input FST(s). For instance, the following snippet contains a C++ template function which compiles an FSA from a string, placing each byte (i.e., `char`) on its own arc.

```
template <class Arc>
void CompileString(const std::string &s,
                   fst::VectorFst<Arc> *fst) {
  fst->DeleteStates();
  fst->AddStates(s.size() + 1);
  typename Arc::StateId state = 0;
  fst->SetStart(state);
  for (const char c : s) {
    fst->AddArc(state, Arc(c, c, state + 1));
    ++state;
```

```
  }
  fst->SetFinal(state);
}
```

In addition to templated functions and classes, OpenFst provides a second layer known as the scripting API. This layer does away with the template arguments using virtual dispatch for methods and a registration mechanism for functions. This allows the user to abstract away from the choice of semiring—at least for a set of pre-registered semirings—but is otherwise just as verbose as the lower layer. A Python extension module, `pywrapfst`, included with OpenFst, wraps the scripting API. This additional layer eliminates the need for compilation since Python is interpreted. Ignoring differences in syntax and naming conventions, a similar string compilation function, shown below, closely resembles the C++ version given above.[1]

```
def compile_string(s: bytes) -> pywrapfst.VectorFst:
    fst = pywrapfst.VectorFst()
    fst.add_states(len(s) + 1)
    state = 0
    fst.set_start(state)
    for c in s:
        fst.add_arc(state, pywrapfst.Arc(c, c, None, state + 1))
        state += 1
    fst.set_final(state)
    return fst
```

Pynini, a Python extension module, greatly simplifies many of the drudgeries of finite-state development. It includes several algorithms not included in OpenFst, including methods for converting between automata and strings (section 2.3), **range concatenation** (section 3.3) and **cross-product** (section 3.7) operators, general-purpose optimization routines (section 4.1), and **context-dependent rewrite rule compilation** (section 5.2), all key tools for finite-state grammar development. The built-in operations provided by Pynini largely eliminate the above snippets' low-level manipulation of a WFSTs' states and arcs in favor of high-level operators like composition, union, and so on.

2.2 CONVENTIONS

FSAs and FSTs can be thought of as subsets of WFSAs and WFSTs, respectively, whose weights are limited to $\{\bar{0}, \bar{1}\}$. By the same token, FSAs can be thought of as subsets of FSTs for which all transitions have the same input and output labels. Therefore, Pynini follows the practices of OpenFst, in using a single `Fst` type, a weighted transducer, for all four types of finite automata

[1] Throughout, Python functions are annotated with the optional type hints introduced in Python 3.5. Those unfamiliar or uncomfortable with these annotations are welcome to ignore them.

discussed in chapter 1. Thus, a weighted FSA is merely a WFST for which all input and output labels happen to match, and similarly, an unweighted transducer is one which only uses the "free" weight $\bar{1}$ and/or the "infinite" weight $\bar{0}$.

2.2.1 COPYING

The `Fst` class uses **copy-on-write** semantics, meaning that the `copy` method is constant time and only produce deep copies if one of the copies is later mutated. The same is true for `SymbolTable` objects discussed below.

2.2.2 LABELS

Arc input and output labels are represented by non-negative integers, with ϵ represented by label 0. One may use symbol tables to map between integer labels and strings for display, debugging, and string conversion (see section 2.3) but symbol tables are otherwise ignored. Negative-valued labels, while permitted, are reserved for implementation and should generally be avoided.

2.2.3 STATES

States are represented by dense sequences of integers—**state IDs**—ranging from 0 to $|Q| - 1$. As formalized in subsection 1.3.1, at most one state may be designated as the start state. An empty FST—one with no states—uses the constant `pynini.NO_STATE_ID` (equal to -1) as its start state. Thus, the following snippet asserts that an FST `f` is non-empty.

```
assert f.start() != pynini.NO_STATE_ID
```

Each state is associated with a final weight; non-final states have an infinite final weight $\bar{0}$ and final states have a non-$\bar{0}$ weight. Thus, the following snippet asserts that state `q` in an FST `f` is non-final.

```
assert f.final(q) == pynini.Weight.zero(f.weight_type())
```

2.2.4 ITERATION

Pynini does not provide random access to states and arcs; they must be accessed using specialized iterators. These iterators are invalidated—i.e., are no longer safe to use—in the following scenarios.

1. An `ArcIterator` is invalidated if any arcs leaving that state are mutated.

2. A `MutableArcIterator` is invalidated if arcs leaving any other state are mutated.

3. A `StateIterator` is invalidated if the number of states is changed.

2.2.5 WEIGHTS

`Fst` instances are associated with a given semiring and arc type. Pynini includes three built-in arc types.[2] The `standard` arc type, the default, gives the tropical semiring, with weights stored as 32-bit IEEE 754 floating-point numbers; it is commonly used to simulate the Boolean semiring. The `log` arc type gives the log semiring, once again using 32-bit floats. Finally, the `log64` arc type also uses the log semiring but uses "double-precision" 64-bit floating-point numbers. Pynini's `arcmap` function can be used to convert between semirings. For instance, the following snippet makes a deep copy of an FST `f` and converts it to the log semiring.

```
g = pynini.arcmap(f, map_type="to_log")
```

Finally, one can retrieve an `Fst`'s semiring using the `arc_type` and `weight_type` instance methods. For example, the following snippet asserts that `f` has the `standard` arc type and weights over the associated `tropical` semiring.

```
assert f.arc_type() == "standard"
assert f.weight_type() == "tropical"
```

2.2.6 PROPERTIES

Each `Fst` instance bears a set of **properties**, assertions about the FST's topology, weights, and so on. Some properties are binary—either true or false—whereas others are ternary—they also have an "unknown" value. Unknown property values are set when some operation invalidates the value of a property, but recomputing the true value of this property would be computationally expensive. Properties are stored in a single 64-bit unsigned integer, making them somewhat challenging to directly access. Each named property is represented by a module-level constant, a **property mask**. Some property masks are the bitwise union of multiple sets of related properties, and users can construct their own compound property masks using the bitwise OR operator `|`. There are two ways to test whether some FST has a given set of properties, but in both cases one passes the property mask to the instance method `properties`, and then compare the property mask to the properties this method returns. The second argument to the `properties` method is a boolean which specifies whether or not "unknown" properties are to be recomputed; depending on the property mask and the size of the FST, this recomputation may or may not be an expensive operation. When this argument is `True`, this tests whether the FST in question actually has the given property or properties; when it is `False`, it simply tests whether it is known to have the given property or properties. Some examples are given below.

- Asserts that `f` is cyclic:

  ```
  assert f.properties(pynini.CYCLIC, True) == pynini.CYCLIC
  ```

[2] One can recompile Pynini with support for additional semirings but this requires considerable C++ knowledge.

- Asserts that `f` is known to be cyclic:

```
assert f.properties(pynini.CYCLIC, False) == pynini.CYCLIC
```

- Asserts that `f` is an unweighted acceptor:

```
ua_props = pynini.ACCEPTOR | pynini.UNWEIGHTED
assert f.properties(ua_props, True) == ua_props
```

One can set FST properties using the `set_properties` method. The `verify` method tests whether an FST's properties are correct, as shown in the following snippet:

```
assert f.verify()
```

2.3 STRING CONVERSION

Finite-state automata represent sets of strings, and relations between strings. Naturally, then, finite-state grammar development requires one to convert strings into automata, or to extract strings from automata.

2.3.1 TEXT ENCODING

Imagine that one wishes to construct an automaton representing a single string. In a **string** or **chain automaton**,

1. the start state s is labeled 0,
2. the highest-numbered state is final and has no outgoing arcs, and
3. every other state q is non-final and has one outgoing arc to state $q + 1$.

String FSTs, by construction, have exactly one path and one (output) string. It is relatively straightforward to construct such an FST given a list of arc labels (and optionally, a final weight), but how does one convert a string to a list of arc labels, or inversely, a list of arc labels to a string?

Modern digital computers represent characters using low-precision integers, and strings as contiguous sequences of these integers. **Character encodings** define a bidirectional mappings between these numeric sequences and human-readable strings. Conversion from strings to number sequences is referred to as **encoding**, and from number sequences to strings as **decoding**. Character encodings predate digital computing by at least a century, having been used since the earliest days of telegraphy, and have even earlier roots in the cryptographic methods of antiquity. One of the most widely known character encodings is **ASCII** (the American Standard Code for Information Interchange), first published in 1963. ASCII defines a set of 128 distinct "characters". These include

- 26 uppercase Latin letters,

- 26 lowercase Latin letters,
- 10 Arabic numerals,
- 33 punctuation characters,
- and 33 **control characters**.

The control characters are used for a variety of functions such as delimiting the start of a new line, but some are obsolete telegraphy signals. The full ASCII table is shown in Table 2.1. Naturally, ASCII is only sufficient for English text, and does not support diaeresis (e.g., *Brontë*, *coöperate*, *Häagen-Dazs*, *Motörhead*, *naïve*) or other commonly used Latin-script diacritics.

ASCII is a 7-bit encoding scheme because it defines 128 ($= 2^7$) unique symbols. However, ASCII characters are usually stored in bytes, which have 256 ($= 2^8$) distinct values. This extra bit is exploited by **ISO/IEC 8859**, an encoding standard published incrementally from 1987–2001. Each of the 16 encodings in this standard adds up to 128 additional characters to ASCII. For example, the Part 1 encoding, sometimes called "Latin-1", covers most of the Latin scripts of Western Europe, although it lacks a handful of characters used in Catalan, Danish, Dutch, Estonian, Finnish, French, German, Hungarian, and Welsh; a later revision, Part 15, fills some of these gaps and adds the Euro sign €. Part 2 covers central European languages that use the Latin alphabet. Other ISO/IEC 8859 encodings cover Hebrew, Greek, and various European languages written in Cyrillic. However, the ISO/IEC 8859 encodings have several major limitations. First, they only cover a tiny number of the alphabetic scripts in existence, and with the exception of Thai, make no effort to cover the indigenous scripts of Asia. Second, they provide no mechanism for specifying which of the 16 encodings was used for a given document. It is often possible to guess or "sniff" a document's encoding (e.g., Li and Momoi 2001), though such methods are necessarily heuristic. Finally, they provide no mechanism for mixing or switching between scripts. Each ISO/IEC 8859 encoding is a superset of ASCII, so one can mix English and Cyrillic using the Part 5 encoding, for example, but there is no way to combine Cyrillic and French, or Greek and Hebrew, for example.

The Unicode Consortium, a non-profit organization incorporated in 1991, was founded to address the deficiencies of earlier encoding standards. The consortium, working in concert with tech companies and international standards organizations, has produced over a dozen versions of their standard, **Unicode**. Unicode defines a universal character set of over one million distinct codepoints. Roughly 140,000 of these codepoints are currently in use, covering 154 modern and historical scripts at the time of writing. It also contains a huge number of non-linguistic symbols including those used in linguistics, mathematics, and music, various geometric shapes and arrows, and **emoji**. Because Unicode is designed for backward compatibility with many earlier standards, there may be more than one Unicode representation for a given string. For example, diacriticized Latin characters like *é*, can either be a character in its own right or an *e* plus a combining acute diacritic. Similarly, in the Latin script used to write Serbo-Croatian,

Table 2.1: The ASCII encoding; control characters are indicated by angle brackets.

0	<NUL>	32		64	@	96	`
1	<SOH>	33	!	65	A	97	a
2	<STX>	34	"	66	B	98	b
3	<ETX>	35	#	67	C	99	c
4	<EQT>	36	$	68	D	100	d
5	<ENQ>	37	%	69	E	101	e
6	<ACK>	38	&	70	F	102	f
7	<BEL>	39	'	71	G	103	g
8	<BS>	40	(	72	H	104	h
9	<TAB>	41	)	73	I	105	i
10	<LF>	42	*	74	J	106	j
11	<VT>	43	+	75	K	107	k
12	<FF>	44	,	76	L	108	l
13	<CR>	45	-	77	M	109	m
14	<SO>	46	.	78	N	110	n
15	<SI>	47	/	79	O	111	o
16	<DLE>	48	0	80	P	112	p
17	<DC1>	49	1	81	Q	113	q
18	<DC2>	50	2	82	R	114	r
19	<DC3>	51	3	83	S	115	s
20	<DC4>	52	4	84	T	116	t
21	<NAK>	53	5	85	U	117	u
22	<SYN>	54	6	86	V	118	v
23	<ETB>	55	7	87	W	119	w
24	<CAN>	56	8	88	X	120	x
25	<EM>	57	9	89	Y	121	y
26	<SUB>	58	:	90	Z	122	z
27	<ESC>	59	;	91	[	123	{
28	<FS>	60	<	92	\	124	|
29	<GS>	61	=	93	]	125	}
30	<RS>	62	>	94	^	126	~
31	<US>	63	?	95	_	127	<DEL>

digraphs such as *dž*, *lj*, and *nj* are sometimes considered single characters, and in Korean, each hangul glyph represents a full syllable but can also be represented by their constituent jamo, each roughly a phoneme. In all three cases, Unicode supports both "composed" and "decomposed" representations of the above characters. For instance, *é* can be represented as a single codepoint (U+00E9) or as *e* (U+0065) followed by a combining acute accent (U+00E9). In Python, one can convert between various **Unicode normalization forms** using the `normalize` function from the built-in `unicodedata` module.[3]

Unicode defines several encodings for its character set. One, **UTF-32**, uses four bytes to represent all Unicode characters. However, this is massively inefficient, particularly for texts that consist primarily of Latin characters, and motivates an alternative encoding known as **UTF-8**. In this encoding, each character is encoded by one to four bytes, depending on the script. UTF-8 is backward-compatible with ASCII, in the sense that ASCII characters have the same encoding in both systems. Most non-ASCII Latin characters and characters from alphabets—Armenian, Greek, Cyrillic, etc.—require two bytes. Most of the remaining characters are encoded with three bytes; only rare East Asian characters, mathematical symbols, and historical scripts require four bytes to encode. At time of writing, UTF-8 is the most commonly used encoding, accounting for upward of 95% of all web pages at time of writing, and is the default encoding for most modern computers.

2.3.2 STRING COMPILATION

In Python 3, the `str` type represents a sequence of Unicode codepoints.[4] Pynini's `accep` function is used to compile a single Python `str` into a string `Fst`, an acyclic acceptor (in the sense of subsection 1.3.1) with a single final state and a single arc leaving each non-final state. It takes an input string argument and parses it according to the specified `token_type`, i.e., string parsing mode. Pynini provides three such modes. In all three modes, one can optionally specify the arc type (using the `arc_type` argument; this defaults to `"standard"`, i.e., the tropical semiring) or the final weight for the string (using the `weight` argument; this defaults to $\bar{1}$ in the appropriate semiring).

In `byte` mode, the default mode, the string is encoded as a UTF-8 string and each arc contains a single byte. In `utf8` mode, however, each arc contains a single Unicode codepoint (here specified in decimal).[5] The two modes are illustrated in Figure 2.1, which shows FSAs encoding a Unicode string in both `byte` and `utf8` mode. Note that since UTF-8 is a superset of ASCII, there is no distinction between `byte` and `utf8` modes for strings composed solely of ASCII characters.

Both `byte` and `utf8` modes use special conventions for interpreting the ASCII square bracket characters `[` and `]`. In the most common case, any string surrounded by `[` on its left

[3] https://docs.python.org/3/library/unicodedata.html#unicodedata.normalize
[4] This is a departure from Python 2, in which `str` is a bytestring.
[5] The somewhat unintuitive name for this mode comes from the use of UTF-8 as an intermediate encoding.

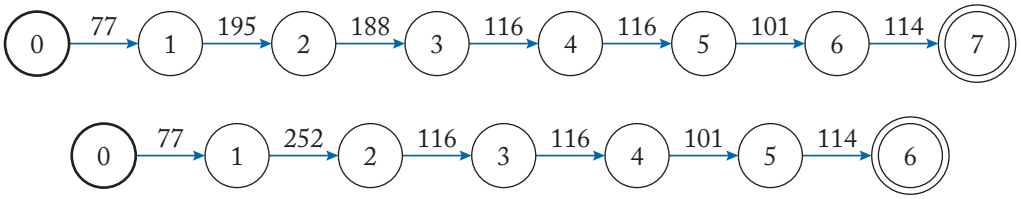

Figure 2.1: The German word *Mütter* 'mothers' compiled into a string in `byte` (above) and `utf8` (below) modes. In `byte` mode, *ü* is encoded using two arcs (labeled 195 and 188, the decimal representation of its UTF-8 encoding), but in `utf8` mode only one (252, the decimal representation of its Unicode codepoint) is needed.

and `]` on its right is taken to be an decimal or hexidecimal integer and is processed by the C standard library function `strtoll`. If this routine succeeds, the integer is used as an arc label and the enclosing square brackets are subsequentially ignored. For example, in both `byte` and `utf8` modes, strings `abc`, `[97][98][99]`, and `[0x61][0x62][0x63]` all give rise to the label sequence [97, 98, 99]. To prevent `[` and `]` from being interpreted as a delimiter for bracketed spans, rather than a literal character, one can "escape" them by adding a preceding backslash, and escaping can be automated using Pynini's `escape` function. The complete bracket parsing procedure is described in Appendix B.

The third mode, symbol table mode, is triggered by providing a `SymbolTable`, a bidirectional hash table mapping between strings and integers, as the `token_type` argument. In this mode, the string is assumed to consist of substrings separated by a space character, and each substring is assumed to be assigned to an integer label by the provided symbol table. Sample snippets are given below.

- Compiles string `s` in byte mode:

```
f = pynini.accep(s)
```

- Compiles string `s` in UTF-8 mode with final weight 2:

```
f = pynini.accep(s, token_type="utf8", weight=2)
```

- Compiles string `s` in symbol table mode using symbol table `sym`:

```
f = pynini.accep(s, token_type=sym)
```

Whenever possible, Pynini functions and methods that expect FST arguments will implicitly cast `str` instances to `Fst` by calling the `accep` function on the input string. This allows one to, for example, construct an FSA representing the union of a set of strings by passing string arguments to the `union` function (section 3.4) without explicitly compiling those strings first.