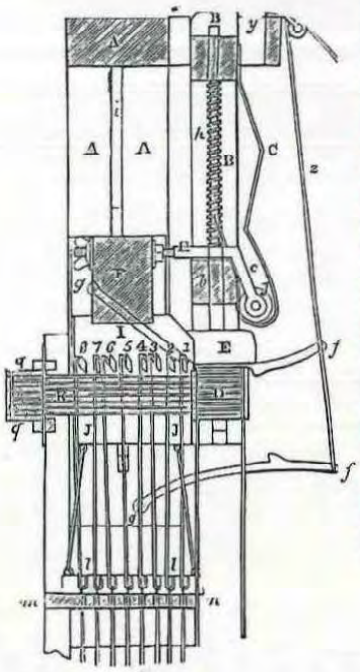
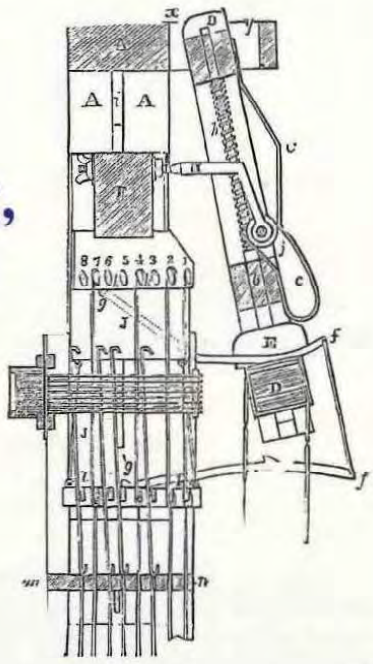


SPEECH and LANGUAGE PROCESSING

An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition



DANIEL JURAFSKY & JAMES H. MARTIN

Speech and Language Processing

"This book is an absolute necessity for instructors at all levels, as well as an indispensible reference for researchers. Introducing NLP, computational linguistics, and speech recognition comprehensively in a single book is an ambitious enterprise. The authors have managed it admirably, paying careful attention to traditional foundations, relating recent developments and trends to those foundations, and tying it all together with insight and humor. Remarkable."

– Philip Resnik, University of Maryland

"... ideal for ... linguists who want to learn more about computational modeling and techniques in language processing; computer scientists building language applications who want to learn more about the linguistic underpinnings of the field; speech technologists who want to learn more about language understanding, semantics and discourse; and all those wanting to learn more about speech processing. For instructors ... this book is a dream. It covers virtually every aspect of NLP... What's truly astounding is that the book covers such a broad range of topics, while giving the reader the depth to understand and make use of the concepts, algorithms and techniques that are presented... ideal as a course textbook for advanced undergraduates, as well as graduate students and researchers in the field.

– Johanna Moore, University of Edinburgh

"*Speech and Language Processing* is a comprehensive, reader-friendly, and up-to-date guide to computational linguistics, covering both statistical and symbolic methods and their application. It will appeal both to senior undergraduate students, who will find it neither too technical nor too simplistic, and to researchers, who will find it to be a helpful guide to the newly established techniques of a rapidly growing research field."

– Graeme Hirst, University of Toronto

"The field of human language processing encompasses a diverse array of disciplines, and as such is an incredibly challenging field to master. This book does a wonderful job of bringing together this vast body of knowledge in a form that is both accessible and comprehensive. Its encyclopedic coverage makes it a must-have for people already in the field, while the clear presentation style and many examples make it an ideal textbook."

– Eric Brill, Microsoft Research

This is quite simply the most complete introduction to natural language and speech technology ever written. Virtually every topic in the field is covered, in a prose style that is both clear and engaging. The discussion is linguistically informed, and strikes a nice balance between theoretical computational models, and practical applications. It is an extremely impressive achievement.

– Richard Sproat, AT&T Labs – Research

PRENTICE HALL SERIES
IN ARTIFICIAL INTELLIGENCE
Stuart Russell and Peter Norvig, Editors

GRAHAM	ANSI *Common Lisp*
RUSSELL & NORVIG	*Artificial Intelligence: A Modern Approach*
JURAFSKY & MARTIN	*Speech and Language Processing*

Speech and Language Processing

An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition

Daniel Jurafsky and James H. Martin

University of Colorado, Boulder

Contributing writers:
Andrew Kehler, Keith Vander Linden, and Nigel Ward

Prentice Hall
Upper Saddle River, New Jersey 07458

Library of Congress Cataloging-in-Publication Data

Jurafsky, Daniel S. (Daniel Saul)
Speech and Langauge Processing / Daniel Jurafsky, James H. Martin.
p. cm.
Includes bibliographical references and index.
ISBN 0-13-095069-6

Editor-in-Chief: *Marcia Horton*
Publisher: *Alan Apt*
Editorial/production supervision: *Scott Disanno*
Editorial assistant: *Toni Holm*
Executive managing editor: *Vince O'Brien*
Cover design director: *Heather Scott*
Cover design execution: *John Christiana*
Manufacturing manager: *Trudy Pisciotti*
Manufacturing buyer: *Pat Brown*
Assistant vice-president of production and manufacturing: *David W. Riccardi*

Cover design: *Daniel Jurafsky, James H. Martin, and Linda Martin.* The front cover drawing is the action for the Jacquard Loom (Usher, 1954). The back cover drawing is Alexander Graham Bell's Gallows telephone (Rhodes, 1929).

This book was set in Times-Roman, TIPA (IPA), and PMC (Chinese) by the authors using LaTeX2e.

Pearson Higher Education
Upper Saddle River, New Jersey 07458

Printed in the United States of America

10 9 8 7 6 5 4 3
ISBN 0-13-095069-6

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada, Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Prentice-Hall Asia Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

For my parents, Ruth and Al Jurafsky — D.J.

For Linda — J.M.

Summary of Contents

Contents

Foreword

Linguistics has a hundred-year history as a scientific discipline, and computational linguistics has a forty-year history as a part of computer science. But it is only in the last five years that language understanding has emerged as an industry reaching millions of people, with information retrieval and machine translation available on the internet, and speech recognition becoming popular on desktop computers. This industry has been enabled by theoretical advances in the representation and processing of language information.

Speech and Language Processing is the first book to thoroughly cover language technology, at all levels and with all modern technologies. It combines deep linguistic analysis with robust statistical methods. From the point of view of levels, the book starts with the word and its components, moving up to the way words fit together (or syntax), to the meaning (or semantics) of words, phrases and sentences, and concluding with issues of coherent texts, dialog, and translation. From the point of view of technologies, the book covers regular expressions, information retrieval, context free grammars, unification, first-order predicate calculus, hidden Markov and other probabilistic models, rhetorical structure theory, and others. Previously you would need two or three books to get this kind of coverage. *Speech and Language Processing* covers the full range in one book, but more importantly, it relates the technologies to each other, giving the reader a sense of how each one is best used, and how they can be used together. It does all this with an engaging style that keeps the reader's interest and motivates the technical details in a way that is thorough but not dry. Whether you're interested in the field from the scientific or the industrial point of view, this book serves as an ideal introduction, reference, and guide to future study of this fascinating field.

Peter Norvig & Stuart Russell, Editors
Prentice Hall Series in Artificial Intelligence

Preface

This is an exciting time to be working in speech and language processing. Historically distinct fields (natural language processing, speech recognition, computational linguistics, computational psycholinguistics) have begun to merge. The commercial availability of speech recognition and the need for Web-based language techniques have provided an important impetus for development of real systems. The availability of very large on-line corpora has enabled statistical models of language at every level, from phonetics to discourse. We have tried to draw on this emerging state of the art in the design of this pedagogical and reference work:

1. *Coverage*
   In attempting to describe a unified vision of speech and language processing, we cover areas that traditionally are taught in different courses in different departments: speech recognition in electrical engineering; parsing, semantic interpretation, and pragmatics in natural language processing courses in computer science departments; and computational morphology and phonology in computational linguistics courses in linguistics departments. The book introduces the fundamental algorithms of each of these fields, whether originally proposed for spoken or written language, whether logical or statistical in origin, and attempts to tie together the descriptions of algorithms from different domains. We have also included coverage of applications like spelling-checking and information retrieval and extraction as well as areas like cognitive modeling. A potential problem with this broad-coverage approach is that it required us to include introductory material for each field; thus linguists may want to skip our description of articulatory phonetics, computer scientists may want to skip such sections as regular expressions, and electrical engineers skip the sections on signal processing. Of course, even in a book this long, we didn't have room for everything. Thus this book should not be considered a substitute for important relevant courses in linguistics, automata and formal language theory, or, especially, statistics and information theory.
2. *Emphasis on practical applications*
   It is important to show how language-related algorithms and techniques (from HMMs to unification, from the lambda calculus to transformation-based learning) can be applied to important real-world problems: spelling checking, text document search, speech recogni-

tion, Web-page processing, part-of-speech tagging, machine translation, and spoken-language dialogue agents. We have attempted to do this by integrating the description of language processing applications into each chapter. The advantage of this approach is that as the relevant linguistic knowledge is introduced, the student has the background to understand and model a particular domain.

3. *Emphasis on scientific evaluation*
   The recent prevalence of statistical algorithms in language processing and the growth of organized evaluations of speech and language processing systems has led to a new emphasis on evaluation. We have, therefore, tried to accompany most of our problem domains with a **Methodology Box** describing how systems are evaluated (e.g., including such concepts as training and test sets, cross-validation, and information-theoretic evaluation metrics like perplexity).
4. *Description of widely available language processing resources*
   Modern speech and language processing is heavily based on common resources: raw speech and text corpora, annotated corpora and treebanks, standard tagsets for labeling pronunciation, part-of-speech, parses, word-sense, and dialogue-level phenomena. We have tried to introduce many of these important resources throughout the book (e.g., the Brown, Switchboard, callhome, ATIS, TREC, MUC, and BNC corpora) and provide complete listings of many useful tagsets and coding schemes (such as the Penn Treebank, CLAWS C5 and C7, and the ARPAbet) but some inevitably got left out. Furthermore, rather than include references to URLs for many resources directly in the textbook, we have placed them on the book's Web site, where they can more readily updated.

The book is primarily intended for use in a graduate or advanced undergraduate course or sequence. Because of its comprehensive coverage and the large number of algorithms, the book is also useful as a reference for students and professionals in any of the areas of speech and language processing.

Overview of the Book

The book is divided into four parts in addition to an introduction and end matter. Part I, "Words", introduces concepts related to the processing of words: phonetics, phonology, morphology, and algorithms used to process them: finite automata, finite transducers, weighted transducers, N-grams,

and Hidden Markov Models. Part II, "Syntax", introduces parts-of-speech and phrase structure grammars for English and gives essential algorithms for processing word classes and structured relationships among words: part-of-speech taggers based on HMMs and transformation-based learning, the CYK and Earley algorithms for parsing, unification and typed feature structures, lexicalized and probabilistic parsing, and analytical tools like the Chomsky hierarchy and the pumping lemma. Part III, "Semantics", introduces first order predicate calculus and other ways of representing meaning, several approaches to compositional semantic analysis, along with applications to information retrieval, information extraction, speech understanding, and machine translation. Part IV, "Pragmatics", covers reference resolution and discourse structure and coherence, spoken dialogue phenomena like dialogue and speech act modeling, dialogue structure and coherence, and dialogue managers, as well as a comprehensive treatment of natural language generation and of machine translation.

Using this Book

The book provides enough material to be used for a full-year sequence in speech and language processing. It is also designed so that it can be used for a number of different useful one-term courses:

NLP 1 quarter	NLP 1 semester	Speech + NLP 1 semester	Comp. Linguistics 1 quarter
1. Intro	1. Intro	1. Intro	1. Intro
2. Regex, FSA	2. Regex, FSA	2. Regex, FSA	2. Regex, FSA
8. POS tagging	3. Morph., FST	3. Morph., FST	3. Morph., FST
9. CFGs	6. *N*-grams	4. Comp. Phonol.	4. Comp. Phonol.
10. Parsing	8. POS tagging	5. Prob. Pronun.	10. Parsing
11. Unification	9. CFGs	6. *N*-grams	11. Unification
14. Semantics	10. Parsing	7. HMMs & ASR	13. Complexity
15. Sem. Analysis	11. Unification	8. POS tagging	16. Lex. Semantics
18. Discourse	12. Prob. Parsing	9. CFGs	18. Discourse
20. Generation	14. Semantics	10. Parsing	19. Dialogue
	15. Sem. Analysis	12. Prob. Parsing	
	16. Lex. Semantics	14. Semantics	
	17. WSD and IR	15. Sem. Analysis	
	18. Discourse	19. Dialogue	
	20. Generation	21. Mach. Transl.	
	21. Mach. Transl.		

Selected chapters from the book could also be used to augment courses in Artificial Intelligence, Cognitive Science, or Information Retrieval.

Acknowledgments

The three contributing writers for the book are Andy Kehler, who wrote Chapter 18 (Discourse), Keith Vander Linden, who wrote Chapter 20 (Generation), and Nigel Ward, who wrote most of Chapter 21 (Machine Translation). Andy Kehler also wrote Section 19.4 of Chapter 19. Paul Taylor wrote most of Section 4.7 and Section 7.8.

Dan would like to thank his parents for encouraging him to do a really good job of everything he does, finish it in a timely fashion, and make time for going to the gym. He would also like to thank Nelson Morgan, for introducing him to speech recognition and teaching him to ask "but does it work?"; Jerry Feldman, for sharing his intense commitment to finding the right answers and teaching him to ask "but is it really important?"; Chuck Fillmore, his first advisor, for sharing his love for language and especially argument structure, and teaching him to always go look at the data, (and all of them for teaching by example that it's only worthwhile if it's fun); and Robert Wilensky, his dissertation advisor, for teaching him the importance of collaboration and group spirit in research. He is also grateful to the CU Lyric Theater program and the casts of *South Pacific*, *Gianni Schicchi*, *Guys and Dolls*, *Gondoliers*, *Iolanthe*, and *Oklahoma*, and to Doris and Cary, Elaine and Eric, Irene and Sam, Susan and Richard, Lisa and Mike, Mike and Fia, Erin and Chris, Eric and Beth, Pearl and Tristan, Bruce and Peggy, Ramon and Rebecca, Adele and Ali, Terry, Kevin, Becky, Temmy, Lil, Lin and Ron and David, Mike, and Jessica and Bill, and all their families for providing lots of emotional support and often a place to stay during the writing.

Jim would like to thank his parents for encouraging him and allowing him to follow what must have seemed like an odd path at the time. He would also like to thank his thesis advisor, Robert Wilensky, for giving him his start in NLP at Berkeley; Peter Norvig, for providing many positive examples along the way; Rick Alterman, for encouragement and inspiration at a critical time; and Chuck Fillmore, George Lakoff, Paul Kay, and Susanna Cumming for teaching him what little he knows about linguistics. He'd also like to thank Michael Main for covering for him while he shirked his departmental duties. Finally, he'd like to thank his wife Linda for all her support and patience through all the years it took to complete this book.

Boulder is a very rewarding place to work on speech and language processing. We'd like to thank our colleagues here for their collaborations, which have greatly influenced our research and teaching: Alan Bell, Barbara Fox, Laura Michaelis and Lise Menn in linguistics; Clayton Lewis, Gerhard

Fischer, Mike Eisenberg, Mike Mozer, Liz Jessup, and Andrzej Ehrenfeucht in computer science; Walter Kintsch, Tom Landauer, and Alice Healy in psychology; Ron Cole, John Hansen, and Wayne Ward in the Center for Spoken Language Understanding, and our current and former students in the computer science and linguistics departments: Marion Bond, Noah Coccaro, Michelle Gregory, Keith Herold, Michael Jones, Patrick Juola, Keith Vander Linden, Laura Mather, Taimi Metzler, Douglas Roland, and Patrick Schone.

This book has benefited from careful reading and enormously helpful comments from a number of readers and from course-testing. We are deeply indebted to colleagues who each took the time to read and give extensive comments and advice, which vastly improved large parts of the book, including Alan Bell, Bob Carpenter, Jan Daciuk, Graeme Hirst, Andy Kehler, Kemal Oflazer, Andreas Stolcke, and Nigel Ward. Our editor Alan Apt, our series editors Peter Norvig and Stuart Russell, and our production editor Scott DiSanno made many helpful suggestions on design and content. We are also indebted to many friends and colleagues who read individual sections of the book or answered our many questions for their comments and advice, including the students in our classes at the University of Colorado, Boulder, and in Dan's classes at the University of California, Berkeley, and the LSA Summer Institute at the University of Illinois at Urbana-Champaign, as well as

Yoshi Asano, Todd M. Bailey, John Bateman, Giulia Bencini, Lois Boggess, Michael Braverman, Nancy Chang, Jennifer Chu-Carroll, Noah Coccaro, Gary Cottrell, Gary Dell, Jeff Elman, Robert Dale, Dan Fass, Bill Fisher, Eric Fosler-Lussier, James Garnett, Susan Garnsey, Dale Gerdemann, Dan Gildea, Michelle Gregory, Nizar Habash, Jeffrey Haemer, Jorge Hankamer, Keith Herold, Beth Heywood, Derrick Higgins, Erhard Hinrichs, Julia Hirschberg, Jerry Hobbs, Fred Jelinek, Liz Jessup, Aravind Joshi, Terry Kleeman, Jean-Pierre Koenig, Kevin Knight, Shalom Lappin, Julie Larson, Stephen Levinson, Jim Magnuson, Jim Mayfield, Lise Menn, Laura Michaelis, Corey Miller, Nelson Morgan, Christine Nakatani, Mike Neufeld, Peter Norvig, Mike O'Connell, Mick O'Donnell, Rob Oberbreckling, Martha Palmer, Dragomir Radev, Terry Regier, Ehud Reiter, Phil Resnik, Klaus Ries, Ellen Riloff, Mike Rosner, Dan Roth, Patrick Schone, Liz Shriberg, Richard Sproat, Subhashini Srinivasin, Paul Taylor, Wayne Ward, Pauline Welby, Dekai Wu, and Victor Zue.

We'd also like to thank the Institute of Cognitive Science and the Departments of Computer Science and Linguistics for their support over the years. We are also very grateful to the National Science Foundation: Dan Jurafsky's time on the book was supported in part by NSF CAREER Award IIS-9733067 and Andy Kehler was supported in part by NSF Award IIS-9619126.

Daniel Jurafsky
James H. Martin
Boulder, Colorado

1 INTRODUCTION

Dave Bowman: Open the pod bay doors, HAL.
HAL: I'm sorry Dave, I'm afraid I can't do that.
Stanley Kubrick and Arthur C. Clarke,
screenplay of *2001: A Space Odyssey*

The HAL 9000 computer in Stanley Kubrick's film *2001: A Space Odyssey* is one of the most recognizable characters in twentieth-century cinema. HAL is an artificial agent capable of such advanced language-processing behavior as speaking and understanding English, and at a crucial moment in the plot, even reading lips. It is now clear that HAL's creator Arthur C. Clarke was a little optimistic in predicting when an artificial agent such as HAL would be available. But just how far off was he? What would it take to create at least the language-related parts of HAL? Minimally, such an agent would have to be capable of interacting with humans via language, which includes understanding humans via **speech recognition** and **natural language understanding** (and, of course, **lip-reading**), and of communicating with humans via **natural language generation** and **speech synthesis**. HAL would also need to be able to do **information retrieval** (finding out where needed textual resources reside), **information extraction** (extracting pertinent facts from those textual resources), and **inference** (drawing conclusions based on known facts).

Although these problems are far from completely solved, much of the language-related technology that HAL needs is currently being developed, with some of it already available commercially. Solving these problems, and others like them, is the main concern of the fields known as Natural Language Processing, Computational Linguistics, and Speech Recognition and Synthesis, which together we call **Speech and Language Processing**. The goal of this book is to describe the state of the art of this technology

at the start of the twenty-first century. The applications we will consider are all of those needed for agents like HAL as well as other valuable areas of language processing such as **spelling correction**, **grammar checking**, **information retrieval**, and **machine translation**.

1.1 KNOWLEDGE IN SPEECH AND LANGUAGE PROCESSING

By speech and language processing, we have in mind those computational techniques that process spoken and written human language, *as language*. As we will see, this is an inclusive definition that encompasses everything from mundane applications such as word counting and automatic hyphenation, to cutting edge applications such as automated question answering on the Web, and real-time spoken language translation.

What distinguishes these language processing applications from other data processing systems is their use of *knowledge of language*. Consider the Unix `wc` program, which is used to count the total number of bytes, words, and lines in a text file. When used to count bytes and lines, `wc` is an ordinary data processing application. However, when it is used to count the words in a file it requires *knowledge about what it means to be a word*, and thus becomes a language processing system.

Of course, `wc` is an extremely simple system with an extremely limited and impoverished knowledge of language. More-sophisticated language agents such as HAL require much broader and deeper knowledge of language. To get a feeling for the scope and kind of knowledge required in more-sophisticated applications, consider some of what HAL would need to know to engage in the dialogue that begins this chapter.

To determine what Dave is saying, HAL must be capable of analyzing an incoming audio signal and recovering the exact sequence of words Dave used to produce that signal. Similarly, in generating its response, HAL must be able to take a sequence of words and generate an audio signal that Dave can recognize. Both of these tasks require knowledge about **phonetics and phonology**, which can help model how words are pronounced in colloquial speech (Chapters 4 and 5).

Note also that unlike Star Trek's Commander Data, HAL is capable of producing contractions like *I'm* and *can't*. Producing and recognizing these and other variations of individual words (e.g., recognizing that *doors* is plural) requires knowledge about **morphology**, which captures information about the shape and behavior of words in context (Chapters 2 and 3).

Moving beyond individual words, HAL must know how to analyze the structure underlying Dave's request. Such an analysis is necessary among other reasons for HAL to determine that Dave's utterance is a request for action, as opposed to a simple statement about the world or a question about the door, as in the following variations of his original statement.

HAL, the pod bay door is open.

HAL, is the pod bay door open?

In addition, HAL must use similar structural knowledge to properly string together the words that constitute its response. For example, HAL must know that the following sequence of words will not make sense to Dave, despite the fact that it contains precisely the same set of words as the original.

I'm I do, sorry that afraid Dave I'm can't.

The knowledge needed to order and group words together comes under the heading of **syntax**.

Of course, simply knowing the words and the syntactic structure of what Dave said does not tell HAL much about the nature of his request. To know that Dave's command is actually about opening the pod bay door, rather than an inquiry about the day's lunch menu, requires knowledge of the meanings of the component words, the domain of **lexical semantics**, and knowledge of how these components combine to form larger meanings, **compositional semantics**.

Next, despite its bad behavior, HAL knows enough to be polite to Dave. It could, for example, have simply replied *No* or *No, I won't open the door*. Instead, it first embellishes its response with the phrases *I'm sorry* and *I'm afraid*, and then only indirectly signals its refusal by saying *I can't*, rather than the more direct (and truthful) *I won't*.[1] The appropriate use of this kind of polite and indirect language comes under the heading of **pragmatics**.

Finally, rather than simply ignoring Dave's command and leaving the door closed, HAL chooses to engage in a structured conversation relevant to Dave's initial request. HAL's correct use of the word *that* in its answer to Dave's request is a simple illustration of the kind of between-utterance device common in such conversations. Correctly structuring these such conversations requires knowledge of **discourse conventions**.

To summarize, the knowledge of language needed to engage in complex language behavior can be separated into six distinct categories.

[1] For those unfamiliar with HAL, it is neither sorry nor afraid, nor is it incapable of opening the door. It has simply decided in a fit of paranoia to kill its crew.

- Phonetics and Phonology — The study of linguistic sounds
- Morphology — The study of the meaningful components of words
- Syntax — The study of the structural relationships between words
- Semantics — The study of meaning
- Pragmatics — The study of how language is used to accomplish goals
- Discourse — The study of linguistic units larger than a single utterance

1.2 AMBIGUITY

A perhaps surprising fact about the six categories of linguistic knowledge is that most or all tasks in speech and language processing can be viewed as resolving **ambiguity** at one of these levels. We say some input is ambiguous if there are multiple alternative linguistic structures than can be built for it. Consider the spoken sentence *I made her duck.* Here's five different meanings this sentence could have (there are more), each of which exemplifies an ambiguity at some level:

AMBIGUITY

(1.1) I cooked waterfowl for her.

(1.2) I cooked waterfowl belonging to her.

(1.3) I created the (plaster?) duck she owns.

(1.4) I caused her to quickly lower her head or body.

(1.5) I waved my magic wand and turned her into undifferentiated waterfowl.

These different meanings are caused by a number of ambiguities. First, the words *duck* and *her* are morphologically or syntactically ambiguous in their part-of-speech. *Duck* can be a verb or a noun, while *her* can be a dative pronoun or a possessive pronoun. Second, the word *make* is semantically ambiguous; it can mean *create* or *cook*. Finally, the verb *make* is syntactically ambiguous in a different way. *Make* can be transitive, that is, taking a single direct object (1.2), or it can be ditransitive, that is, taking two objects (1.5), meaning that the first object (*her*) got made into the second object (*duck*). Finally, *make* can take a direct object and a verb (1.4), meaning that the object (*her*) got caused to perform the verbal action (*duck*). Furthermore, in a spoken sentence, there is an even deeper kind of ambiguity; the first word could have been *eye* or the second word *maid*.

We will often introduce the models and algorithms we present throughout the book as ways to **resolve** or **disambiguate** these ambiguities. For

example deciding whether *duck* is a verb or a noun can be solved by **part-of-speech tagging**. Deciding whether *make* means "create" or "cook" can be solved by **word sense disambiguation**. Resolution of part-of-speech and word sense ambiguities are two important kinds of **lexical disambiguation**. A wide variety of tasks can be framed as lexical disambiguation problems. For example, a text-to-speech synthesis system reading the word *lead* needs to decide whether it should be pronounced as in *lead pipe* or as in *lead me on*. By contrast, deciding whether *her* and *duck* are part of the same entity (as in (1.1) or (1.4)) or are different entity (as in (1.2)) is an example of **syntactic disambiguation** and can be addressed by **probabilistic parsing**. Ambiguities that don't arise in this particular example (like whether a given sentence is a statement or a question) will also be resolved, for example by **speech act interpretation**.

1.3 Models and Algorithms

One of the key insights of the last 50 years of research in language processing is that the various kinds of knowledge described in the last sections can be captured through the use of a small number of formal models, or theories. Fortunately, these models and theories are all drawn from the standard toolkits of Computer Science, Mathematics, and Linguistics and should be generally familiar to those trained in those fields. Among the most important elements in this toolkit are **state machines**, **formal rule systems**, **logic**, as well as **probability theory** and other machine learning tools. These models, in turn, lend themselves to a small number of algorithms from well-known computational paradigms. Among the most important of these are **state space search** algorithms and **dynamic programming** algorithms.

In their simplest formulation, state machines are formal models that consist of states, transitions among states, and an input representation. Some of the variations of this basic model that we will consider are **deterministic** and **non-deterministic finite-state automata**, **finite-state transducers**, which can write to an output device, **weighted automata**, **Markov models**, and **hidden Markov models**, which have a probabilistic component.

Closely related to these somewhat procedural models are their declarative counterparts: formal rule systems. Among the more important ones we will consider are **regular grammars** and **regular relations**, **context-free grammars**, **feature-augmented grammars**, as well as probabilistic variants of them all. State machines and formal rule systems are the main tools

used when dealing with knowledge of phonology, morphology, and syntax.

The algorithms associated with both state-machines and formal rule systems typically involve a search through a space of states representing hypotheses about an input. Representative tasks include searching through a space of phonological sequences for a likely input word in speech recognition, or searching through a space of trees for the correct syntactic parse of an input sentence. Among the algorithms that are often used for these tasks are well-known graph algorithms such as **depth-first search**, as well as heuristic variants such as **best-first**, and **A* search**. The dynamic programming paradigm is critical to the computational tractability of many of these approaches by ensuring that redundant computations are avoided.

The third model that plays a critical role in capturing knowledge of language is logic. We will discuss **first order logic**, also known as the **predicate calculus**, as well as such related formalisms as feature-structures, semantic networks, and conceptual dependency. These logical representations have traditionally been the tool of choice when dealing with knowledge of semantics, pragmatics, and discourse (although, as we will see, applications in these areas are increasingly relying on the simpler mechanisms used in phonology, morphology, and syntax).

Probability theory is the final element in our set of techniques for capturing linguistic knowledge. Each of the other models (state machines, formal rule systems, and logic) can be augmented with probabilities. One major use of probability theory is to solve the many kinds of ambiguity problems that we discussed earlier; almost any speech and language processing problem can be recast as: "given N choices for some ambiguous input, choose the most probable one".

Another major advantage of probabilistic models is that they are one of a class of **machine learning** models. Machine learning research has focused on ways to automatically learn the various representations described above; automata, rule systems, search heuristics, classifiers. These systems can be trained on large corpora and can be used as a powerful modeling technique, especially in places where we don't yet have good causal models. Machine learning algorithms will be described throughout the book.

1.4 Language, Thought, and Understanding

To many, the ability of computers to process language as skillfully as we do will signal the arrival of truly intelligent machines. The basis of this belief is

the fact that the effective use of language is intertwined with our general cognitive abilities. Among the first to consider the computational implications of this intimate connection was Alan Turing (1950). In this famous paper, Turing introduced what has come to be known as the **Turing Test**. Turing began with the thesis that the question of what it would mean for a machine to think was essentially unanswerable due to the inherent imprecision in the terms *machine* and *think*. Instead, he suggested an empirical test, a game, in which a computer's use of language would form the basis for determining if it could think. If the machine could win the game it would be judged intelligent. TURING TEST

In Turing's game, there are three participants: two people and a computer. One of the people is a contestant and plays the role of an interrogator. To win, the interrogator must determine which of the other two participants is the machine by asking a series of questions via a teletype. The task of the machine is to fool the interrogator into believing it is a person by responding as a person would to the interrogator's questions. The task of the second human participant is to convince the interrogator that the other participant is the machine, and that they are human.

The following interaction from Turing's paper illustrates the kind of interactions he had in mind. Clearly a convincing simulation of humanity doesn't require being an expert in all domains:

> Q: Please write me a sonnet on the topic of the Forth Bridge.
> A: Count me out on this one. I never could write poetry.
> Q: Add 34957 to 70764.
> A: (Pause about 30 seconds and then give answer as) 105621.

Turing predicted that roughly by the end of the twentieth century a machine with 10 gigabytes of memory would have around a 30% chance of fooling a human interrogator after 5 minutes of questioning. Given the fact that you can fool some of the people all the time, it is not clear how rigorous this particular standard is. Nevertheless, the critical issue for Turing was that using language as humans do is sufficient, by itself, as an operational test for intelligence.

Surprisingly, many of the issues raised by Turing became relevant as early as 1966 in the context of the ELIZA program (Weizenbaum, 1966). ELIZA was an early natural language processing system capable of carrying on a limited form of conversation with a user. Consider the following session with a version of ELIZA that imitated the responses of a Rogerian psychotherapist.

$User_1$: You are like my father in some ways.
$ELIZA_1$: WHAT RESEMBLANCE DO YOU SEE
$User_2$: You are not very aggressive but I think you don't want me to notice that.
$ELIZA_2$: WHAT MAKES YOU THINK I AM NOT AGGRESSIVE
$User_3$: You don't argue with me.
$ELIZA_3$: WHY DO YOU THINK I DON'T ARGUE WITH YOU
$User_4$: You are afraid of me.
$ELIZA_4$: DOES IT PLEASE YOU TO BELIEVE I AM AFRAID OF YOU

As we will discuss in more detail in Chapter 2, ELIZA is a remarkably simple program that makes use of pattern-matching to process the input and translate it into suitable outputs. The success of this simple technique in this domain is due to the fact that ELIZA doesn't actually need to *know* anything to mimic a Rogerian psychotherapist. As Weizenbaum notes, this is one of the few dialogue genres where the listener can act as if they know nothing of the world.

ELIZA's deep relevance to Turing's ideas is that many people who interacted with ELIZA came to believe that it really *understood* them and their problems. Indeed, Weizenbaum (1976) notes that many of these people continued to believe in ELIZA's abilities even after the program's operation was explained to them. In more recent years, Weizenbaum's informal reports have been repeated in a somewhat more controlled setting. Since 1991, an event known as the Loebner Prize competition has attempted to put various computer programs to the Turing test. Although these contests have proven to have little scientific interest, a consistent result over the years has been that even the crudest programs can fool some of the judges some of the time (Shieber, 1994). Not surprisingly, these results have done nothing to quell the ongoing debate over the suitability of the Turing test as a test for intelligence among philosophers and AI researchers (Searle, 1980).

Fortunately, for the purposes of this book, the relevance of these results does not hinge on whether or not computers will ever be intelligent, or understand natural language. Far more important is recent related research in the social sciences that has confirmed another of Turing's predictions from the same paper.

> Nevertheless I believe that at the end of the century the use of words and educated opinion will have altered so much that we will be able to speak of machines thinking without expecting to be contradicted.

It is now clear that regardless of what people believe or know about the inner workings of computers, they talk about them and interact with them as

social entities. People act toward computers as if they were people; they are polite to them, treat them as team members, and expect among other things that computers should be able to understand their needs, and be capable of interacting with them naturally. For example, Reeves and Nass (1996) found that when a computer asked a human to evaluate how well the computer had been doing, the human gives more positive responses than when a different computer asks the same questions. People seemed to be afraid of being impolite. In a different experiment, Reeves and Nass found that people also give computers higher performance ratings if the computer has recently said something flattering to the human. Given these predispositions, speech and language-based systems may provide many users with the most natural interface for many applications. This fact has led to a long-term focus in the field on the design of **conversational agents**, artificial entities that communicate conversationally.

1.5 The State of the Art and the Near-Term Future

> We can only see a short distance ahead, but we can see plenty there that needs to be done.
>
> Alan Turing.

This is an exciting time for the field of speech and language processing. The recent commercialization of robust speech recognition systems, and the rise of the Web, have placed speech and language processing applications in the spotlight, and have pointed out a plethora of exciting possible applications. The following scenarios serve to illustrate some current applications and near-term possibilities.

A Canadian computer program accepts daily weather data and generates weather reports that are passed along unedited to the public in English and French (Chandioux, 1976).

The *Babel Fish* translation system from Systran handles over 1,000,000 translation requests a day from the AltaVista search engine site.

A visitor to Cambridge, Massachusetts, asks a computer about places to eat using only spoken language. The system returns relevant information from a database of facts about the local restaurant scene (Zue et al., 1991).

These scenarios represent just a few of applications possible given current technology. The following, somewhat more speculative scenarios, give

some feeling for applications currently being explored at research and development labs around the world.

A computer reads hundreds of typed student essays and grades them in a manner that is indistinguishable from human graders (Landauer et al., 1997).

An automated reading tutor helps improve literacy by having children read stories and using a speech recognizer to intervene when the reader asks for reading help or makes mistakes (Mostow and Aist, 1999).

A computer equipped with a vision system watches a short video clip of a soccer match and provides an automated natural language report on the game (Wahlster, 1989).

A computer predicts upcoming words or expands telegraphic speech to assist people with a speech or communication disability (Newell et al., 1998; McCoy et al., 1998).

1.6 Some Brief History

Historically, speech and language processing has been treated very differently in computer science, electrical engineering, linguistics, and psychology/cognitive science. Because of this diversity, speech and language processing encompasses a number of different but overlapping fields in these different departments: **computational linguistics** in linguistics, **natural language processing** in computer science, **speech recognition** in electrical engineering, **computational psycholinguistics** in psychology. This section summarizes the different historical threads which have given rise to the field of speech and language processing. This section will provide only a sketch; see the individual chapters for more detail on each area and its terminology.

Foundational Insights: 1940s and 1950s

The earliest roots of the field date to the intellectually fertile period just after World War II that gave rise to the computer itself. This period from the 1940s through the end of the 1950s saw intense work on two foundational paradigms: the **automaton** and **probabilistic** or **information-theoretic models**.

The automaton arose in the 1950s out of Turing's (1936) model of algorithmic computation, considered by many to be the foundation of modern computer science. Turing's work led first to the **McCulloch-Pitts neuron** (McCulloch and Pitts, 1943), a simplified model of the neuron as a kind of

computing element that could be described in terms of propositional logic, and then to the work of Kleene (1951) and (1956) on finite automata and regular expressions. Shannon (1948) applied probabilistic models of discrete Markov processes to automata for language. Drawing the idea of a finite-state Markov process from Shannon's work, Chomsky (1956) first considered finite-state machines as a way to characterize a grammar, and defined a finite-state language as a language generated by a finite-state grammar. These early models led to the field of **formal language theory**, which used algebra and set theory to define formal languages as sequences of symbols. This includes the context-free grammar, first defined by Chomsky (1956) for natural languages but independently discovered by Backus (1959) and Naur et al. (1960) in their descriptions of the ALGOL programming language.

The second foundational insight of this period was the development of probabilistic algorithms for speech and language processing, which dates to Shannon's other contribution: the metaphor of the **noisy channel** and **decoding** for the transmission of language through media like communication channels and speech acoustics. Shannon also borrowed the concept of **entropy** from thermodynamics as a way of measuring the information capacity of a channel, or the information content of a language, and performed the first measure of the entropy of English using probabilistic techniques.

It was also during this early period that the sound spectrograph was developed (Koenig et al., 1946), and foundational research was done in instrumental phonetics that laid the groundwork for later work in speech recognition. This led to the first machine speech recognizers in the early 1950s. In 1952, researchers at Bell Labs built a statistical system that could recognize any of the 10 digits from a single speaker (Davis et al., 1952). The system had 10 speaker-dependent stored patterns roughly representing the first two vowel formants in the digits. They achieved 97–99% accuracy by choosing the pattern which had the highest relative correlation coefficient with the input.

The Two Camps: 1957–1970

By the end of the 1950s and the early 1960s, speech and language processing had split very cleanly into two paradigms: symbolic and stochastic.

The symbolic paradigm took off from two lines of research. The first was the work of Chomsky and others on formal language theory and generative syntax throughout the late 1950s and early to mid 1960s, and the work of many linguistics and computer scientists on parsing algorithms, initially top-down and bottom-up and then via dynamic programming. One of the earliest

complete parsing systems was Zelig Harris's Transformations and Discourse Analysis Project (TDAP), which was implemented between June 1958 and July 1959 at the University of Pennsylvania (Harris, 1962).[2] The second line of research was the new field of artificial intelligence. In the summer of 1956 John McCarthy, Marvin Minsky, Claude Shannon, and Nathaniel Rochester brought together a group of researchers for a two-month workshop on what they decided to call artificial intelligence (AI). Although AI always included a minority of researchers focusing on stochastic and statistical algorithms (include probabilistic models and neural nets), the major focus of the new field was the work on reasoning and logic typified by Newell and Simon's work on the Logic Theorist and the General Problem Solver. At this point early natural language understanding systems were built, These were simple systems that worked in single domains mainly by a combination of pattern matching and keyword search with simple heuristics for reasoning and question-answering. By the late 1960s more formal logical systems were developed.

The stochastic paradigm took hold mainly in departments of statistics and of electrical engineering. By the late 1950s the Bayesian method was beginning to be applied to the problem of optical character recognition. Bledsoe and Browning (1959) built a Bayesian system for text-recognition that used a large dictionary and computed the likelihood of each observed letter sequence given each word in the dictionary by multiplying the likelihoods for each letter. Mosteller and Wallace (1964) applied Bayesian methods to the problem of authorship attribution on *The Federalist* papers.

The 1960s also saw the rise of the first serious testable psychological models of human language processing based on transformational grammar, as well as the first on-line corpora: the Brown corpus of American English, a 1 million word collection of samples from 500 written texts from different genres (newspaper, novels, non-fiction, academic, etc.), which was assembled at Brown University in 1963–64 (Kučera and Francis, 1967; Francis, 1979; Francis and Kučera, 1982), and William S. Y. Wang's 1967 DOC (Dictionary on Computer), an on-line Chinese dialect dictionary.

Four Paradigms: 1970–1983

The next period saw an explosion in research in speech and language processing and the development of a number of research paradigms that still dominate the field.

[2] This system was reimplemented recently and is described by Joshi and Hopely (1999) and Karttunen (1999), who note that the parser was essentially implemented as a cascade of finite-state transducers.

The **stochastic** paradigm played a huge role in the development of speech recognition algorithms in this period, particularly the use of the Hidden Markov Model and the metaphors of the noisy channel and decoding, developed independently by Jelinek, Bahl, Mercer, and colleagues at IBM's Thomas J. Watson Research Center, and by Baker at Carnegie Mellon University, who was influenced by the work of Baum and colleagues at the Institute for Defense Analyses in Princeton. AT&T's Bell Laboratories was also a center for work on speech recognition and synthesis; see Rabiner and Juang (1993) for descriptions of the wide range of this work.

The **logic-based** paradigm was begun by the work of Colmerauer and his colleagues on Q-systems and metamorphosis grammars (Colmerauer, 1970, 1975), the forerunners of Prolog, and Definite Clause Grammars (Pereira and Warren, 1980). Independently, Kay's (1979) work on functional grammar, and shortly later, Bresnan and Kaplan's (1982) work on LFG, established the importance of feature structure unification.

The **natural language understanding** field took off during this period, beginning with Terry Winograd's SHRDLU system, which simulated a robot embedded in a world of toy blocks (Winograd, 1972a). The program was able to accept natural language text commands *(Move the red block on top of the smaller green one)* of a hitherto unseen complexity and sophistication. His system was also the first to attempt to build an extensive (for the time) grammar of English, based on Halliday's systemic grammar. Winograd's model made it clear that the problem of parsing was well-enough understood to begin to focus on semantics and discourse models. Roger Schank and his colleagues and students (in what was often referred to as the *Yale School*) built a series of language understanding programs that focused on human conceptual knowledge such as scripts, plans and goals, and human memory organization (Schank and Albelson, 1977; Schank and Riesbeck, 1981; Cullingford, 1981; Wilensky, 1983; Lehnert, 1977). This work often used network-based semantics (Quillian, 1968; Norman and Rumelhart, 1975; Schank, 1972; Wilks, 1975c, 1975b; Kintsch, 1974) and began to incorporate Fillmore's notion of case roles (Fillmore, 1968) into their representations (Simmons, 1973).

The logic-based and natural-language understanding paradigms were unified on systems that used predicate logic as a semantic representation, such as the LUNAR question-answering system (Woods, 1967, 1973).

The **discourse modeling** paradigm focused on four key areas in discourse. Grosz and her colleagues introduced the study of substructure in discourse, and of discourse focus (Grosz, 1977a; Sidner, 1983), a number of

researchers began to work on automatic reference resolution (Hobbs, 1978), and the **BDI** (Belief-Desire-Intention) framework for logic-based work on speech acts was developed (Perrault and Allen, 1980; Cohen and Perrault, 1979).

Empiricism and Finite State Models Redux: 1983–1993

This next decade saw the return of two classes of models which had lost popularity in the late 1950s and early 1960s, partially due to theoretical arguments against them such as Chomsky's influential review of Skinner's *Verbal Behavior* (Chomsky, 1959b). The first class was finite-state models, which began to receive attention again after work on finite-state phonology and morphology by Kaplan and Kay (1981) and finite-state models of syntax by Church (1980). A large body of work on finite-state models will be described throughout the book.

The second trend in this period was what has been called the "return of empiricism"; most notably here was the rise of probabilistic models throughout speech and language processing, influenced strongly by the work at the IBM Thomas J. Watson Research Center on probabilistic models of speech recognition. These probabilistic methods and other such data-driven approaches spread into part-of-speech tagging, parsing and attachment ambiguities, and connectionist approaches from speech recognition to semantics.

This period also saw considerable work on natural language generation.

The Field Comes Together: 1994–1999

By the last five years of the millennium it was clear that the field was vastly changing. First, probabilistic and data-driven models had become quite standard throughout natural language processing. Algorithms for parsing, part-of-speech tagging, reference resolution, and discourse processing all began to incorporate probabilities, and employ evaluation methodologies borrowed from speech recognition and information retrieval. Second, the increases in the speed and memory of computers had allowed commercial exploitation of a number of subareas of speech and language processing, in particular speech recognition and spelling and grammar checking. Speech and language processing algorithms began to be applied to Augmentative and Alternative Communication (AAC). Finally, the rise of the Web emphasized the need for language-based information retrieval and information extraction.

On Multiple Discoveries

Even in this brief historical overview, we have mentioned a number of cases of multiple independent discoveries of the same idea. Just a few of the "multiples" to be discussed in this book include the application of dynamic programming to sequence comparison by Viterbi, Vintsyuk, Needleman and Wunsch, Sakoe and Chiba, Sankoff, Reichert *et al.*, and Wagner and Fischer (Chapters 5 and 7); the HMM/noisy channel model of speech recognition by Baker and by Jelinek, Bahl, and Mercer (Chapter 7); the development of context-free grammars by Chomsky and by Backus and Naur (Chapter 9); the proof that Swiss-German has a non-context-free syntax by Huybregts and by Shieber (Chapter 13); the application of unification to language processing by Colmerauer *et al.* and by Kay in (Chapter 11).

Are these multiples to be considered astonishing coincidences? A well-known hypothesis by sociologist of science Robert K. Merton (1961) argues, quite the contrary, that

> all scientific discoveries are in principle multiples, including those that on the surface appear to be singletons.

Of course there are many well-known cases of multiple discovery or invention; just a few examples from an extensive list in Ogburn and Thomas (1922) include the multiple invention of the calculus by Leibnitz and by Newton, the multiple development of the theory of natural selection by Wallace and by Darwin, and the multiple invention of the telephone by Gray and Bell.[3] But Merton gives an further array of evidence for the hypothesis that multiple discovery is the rule rather than the exception, including many cases of putative singletons that turn out be a rediscovery of previously unpublished or perhaps inaccessible work. An even stronger piece of evidence is his ethnomethodological point that scientists themselves act under the assumption that multiple invention is the norm. Thus many aspects of scientific life are designed to help scientists avoid being "scooped"; submission dates on journal articles; careful dates in research records; circulation of preliminary or technical reports.

[3] Ogburn and Thomas are generally credited with noticing that the prevalence of multiple inventions suggests that the cultural milieu and not individual genius is the deciding causal factor in scientific discovery. In an amusing bit of recursion, however, Merton notes that even this idea has been multiply discovered, citing sources from the 19th century and earlier!

A Final Brief Note on Psychology

Many of the chapters in this book include short summaries of psychological research on human processing. Of course, understanding human language processing is an important scientific goal in its own right and is part of the general field of cognitive science. However, an understanding of human language processing can often be helpful in building better machine models of language. This seems contrary to the popular wisdom, which holds that direct mimicry of nature's algorithms is rarely useful in engineering applications. For example, the argument is often made that if we copied nature exactly, airplanes would flap their wings; yet airplanes with fixed wings are a more successful engineering solution. But language is not aeronautics. Cribbing from nature is sometimes useful for aeronautics (after all, airplanes do have wings), but it is particularly useful when we are trying to solve human-centered tasks. Airplane flight has different goals than bird flight; but the goal of speech recognition systems, for example, is to perform exactly the task that human court reporters perform every day: transcribe spoken dialog. Since people already do this well, we can learn from nature's previous solution. Since an important application of speech and language processing systems is for human-computer interaction, it makes sense to copy a solution that behaves the way people are accustomed to.

1.7 SUMMARY

This chapter introduces the field of speech and language processing. The following are some of the highlights of this chapter.

- A good way to understand the concerns of speech and language processing research is to consider what it would take to create an intelligent agent like HAL from 2001: A Space Odyssey.
- Speech and language technology relies on formal models, or representations, of knowledge of language at the levels of phonology and phonetics, morphology, syntax, semantics, pragmatics and discourse. A small number of formal models including state machines, formal rule systems, logic, and probability theory are used to capture this knowledge.
- The foundations of speech and language technology lie in computer science, linguistics, mathematics, electrical engineering and psychology. A small number of algorithms from standard frameworks are used

throughout speech and language processing,

- The critical connection between language and thought has placed speech and language processing technology at the center of debate over intelligent machines. Furthermore, research on how people interact with complex media indicates that speech and language processing technology will be critical in the development of future technologies.
- Revolutionary applications of speech and language processing are currently in use around the world. Recent advances in speech recognition and the creation of the World-Wide Web will lead to many more applications.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

Research in the various subareas of speech and language processing is spread across a wide number of conference proceedings and journals. The conferences and journals most centrally concerned with computational linguistics and natural language processing are associated with the Association for Computational Linguistics (ACL), its European counterpart (EACL), and the International Conference on Computational Linguistics (COLING). The annual proceedings of ACL and EACL, and the biennial COLING conference are the primary forums for work in this area. Related conferences include the biennial conference on Applied Natural Language Processing (ANLP) and the conference on Empirical Methods in Natural Language Processing (EMNLaP). The journal *Computational Linguistics* is the premier publication in the field, although it has a decidedly theoretical and linguistic orientation. The journal *Natural Language Engineering* covers more practical applications of speech and language research.

Research on speech recognition, understanding, and synthesis is presented at the biennial International Conference on Spoken Language Processing (ICSLP) which alternates with the European Conference on Speech Communication and Technology (EUROSPEECH). The IEEE International Conference on Acoustics, Speech, and Signal Processing (IEEE ICASSP) is held annually, as is the meeting of the Acoustical Society of America. Speech journals include *Speech Communication, Computer Speech and Language*, and the *IEEE Transactions on Pattern Analysis and Machine Intelligence*.

Work on language processing from an Artificial Intelligence perspective can be found in the annual meetings of the American Association for Artificial Intelligence (AAAI), as well as the biennial International Joint Conference on Artificial Intelligence (IJCAI) meetings. The following artificial intelligence publications periodically feature work on speech and language processing: *Artificial Intelligence*, *Computational Intelligence*, *IEEE Transactions on Intelligent Systems*, and the *Journal of Artificial Intelligence Research*. Work on cognitive modeling of language can be found at the annual meeting of the Cognitive Science Society, as well as its journal *Cognitive Science*. An influential series of invitation-only workshops was held by ARPA, called variously the *DARPA Speech and Natural Language Processing Workshop* or the *ARPA Workshop on Human Language Technology*.

There are a fair number of textbooks available covering various aspects of speech and language processing. Manning and Schütze (1999) (*Foundations of Statistical Language Processing*) focuses on statistical models of tagging, parsing, disambiguation, collocations, and other areas. Charniak (1993) (*Statistical Language Learning*) is an accessible, though older and less-extensive, introduction to similar material. Allen (1995) (*Natural Language Understanding*) provides extensive coverage of language processing from the AI perspective. Gazdar and Mellish (1989) (*Natural Language Processing in Lisp/Prolog*) covers especially automata, parsing, features, and unification. Pereira and Shieber (1987) gives a Prolog-based introduction to parsing and interpretation. Russell and Norvig (1995) is an introduction to artificial intelligence that includes chapters on natural language processing. Partee et al. (1990) has a very broad coverage of mathematical linguistics. Cole (1997) is a volume of survey papers covering the entire field of speech and language processing. A somewhat dated but still tremendously useful collection of foundational papers can be found in Grosz et al. (1986) (*Readings in Natural Language Processing*).

Of course, a wide-variety of speech and language processing resources are now available on the World-Wide Web. Pointers to these resources are maintained on the home-page for this book at:

`http://www.cs.colorado.edu/~martin/slp.html`.

Part I

WORDS

Words are the fundamental building block of language. Every human language, spoken, signed, or written, is composed of words. Every area of speech and language processing, from speech recognition to machine translation to information retrieval on the Web, requires extensive knowledge about words. Psycholinguistic models of human language processing and models from generative linguistics are also heavily based on lexical knowledge.

The six chapters in this part introduce computational models of the spelling, pronunciation, and morphology of words and cover three important real-world tasks that rely on lexical knowledge: automatic speech recognition (ASR), text-to-speech synthesis (TTS), and the correction of spelling errors. Finally, these chapters define perhaps the most important computational model for speech and language processing: the automaton. Four kinds of automata are covered: finite-state automata (FSAs) and regular expressions, finite-state transducers (FSTs), weighted transducers, and the Hidden Markov Model (HMM), as well as the N-gram model of word sequences.

2 REGULAR EXPRESSIONS AND AUTOMATA

In the old days, if you wanted to impeach a witness you had to go back and fumble through endless transcripts. Now it's on a screen somewhere or on a disk and I can search for a particular word — say every time the witness used the word glove *— and then quickly ask a question about what he said years ago. Right away you see the witness get flustered.*

Johnnie L. Cochran Jr., attorney, *New York Times*, 9/28/97

Imagine that you have become a passionate fan of woodchucks. Desiring more information on this celebrated woodland creature, you turn to your favorite Web browser and type in *woodchuck*. Your browser returns a few sites. You have a flash of inspiration and type in *woodchucks*. This time you discover "interesting links to woodchucks and lemurs" and "all about Vermont's unique, endangered species". Instead of having to do this search twice, you would have rather typed one search command specifying something like *woodchuck with an optional final s*. Furthermore, you might want to find a site whether or not it spelled *woodchucks* with a capital *W* (*Woodchuck*). Or perhaps you might want to search for all the prices in some document; you might want to see all strings that look like *$199* or *$25* or *$24.99*. In this chapter we introduce the **regular expression**, the standard notation for characterizing text sequences. The regular expression is used for specifying text strings in situations like this Web-search example, and in other information retrieval applications, but also plays an important role in word-processing (in PC, Mac, or UNIX applications), computation of frequencies from corpora, and other such tasks.

After we have defined regular expressions, we show how they can be implemented via the **finite-state automaton**. The finite-state automaton is not only the mathematical device used to implement regular expressions, but

also one of the most significant tools of computational linguistics. Variations of automata such as finite-state transducers, Hidden Markov Models, and N-gram grammars are important components of the speech recognition and synthesis, spell-checking, and information-extraction applications that we will introduce in later chapters.

2.1 Regular Expressions

SIR ANDREW: Her C's, her U's and her T's: why that?
Shakespeare, *Twelfth Night*

REGULAR EXPRESSION

One of the unsung successes in standardization in computer science has been the **regular expression** (**RE**), a language for specifying text search strings. The regular expression languages used for searching texts in UNIX (vi, Perl, Emacs, grep), Microsoft Word (version 6 and beyond), and WordPerfect are almost identical, and many RE features exist in the various Web search engines. Besides this practical use, the regular expression is an important theoretical tool throughout computer science and linguistics.

STRINGS

A regular expression (first developed by Kleene (1956) but see the History section for more details) is a formula in a special language that is used for specifying simple classes of **strings**. A string is a sequence of symbols; for the purpose of most text-based search techniques, a string is any sequence of alphanumeric characters (letters, numbers, spaces, tabs, and punctuation). For these purposes a space is just a character like any other, and we represent it with the symbol ␣.

Formally, a regular expression is an algebraic notation for characterizing a set of strings. Thus they can be used to specify search strings as well as to define a language in a formal way. We will begin by talking about regular expressions as a way of specifying searches in texts, and proceed to other uses. Section 2.3 shows that the use of just three regular expression operators is sufficient to characterize strings, but we use the more convenient and commonly-used regular expression syntax of the Perl language throughout this section. Since common text-processing programs agree on most of the syntax of regular expressions, most of what we say extends to all UNIX, Microsoft Word, and WordPerfect regular expressions. Appendix A shows the few areas where these programs differ from the Perl syntax.

CORPUS

Regular expression search requires a **pattern** that we want to search for, and a **corpus** of texts to search through. A regular expression search

function will search through the corpus returning all texts that contain the pattern. In an information retrieval (IR) system such as a Web search engine, the texts might be entire documents or Web pages. In a word-processor, the texts might be individual words, or lines of a document. In the rest of this chapter, we will use this last paradigm. Thus when we give a search pattern, we will assume that the search engine returns the *line of the document* returned. This is what the UNIX `grep` command does. We will underline the exact part of the pattern that matches the regular expression. A search can be designed to return all matches to a regular expression or only the first match. We will show only the first match.

Basic Regular Expression Patterns

The simplest kind of regular expression is a sequence of simple characters. For example, to search for *woodchuck*, we type `/woodchuck/`. So the regular expression `/Buttercup/` matches any string containing the substring *Buttercup*, for example the line *I'm called little Buttercup*) (recall that we are assuming a search application that returns entire lines). From here on we will put slashes around each regular expression to make it clear what is a regular expression and what is a pattern. We use the slash since this is the notation used by Perl, but the slashes are *not* part of the regular expressions.

The search string can consist of a single letter (like `/!/`) or a sequence of letters (like `/urgl/`); The *first* instance of each match to the regular expression is underlined below (although a given application might choose to return more than just the first instance):

RE	Example Patterns Matched
`/woodchucks/`	"interesting links to woodchucks and lemurs"
`/a/`	"Mary Ann stopped by Mona's"
`/Claire␣says,/`	"Dagmar, my gift please," Claire says,"
`/song/`	"all our pretty songs"
`/!/`	"You've left the burglar behind again!" said Nori

Regular expressions are **case sensitive**; lowercase `/s/` is distinct from uppercase `/S/`; (`/s/` matches a lower case *s* but not an uppercase *S*). This means that the pattern `/woodchucks/` will not match the string *Woodchucks*. We can solve this problem with the use of the square braces `[` and `]`. The string of characters inside the braces specify a **disjunction** of characters to match. For example Figure 2.1 shows that the pattern `/[wW]/` matches patterns containing either *w* or *W*.

RE	Match	Example Patterns
/[wW]oodchuck/	Woodchuck or woodchuck	"Woodchuck"
/[abc]/	'a', 'b', *or* 'c'	"In uomini, in soldati"
/[1234567890]/	any digit	"plenty of 7 to 5"

Figure 2.1 The use of the brackets [] to specify a disjunction of characters.

The regular expression /[1234567890]/ specified any single digit. While classes of characters like digits or letters are important building blocks in expressions, they can get awkward (e.g., it's inconvenient to specify

```
/[ABCDEFGHIJKLMNOPQRSTUVWXYZ]/
```

to mean "any capital letter"). In these cases the brackets can be used with the dash (-) to specify any one character in a **range**. The pattern /[2-5]/ specifies any one of the characters *2*, *3*, *4*, or *5*. The pattern /[b-g]/ specifies one of the characters *b*, *c*, *d*, *e*, *f*, or *g*. Some other examples:

RANGE

RE	Match	Example Patterns Matched
/[A-Z]/	an uppercase letter	"we should call it 'Drenched Blossoms'"
/[a-z]/	a lowercase letter	"my beans were impatient to be hoed!"
/[0-9]/	a single digit	"Chapter 1: Down the Rabbit Hole"

Figure 2.2 The use of the brackets [] plus the dash - to specify a range.

The square braces can also be used to specify what a single character *cannot* be, by use of the caret ^. If the caret ^ is the first symbol after the open square brace [, the resulting pattern is negated. For example, the pattern /[^a]/ matches any single character (including special characters) except *a*. This is only true when the caret is the first symbol after the open square brace. If it occurs anywhere else, it usually stands for a caret; Figure 2.3 shows some examples.

RE	Match (single characters)	Example Patterns Matched
[^A-Z]	not an uppercase letter	"Oyfn pripetchik"
[^Ss]	neither 'S' nor 's'	"I have no exquisite reason for't"
[^\.]	not a period	"our resident Djinn"
[e^]	either 'e' or '^'	"look up ^ now"
a^b	the pattern 'a^b'	"look up a^ b now"

Figure 2.3 Uses of the caret ^ for negation or just to mean ^ .

The use of square braces solves our capitalization problem for *woodchucks*. But we still haven't answered our original question; how do we specify both *woodchuck* and *woodchucks*? We can't use the square brackets, because while they allow us to say "s or S", they don't allow us to say "s or nothing". For this we use the question-mark `/?/`, which means "the preceding character or nothing", as shown in Figure 2.4.

RE	Match	Example Patterns Matched
`woodchucks?`	woodchuck or woodchucks	"woodchuck"
`colou?r`	color or colour	"colour"

Figure 2.4 The question-mark ? marks optionality of the previous expression.

We can think of the question-mark as meaning "zero or one instances of the previous character". That is, it's a way of specifying how many of something that we want. So far we haven't needn't to specify that we want more than one of something. But sometimes we need regular expressions that allow repetitions of things. For example, consider the language of (certain) sheep, which consists of strings that look like the following:

baa!
baaa!
baaaa!
baaaaa!
baaaaaa!
...

This language consists of strings with a *b*, followed by at least two *a*s, followed by an exclamation point. The set of operators that allow us to say things like "some number of *a*s" are based on the asterisk or *, commonly called the **Kleene** * (pronounced "cleany star"). The Kleene star means "zero or more occurrences of the immediately previous character or regular expression". So `/a*/` means "any string of zero or more *a*s". This will match *a* or *aaaaaa* but it will also match *Off Minor*, since the string *Off Minor* has zero *a*s. So the regular expression for matching one or more *a* is `/aa*/`, meaning one *a* followed by zero or more *a*s. More complex patterns can also be repeated. So `/[ab]*/` means "zero or more *a*s or *b*s" (not "zero or more right square braces"). This will match strings like *aaaa* or *ababab* or *bbbb*.

KLEENE *

We now know enough to specify part of our regular expression for prices: multiple digits. Recall that the regular expression for an individual digit was `/[0-9]/`. So the regular expression for an integer (a string of digits) is `/[0-9][0-9]*/`. (Why isn't it just `/[0-9]*/`)?

Sometimes it's annoying to have to write the regular expression for digits twice, so there is a shorter way to specify "at least one" of some character.

KLEENE +

This is the **Kleene +**, which means "one or more of the previous character". Thus the expression `/[0-9]+/` is the normal way to specify "a sequence of digits". There are thus two ways to specify the sheep language: `/baaa*!/` or `/baa+!/`.

One very important special character is the period (`/./`, a **wildcard** expression that matches any single character (*except* a carriage return):

RE	Match	Example Patterns
`/beg.n/`	any character between *beg* and *n*	begin, beg'n, begun

Figure 2.5 The use of the period . to specify any character.

The wildcard is often used together with the Kleene star to mean "any string of characters". For example suppose we want to find any line in which a particular word, for example *aardvark*, appears twice. We can specify this with the regular expression `/aardvark.*aardvark/`.

ANCHORS

Anchors are special characters that anchor regular expressions to particular places in a string. The most common anchors are the caret ^ and the dollar-sign $. The caret ^ matches the start of a line. The pattern `/^The/` matches the word *The* only at the start of a line. Thus there are three uses of the caret ^: to match the start of a line, as a negation inside of square brackets, and just to mean a caret. (What are the contexts that allow Perl to know which function a given caret is supposed to have?). The dollar sign $ matches the end of a line. So the pattern `␣$` is a useful pattern for matching a space at the end of a line, and `/^The dog\.$/` matches a line that contains only the phrase *The dog*. (We have to use the backslash here since we want the . to mean "period" and not the wildcard).

There are also two other anchors: `\b` matches a word boundary, while `\B` matches a non-boundary. Thus `/\bthe\b/` matches the word *the* but not the word *other*. More technically, Perl defines a word as any sequence of digits, underscores or letters; this is based on the definition of "words" in programming languages like Perl or C. For example, `/\b99/` will match the string *99* in *There are 99 bottles of beer on the wall* (because 99 follows

a space) but not *99* in *There are 299 bottles of beer on the wall* (since 99 follows a number). But it will match *99* in *$99* (since *99* follows a dollar sign ($), which is not a digit, underscore, or letter).

Disjunction, Grouping, and Precedence

Suppose we need to search for texts about pets; perhaps we are particularly interested in cats and dogs. In such a case we might want to search for either the string *cat* or the string *dog*. Since we can't use the square-brackets to search for "cat or dog" (why not?) we need a new operator, the **disjunction** operator, also called the **pipe** symbol |. The pattern `/cat|dog/` matches either the string `cat` or the string `dog`.

DISJUNCTION

Sometimes we need to use this disjunction operator in the midst of a larger sequence. For example, suppose I want to search for information about pet fish for my cousin David. How can I specify both *guppy* and *guppies*? We cannot simply say `/guppy|ies/`, because that would match only the strings *guppy* and *ies*. This is because sequences like `guppy` take **precedence** over the disjunction operator |. In order to make the disjunction operator apply only to a specific pattern, we need to use the parenthesis operators (and). Enclosing a pattern in parentheses makes it act like a single character for the purposes of neighboring operators like the pipe | and the Kleene*. So the pattern `/gupp(y|ies)/` would specify that we meant the disjunction only to apply to the suffixes `y` and `ies`.

PRECEDENCE

The parenthesis operator (is also useful when we are using counters like the Kleene*. Unlike the | operator, the Kleene* operator applies by default only to a single character, not a whole sequence. Suppose we want to match repeated instances of a string. Perhaps we have a line that has column labels of the form *Column 1 Column 2 Column 3*. The expression `/Column␣[0-9]+␣*/` will not match any column; instead, it will match a column followed by any number of spaces! The star here applies only to the space ␣ that precedes it, not the whole sequence. With the parentheses, we could write the expression `/(Column␣[0-9]+␣*)*/` to match the word *Column*, followed by a number and optional spaces, the whole pattern repeated any number of times.

This idea that one operator may take precedence over another, requiring us to sometimes use parentheses to specify what we mean, is formalized by the **operator precedence hierarchy** for regular expressions. The following table gives the order of RE operator precedence, from highest precedence to lowest precedence:

OPERATOR PRECEDENCE

Parenthesis	`()`
Counters	`* + ? {}`
Sequences and anchors	`the ^my end$`
Disjunction	`|`

Thus, because counters have a higher precedence than sequences, `/the*/` matches *theeeee* but not *thethe*. Because sequences have a higher precedence than disjunction, `/the|any/` matches *the* or *any* but not *theny*.

Patterns can be ambiguous in another way. Consider the expression `/[a-z]*/` when matching against the text *once upon a time*. Since `/[a-z]*/` matches zero or more letters, this expression could match nothing, or just the first letter *o*, or *on*, or *onc*, or *once*. In these cases regular expressions always match the *largest* string they can; we say that patterns are **greedy**, expanding to cover as much of a string as they can.

GREEDY

A Simple Example

Suppose we wanted to write a RE to find cases of the English article *the*. A simple (but incorrect) pattern might be:

```
/the/
```

One problem is that this pattern will miss the word when it begins a sentence and hence is capitalized (i.e., *The*). This might lead us to the following pattern:

```
/[tT]he/
```

But we will still incorrectly return texts with `the` embedded in other words (e.g., *other* or *theology*). So we need to specify that we want instances with a word boundary on both sides:

```
/\b[tT]he\b/
```

Suppose we wanted to do this without the use of `/\b/`? We might want this since `/\b/` won't treat underscores and numbers as word boundaries; but we might want to find *the* in some context where it might also have underlines or numbers nearby (*the_* or *the25*). We need to specify that we want instances in which there are no alphabetic letters on either side of the *the*:

```
/[^a-zA-Z][tT]he[^a-zA-Z]/
```

But there is still one more problem with this pattern: it won't find the word *the* when it begins a line. This is because the regular expression `[^a-zA-Z]`, which we used to avoid embedded *the*s, implies that there must be some single (although non-alphabetic) character before the `the`. We can avoid this by specifying that before the *the* we require *either* the beginning-of-line or a non-alphabetic character:

```
/(^|[^a-zA-Z])[tT]he[^a-zA-Z]/
```

A More Complex Example

Let's try out a more significant example of the power of REs. Suppose we want to build an application to help a user buy a computer on the Web. The user might want "any PC with more than 500 MHz and 32 Gb of disk space for less than $1000". In order to do this kind of retrieval we will first need to be able to look for expressions like *500 MHz* or *32 Gb* or *Compaq* or *Mac* or *$999.99*. In the rest of this section we'll work out some simple regular expressions for this task.

First, let's complete our regular expression for prices. Here's a regular expression for a dollar sign followed by a string of digits. Note that Perl is smart enough to realize that `$` here doesn't mean end-of-line; how might it know that?

```
/$[0-9]+/
```

Now we just need to deal with fractions of dollars. We'll add a decimal point and two digits afterwards:

```
/$[0-9]+\.[0-9][0-9]/
```

This pattern only allows *$199.99* but not *$199*. We need to make the cents optional, and make sure we're at a word boundary:

```
/\b$[0-9]+(\.[0-9][0-9])?\b/
```

How about specifications for processor speed (in megahertz = MHz or gigahertz = GHz)? Here's a pattern for that:

```
/\b[0-9]+␣*(MHz|[Mm]egahertz|GHz|[Gg]igahertz)\b/
```

Note that we use `/␣*/` to mean "zero or more spaces", since there might always be extra spaces lying around. Dealing with disk space (in Gb = gigabytes), or memory size (in Mb = megabytes or Gb = gigabytes), we

need to allow for optional gigabyte fractions again (*5.5 Gb*). Note the use of `?` for making the final `s` optional:

```
/\b[0-9]+␣*(Mb|[Mm]egabytes?)\b/
/\b[0-9](\.[0-9]+)?␣*(Gb|[Gg]igabytes?)\b/
```

Finally, we might want some simple patterns to specify operating systems and vendors:

```
/\b(Win95|Win98|WinNT|Windows␣*(NT|95|98|2000)?)\b/
/\b(Mac|Macintosh|Apple)\b/
```

Advanced Operators

RE	Expansion	Match	Example Patterns
`\d`	`[0-9]`	any digit	Party␣of␣5
`\D`	`[^0-9]`	any non-digit	Blue␣moon
`\w`	`[a-zA-Z0-9␣]`	any alphanumeric or space	Daiyu
`\W`	`[^\w]`	a non-alphanumeric	!!!!
`\s`	`[␣\r\t\n\f]`	whitespace (space, tab)	
`\S`	`[^\s]`	Non-whitespace	in␣Concord

Figure 2.6 Aliases for common sets of characters.

There are also some useful advanced regular expression operators. Figure 2.6 shows some useful aliases for common ranges, which can be used mainly to save typing. Besides the Kleene * and Kleene +, we can also use explicit numbers as counters, by enclosing them in curly brackets. The regular expression `/{3}/` means "exactly 3 occurrences of the previous character or expression". So `/a\.{24}z/` will match *a* followed by 24 dots followed by *z* (but not *a* followed by 23 or 25 dots followed by a *z*).

A range of numbers can also be specified; so `/{n,m}/` specifies from n to m occurrences of the previous char or expression, while `/{n,}/` means at least n occurrences of the previous expression. REs for counting are summarized in Figure 2.7.

NEWLINE

Finally, certain special characters are referred to by special notation based on the backslash (`\`). The most common of these are the **newline** character `\n` and the **tab** character `\t`. To refer to characters that are special themselves, (like `.`, `*`, `[`, and `\`), precede them with a backslash, (i.e., `/\./`, `/\*/`, `/\[/`, and `/\\/`).

RE	Match
*	zero or more occurrences of the previous char or expression
+	one or more occurrences of the previous char or expression
?	exactly zero or one occurrence of the previous char or expression
{n}	*n* occurrences of the previous char or expression
{n,m}	from *n* to *m* occurrences of the previous char or expression
{n,}	at least *n* occurrences of the previous char or expression

Figure 2.7 Regular expression operators for counting.

RE	Match	Example Patterns Matched
*	an asterisk "*"	"K*A*P*L*A*N"
\.	a period "."	"Dr. Livingston, I presume"
\?	a question mark	"Would you light my candle?"
\n	a newline	
\t	a tab	

Figure 2.8 Some characters that need to be backslashed.

The reader should consult Appendix A for further details of regular expressions, and especially for the differences between regular expressions in Perl, UNIX, and Microsoft Word.

Regular Expression Substitution, Memory, and ELIZA

SUBSTITUTION

An important use of regular expressions is in **substitutions**. For example, the Perl substitution operator `s/regexp1/regexp2/` allows a string characterized by one regular expression to be replaced by a string characterized by a different regular expression:

```
s/colour/color/
```

It is often useful to be able to refer to a particular subpart of the string matching the first pattern. For example, suppose we wanted to put angle brackets around all integers in a text, changing e.g., *the 35 boxes* to *the <35> boxes*. We'd like a way to refer back to the integer we've found so that we can easily add the brackets. To do this, we put parentheses (and) around the first pattern, and use the **number** operator `\1` in the second pattern to refer back. Here's how it looks:

```
s/([0-9]+)/<\1>/
```

The parenthesis and number operators can also be used to specify that a certain string or expression must occur twice in the text. For example, suppose we are looking for the pattern "the Xer they were, the Xer they will be", where we want to constrain the two X's to be the same string. We do this by surrounding the first X with the parenthesis operator, and replacing the second X with the number operator `\1`, as follows:

```
/the (.*)er they were, the \1er they will be/
```

Here the `\1` will be replaced by whatever string matched the first item in parentheses. So this will match *The bigger they were, the bigger they will be* but not *The bigger they were, the faster they will be*.

The number operator can be used with other numbers: if you match two different sets of parenthesis, `\2` means whatever matched the *second* set. For example

```
/the (.*)er they (.*), the \1er they \2/
```

will match *The bigger they were, the bigger they were* but not *The bigger they were, the bigger they will be*. These numbered memories are called **registers** (e.g. register 1, register 2, register 3, etc). This memory feature is not part of every regular expression language and is often considered an "extended" feature of regular expressions.

REGISTERS

Substitutions using memory are very useful in implementing a simple natural-language understanding program like ELIZA (Weizenbaum, 1966). Recall that ELIZA simulated a Rogerian psychologist and could carry on conversations with the user like the following:

$User_1$: Men are all alike.
$ELIZA_1$: IN WHAT WAY
$User_2$: They're always bugging us about something or other.
$ELIZA_2$: CAN YOU THINK OF A SPECIFIC EXAMPLE
$User_3$: Well, my boyfriend made me come here.
$ELIZA_3$: YOUR BOYFRIEND MADE YOU COME HERE
$User_4$: He says I'm depressed much of the time.
$ELIZA_4$: I AM SORRY TO HEAR YOU ARE DEPRESSED.

Eliza worked by having a cascade of regular expression substitutions that each matched some part of the input lines and changed them. The first substitutions changed all instances of *my* to *YOUR*, and *I'm* to *YOU ARE*, and so on. The next set of substitutions looked for relevant patterns in the input and created an appropriate output; here are some examples:

```
s/.* YOU ARE (depressed|sad) .*/I AM SORRY TO HEAR YOU ARE \1/
```

```
s/.* YOU ARE (depressed|sad) .*/WHY DO YOU THINK YOU ARE \1/
s/.* all .*/IN WHAT WAY/
s/.* always .*/CAN YOU THINK  OF A SPECIFIC EXAMPLE/
```

Since multiple substitutions could apply to a given input, substitutions were assigned a rank and were applied in order. Creation of such patterns is addressed in Exercise 2.2.

2.2 FINITE-STATE AUTOMATA

The regular expression is more than just a convenient metalanguage for text searching. First, a regular expression is one way of describing a **finite-state automaton (FSA)**. Finite-state automata are the theoretical foundation of a good deal of the computational work we will describe in this book. Any regular expression can be implemented as a finite-state automaton (except regular expressions that use the memory feature; more on this later). Symmetrically, any finite-state automaton can be described with a regular expression. Second, a regular expression is one way of characterizing a particular kind of formal language called a **regular language**. Both regular expressions and finite-state automata can be used to described regular languages. The relation among these three theoretical constructions is sketched out in Figure 2.9.

FINITE-STATE AUTOMATON

FSA

REGULAR LANGUAGE

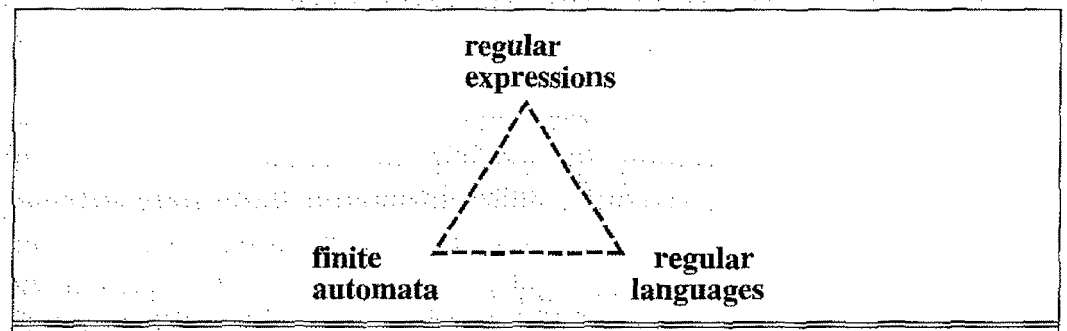


Figure 2.9 The relationship between finite automata, regular expressions, and regular languages; figure suggested by Martin Kay.

This section will begin by introducing finite-state automata for some of the regular expressions from the last section, and then suggest how the mapping from regular expressions to automata proceeds in general. Although we begin with their use for implementing regular expressions, FSAs have a wide variety of other uses that we will explore in this chapter and the next.

Using an FSA to Recognize Sheeptalk

After a while, with the parrot's help, the Doctor got to learn the language of the animals so well that he could talk to them himself and understand everything they said.

Hugh Lofting, *The Story of Doctor Dolittle*

Let's begin with the "sheep language" we discussed previously. Recall that we defined the sheep language as any string from the following (infinite) set:

baa!
baaa!
baaaa!
baaaaa!
baaaaaa!
...

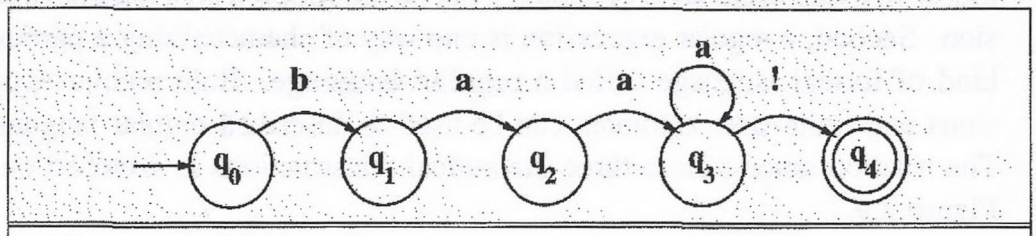


Figure 2.10 A finite-state automaton for talking sheep.

AUTOMATON

The regular expression for this kind of "sheeptalk" is `/baa+!/`. Figure 2.10 shows an **automaton** for modeling this regular expression. The automaton (i.e., machine, also called **finite automaton**, **finite-state automaton**, or **FSA**) recognizes a set of strings, in this case the strings characterizing sheep talk, in the same way that a regular expression does. We represent the automaton as a directed graph: a finite set of vertices (also called nodes), together with a set of directed links between pairs of vertices called arcs. We'll represent vertices with circles and arcs with arrows. The automaton has five **states**, which are represented by nodes in the graph. State 0 is the **start state** which we represent by the incoming arrow. State 4 is the **final state** or **accepting state**, which we represent by the double circle. It also has four **transitions**, which we represent by arcs in the graph.

STATE

START STATE

The FSA can be used for recognizing (we also say **accepting**) strings in the following way. First, think of the input as being written on a long tape

broken up into cells, with one symbol written in each cell of the tape, as in Figure 2.11.

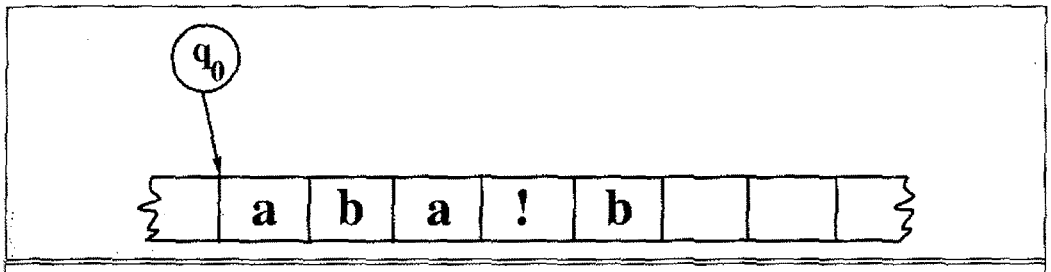


Figure 2.11 A tape with cells.

The machine starts in the start state (q_0), and iterates the following process: Check the next letter of the input. If it matches the symbol on an arc leaving the current state, then cross that arc, move to the next state, and also advance one symbol in the input. If we are in the accepting state (q_4) when we run out of input, the machine has successfully recognized an instance of sheeptalk. If the machine never gets to the final state, either because it runs out of input, or it gets some input that doesn't match an arc (as in Figure 2.11), or if it just happens to get stuck in some non-final state, we say the machine **rejects** or fails to accept an input.

REJECTS

We can also represent an automaton with a **state-transition table**. As in the graph notation, the state-transition table represents the start state, the accepting states, and what transitions leave each state with which symbols. Here's the state-transition table for the FSA of Figure 2.10.

STATE-TRANSITION TABLE

	Input		
State	b	a	!
0	1	Ø	Ø
1	Ø	2	Ø
2	Ø	3	Ø
3	Ø	3	4
4:	Ø	Ø	Ø

Figure 2.12 The state-transition table for the FSA of Figure 2.10.

We've marked state 4 with a colon to indicate that it's a final state (you can have as many final states as you want), and the Ø indicates an illegal or missing transition. We can read the first row as "if we're in state 0 and we see the input **b** we must go to state 1. If we're in state 0 and we see the input **a** or **!**, we fail".

More formally, a finite automaton is defined by the following five parameters:

- Q: a finite set of N states $q_0, q_1, \ldots, q_N$
- Σ: a finite input alphabet of symbols
- q_0: the start state
- F: the set of final states, $F \subseteq Q$
- $\delta(q,i)$: the transition function or transition matrix between states. Given a state $q \in Q$ and an input symbol $i \in \Sigma$, $\delta(q,i)$ returns a new state $q' \in Q$. δ is thus a relation from $Q \times \Sigma$ to Q;

For the sheeptalk automaton in Figure 2.10, $Q = \{q_0, q_1, q_2, q_3, q_4\}$, $\Sigma = \{a, b, !\}$, $F = \{q_4\}$, and $\delta(q,i)$ is defined by the transition table in Figure 2.12.

Figure 2.13 presents an algorithm for recognizing a string using a state-transition table. The algorithm is called D-RECOGNIZE for "deterministic recognizer". A **deterministic** algorithm is one that has no choice points; the algorithm always knows what to do for any input. The next section will introduce non-deterministic automata that must make decisions about which states to move to.

DETERMINISTIC

D-RECOGNIZE takes as input a tape and an automaton. It returns *accept* if the string it is pointing to on the tape is accepted by the automaton, and *reject* otherwise. Note that since D-RECOGNIZE assumes it is already pointing at the string to be checked, its task is only a subpart of the general problem that we often use regular expressions for, finding a string in a corpus. (The general problem is left as an exercise to the reader in Exercise 2.9.)

D-RECOGNIZE begins by initializing the variable *index* the beginning of the tape, and *current-state* to the machine's initial state. D-RECOGNIZE then enters a loop that drives the rest of the algorithm. It first checks whether it has reached the end of its input. If so, it either accepts the input (if the current state is an accept state) or rejects the input (if not).

If there is input left on the tape, D-RECOGNIZE looks at the transition table to decide which state to move to. The variable *current-state* indicates which row of the table to consult, while the current symbol on the tape indicates which column of the table to consult. The resulting transition-table cell is used to update the variable *current-state* and *index* is incremented to move forward on the tape. If the transition-table cell is empty then the machine has nowhere to go and must reject the input.

Figure 2.14 traces the execution of this algorithm on the sheep language FSA given the sample input string *baaa!*.

```
function D-RECOGNIZE(tape, machine) returns accept or reject

  index ← Beginning of tape
  current-state ← Initial state of machine
  loop
   if End of input has been reached then
    if current-state is an accept state then
      return accept
    else
      return reject
   elsif transition-table[current-state,tape[index]] is empty then
     return reject
   else
     current-state ← transition-table[current-state,tape[index]]
     index ← index + 1
  end
```

Figure 2.13 An algorithm for deterministic recognition of FSAs. This algorithm returns *accept* if the entire string it is pointing at is in the language defined by the FSA, and *reject* if the string is not in the language.

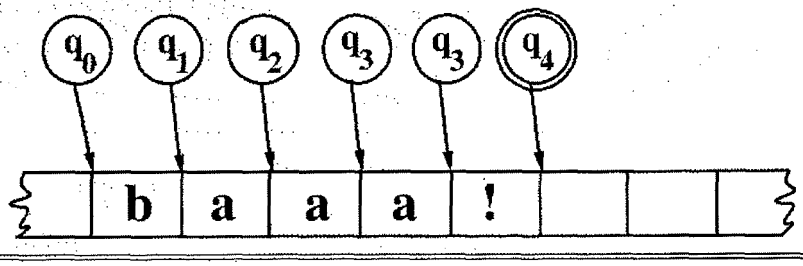


Figure 2.14 Tracing the execution of FSA #1 on some sheeptalk.

Before examining the beginning of the tape, the machine is in state q_0. Finding a *b* on input tape, it changes to state q_1 as indicated by the contents of *transition-table*[q_0,*b*] in Figure 2.12 on page 35. It then finds an *a* and switches to state q_2, another *a* puts it in state q_3, a third *a* leaves it in state q_3, where it reads the "!", and switches to state q_4. Since there is no more input, the `End of input` condition at the beginning of the loop is satisfied for the first time and the machine halts in q_4. State q_4 is an accepting state, and so the machine has accepted the string *baaa!* as a sentence in the sheep language.

The algorithm will fail whenever there is no legal transition for a given combination of state and input. The input *abc* will fail to be recognized since there is no legal transition out of state q_0 on the input a, (i.e., this entry of the transition table in Figure 2.12 on page 35 has a $\emptyset$). Even if the automaton had allowed an initial *a* it would have certainly failed on *c*, since *c* isn't even in the sheeptalk alphabet!. We can think of these "empty" elements in the table as if they all pointed at one "empty" state, which we might call the **fail state** or **sink state**. In a sense then, we could view any machine with empty transitions *as if* we had augmented it with a fail state, and drawn in all the extra arcs, so we always had somewhere to go from any state on any possible input. Just for completeness, Figure 2.15 shows the FSA from Figure 2.10 with the fail state q_F filled in.

FAIL STATE

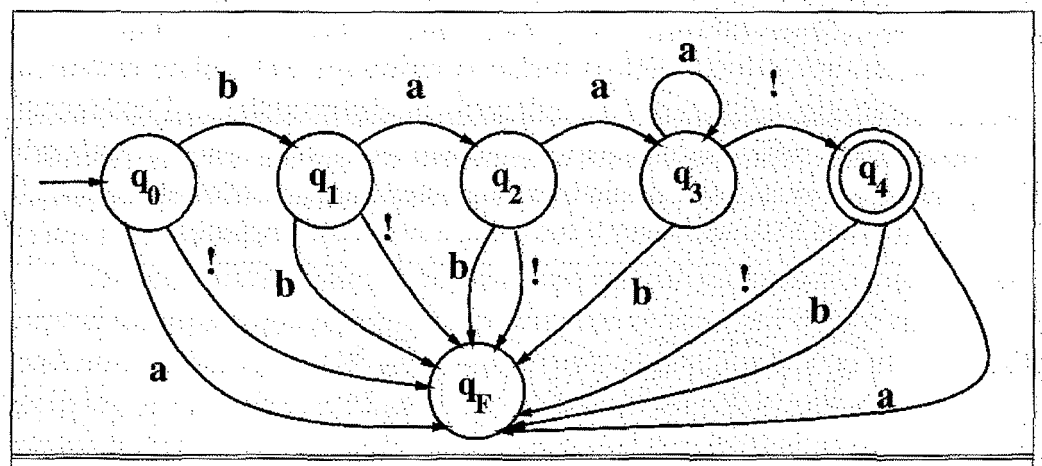


Figure 2.15 Adding a fail state to Figure 2.10.

Formal Languages

We can use the same graph in Figure 2.10 as an automaton for GENERATING sheeptalk. If we do, we would say that the automaton starts at state q_0, and crosses arcs to new states, printing out the symbols that label each arc it follows. When the automaton gets to the final state it stops. Notice that at state 3, the automaton has to chose between printing out a ! and going to state 4, or printing out an **a** and returning to state 3. Let's say for now that we don't care how the machine makes this decision; maybe it flips a coin. For now, we don't care which exact string of sheeptalk we generate, as long as it's a string captured by the regular expression for sheeptalk above.

Key Concept #1. Formal Language: A model which can both generate and recognize all and only the strings of a formal language acts as a *definition* of the formal language.

A **formal language** is a set of strings, each string composed of symbols from a finite symbol-set called an **alphabet** (the same alphabet used above for defining an automaton!). The alphabet for the sheep language is the set $\Sigma = \{a, b, !\}$. Given a model m (such as a particular FSA), we can use $L(m)$ to mean "the formal language characterized by m". So the formal language defined by our sheeptalk automaton m in Figure 2.10 (and Figure 2.12) is the infinite set:

FORMAL LANGUAGE

ALPHABET

$$L(m) = \{baa!, baaa!, baaaa!, baaaaa!, baaaaaa!, \ldots\} \quad (2.1)$$

The usefulness of an automaton for defining a language is that it can express an infinite set (such as this one above) in a closed form. Formal languages are not the same as **natural languages**, which are the kind of languages that real people speak. In fact, a formal language may bear no resemblance at all to a real language (e.g., a formal language can be used to model the different states of a soda machine). But we often use a formal language to model part of a natural language, such as parts of the phonology, morphology, or syntax. The term **generative grammar** is sometimes used in linguistics to mean a grammar of a formal language; the origin of the term is this use of an automaton to define a language by generating all possible strings.

NATURAL LANGUAGES

Another Example

In the previous examples our formal alphabet consisted of letters; but we can also have a higher level alphabet consisting of words. In this way we can write finite-state automata that model facts about word combinations. For example, suppose we wanted to build an FSA that modeled the subpart of English dealing with amounts of money. Such a formal language would model the subset of English consisting of phrases like *ten cents*, *three dollars*, *one dollar thirty-five cents* and so on.

We might break this down by first building just the automaton to account for the numbers from 1 to 99, since we'll need them to deal with cents. Figure 2.16 shows this.

We could now add *cents* and *dollars* to our automaton. Figure 2.17 shows a simple version of this, where we just made two copies of the automaton in Figure 2.16 and appended the words *cents* and *dollars*.

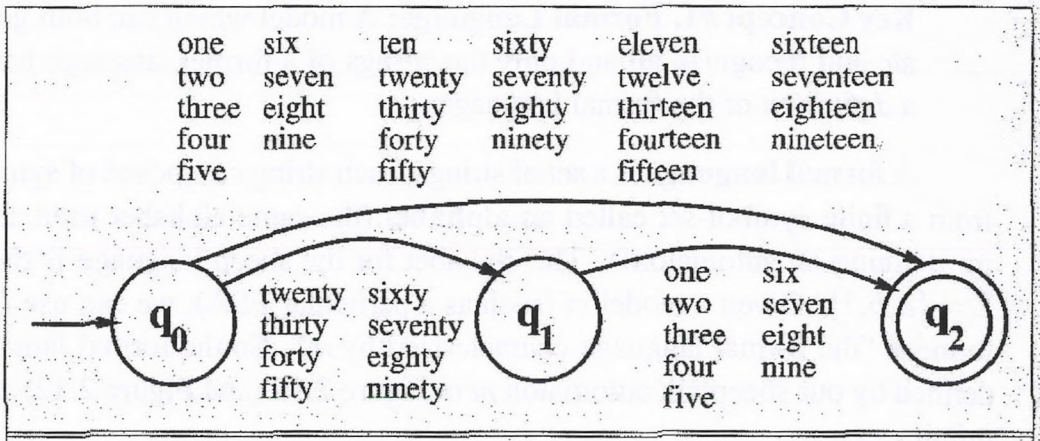


Figure 2.16 An FSA for the words for English numbers 1–99.

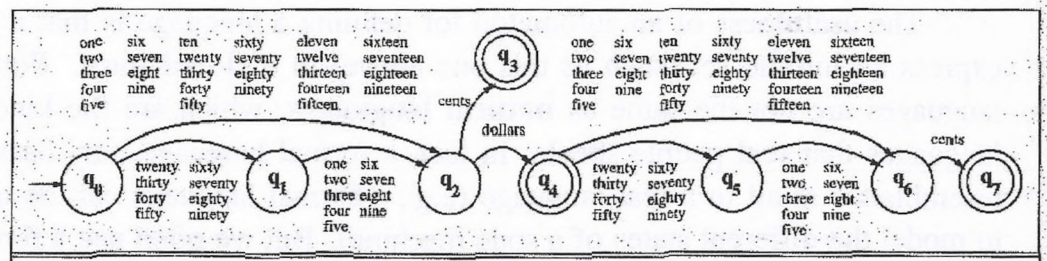


Figure 2.17 FSA for the simple dollars and cents.

We would now need to add in the grammar for different amounts of dollars; including higher numbers like *hundred*, *thousand*. We'd also need to make sure that the nouns like *cents* and *dollars* are singular when appropriate (*one cent*, *one dollar*), and plural when appropriate (*ten cents*, *two dollars*). This is left as an exercise for the reader (Exercise 2.3). We can think of the FSAs in Figure 2.16 and Figure 2.17 as simple grammars of parts of English. We will return to grammar-building in Part II of this book, particularly in Chapter 9.

Non-Deterministic FSAs

Let's extend our discussion now to another class of FSAs: **non-deterministic FSAs** (or **NFSAs**). Consider the sheeptalk automaton in Figure 2.18, which is much like our first automaton in Figure 2.10:

The only difference between this automaton and the previous one is that here in Figure 2.18 the self-loop is on state 2 instead of state 3. Consider using this network as an automaton for recognizing sheeptalk. When we get to state 2, if we see an **a** we don't know whether to remain in state

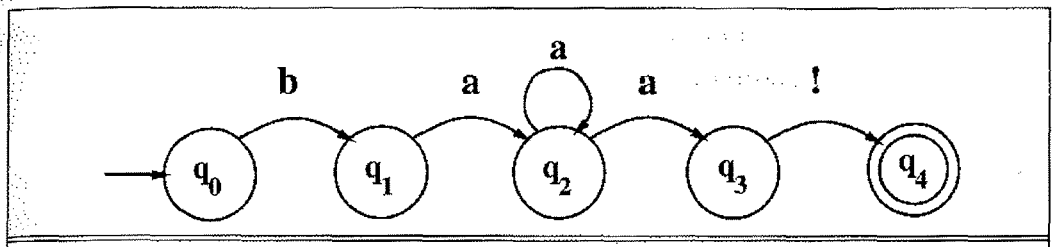


Figure 2.18 A non-deterministic finite-state automaton for talking sheep (NFSA #1). Compare with the deterministic automaton in Figure 2.10.

2 or go on to state 3. Automata with decision points like this are called **non-deterministic FSAs** (or **NFSAs**). Recall by contrast that Figure 2.10 specified a **deterministic** automaton, i.e., one whose behavior during recognition is fully *determined* by the state it is in and the symbol it is looking at. A deterministic automaton can be referred to as a **DFSA**. That is not true for the machine in Figure 2.18 (NFSA #1).

NON-DETERMINISTIC
NFSA
DFSA

There is another common type of non-determinism, caused by arcs that have no symbols on them (called **ε-transitions**). The automaton in Figure 2.19 defines the exact same language as the last one, or our first one, but it does it with an ε-transition.

ε-TRANSITION

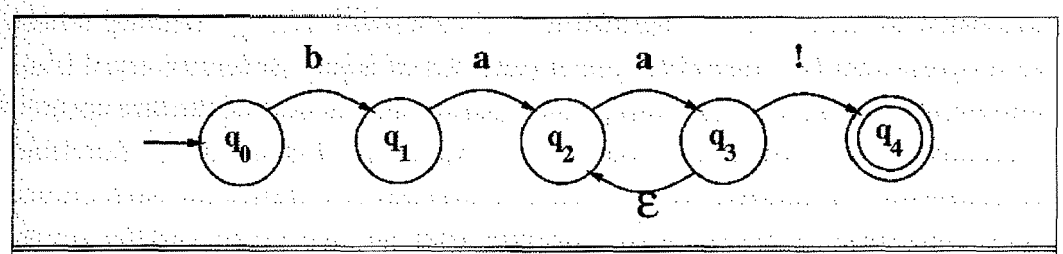


Figure 2.19 Another NFSA for the sheep language (NFSA #2). It differs from NFSA #1 in Figure 2.18 in having an ε-transition.

We interpret this new arc as follows: If we are in state 3, we are allowed to move to state 2 *without* looking at the input, or advancing our input pointer. So this introduces another kind of non-determinism — we might not know whether to follow the ε-transition or the ! arc.

Using an NFSA to Accept Strings

If we want to know whether a string is an instance of sheeptalk or not, and if we use a non-deterministic machine to recognize it, we might follow the wrong arc and reject it when we should have accepted it. That is, since there is more than one choice at some point, we might take the wrong choice. This

problem of choice in non-deterministic models will come up again and again as we build computational models, particularly for parsing.

There are three standard solutions to this problem:

- **Backup:** Whenever we come to a choice point, we could put a *marker* to mark where we were in the input, and what state the automaton was in. Then if it turns out that we took the wrong choice, we could back up and try another path.
- **Look-ahead:** We could look ahead in the input to help us decide which path to take.
- **Parallelism:** Whenever we come to a choice point, we could look at every alternative path in parallel.

We will focus here on the backup approach and defer discussion of the look-ahead and parallelism approaches to later chapters.

The backup approach suggests that we should blithely make choices that might lead to deadends, knowing that we can always return to unexplored alternative choices. There are two keys to this approach: we need to remember all the alternatives for each choice point, and we need to store sufficient information about each alternative so that we can return to it when necessary. When a backup algorithm reaches a point in its processing where no progress can be made (because it runs out of input, or has no legal transitions), it returns to a previous choice point, selects one of the unexplored alternatives, and continues from there. Applying this notion to our non-deterministic recognizer, we need only remember two things for each choice point: the state, or node, of the machine that we can go to and the corresponding position on the tape. We will call the combination of the node and position the **search-state** of the recognition algorithm. To avoid confusion, we will refer to the state of the automaton (as opposed to the state of the search) as a **node** or a **machine-state**. Figure 2.21 presents a recognition algorithm based on this approach.

SEARCH-STATE

Before going on to describe the main part of this algorithm, we should note two changes to the transition table that drives it. First, in order to represent nodes that have outgoing ε-transitions, we add a new **ε-column** to the transition table. If a node has an ε-transition, we list the destination node in the ε-column for that node's row. The second addition is needed to account for multiple transitions to different nodes from the same input symbol. We let each cell entry consist of a list of destination nodes rather than a single node. Figure 2.20 shows the transition table for the machine in Figure 2.18 (NFSA #1). While it has no ε-transitions, it does show that in machine-state

	Input			
State	b	a	!	ε
0	1	∅	∅	∅
1	∅	2	∅	∅
2	∅	2,3	∅	∅
3	∅	∅	4	∅
4:	∅	∅	∅	∅

Figure 2.20 The transition table from NFSA #1 in Figure 2.18.

q_2 the input a can lead back to q_2 or on to q_3.

Figure 2.21 shows the algorithm for using a non-deterministic FSA to recognize an input string. The function ND-RECOGNIZE uses the variable *agenda* to keep track of all the currently unexplored choices generated during the course of processing. Each choice (search state) is a tuple consisting of a node (state) of the machine and a position on the tape. The variable *current-search-state* represents the branch choice being currently explored.

ND-RECOGNIZE begins by creating an initial search-state and placing it on the agenda. For now we don't specify what order the search-states are placed on the agenda. This search-state consists of the initial machine-state of the machine and a pointer to the beginning of the tape. The function NEXT is then called to retrieve an item from the agenda and assign it to the variable *current-search-state*.

As with D-RECOGNIZE, the first task of the main loop is to determine if the entire contents of the tape have been successfully recognized. This is done via a call to ACCEPT-STATE?, which returns *accept* if the current search-state contains both an accepting machine-state and a pointer to the end of the tape. If we're not done, the machine generates a set of possible next steps by calling GENERATE-NEW-STATES, which creates search-states for any ε-transitions and any normal input-symbol transitions from the transition table. All of these search-state tuples are then added to the current agenda.

Finally, we attempt to get a new search-state to process from the agenda. If the agenda is empty we've run out of options and have to reject the input. Otherwise, an unexplored option is selected and the loop continues.

It is important to understand why ND-RECOGNIZE returns a value of reject only when the agenda is found to be empty. Unlike D-RECOGNIZE, it does not return reject when it reaches the end of the tape in an non-accept machine-state or when it finds itself unable to advance the tape from some

machine-state. This is because, in the non-deterministic case, such road-blocks only indicate failure down a given path, not overall failure. We can only be sure we can reject a string when all possible choices have been examined and found lacking.

```
function ND-RECOGNIZE(tape, machine) returns accept or reject

  agenda ← {(Initial state of machine, beginning of tape)}
  current-search-state ← NEXT(agenda)
  loop
    if ACCEPT-STATE?(current-search-state) returns true then
      return accept
    else
      agenda ← agenda ∪ GENERATE-NEW-STATES(current-search-state)
    if agenda is empty then
      return reject
    else
      current-search-state ← NEXT(agenda)
  end

function GENERATE-NEW-STATES(current-state) returns a set of search-states

  current-node ← the node the current search-state is in
  index ← the point on the tape the current search-state is looking at
  return a list of search states from transition table as follows:
    (transition-table[current-node,ε], index)
    ∪
    (transition-table[current-node, tape[index]], index + 1)

function ACCEPT-STATE?(search-state) returns true or false

  current-node ← the node search-state is in
  index ← the point on the tape search-state is looking at
  if index is at the end of the tape and current-node is an accept state of machine
then
    return true
  else
    return false
```

Figure 2.21 An algorithm for NFSA recognition. The word *node* means a state of the FSA, while *state* or *search-state* means "the state of the search process", i.e., a combination of *node* and *tape-position*.

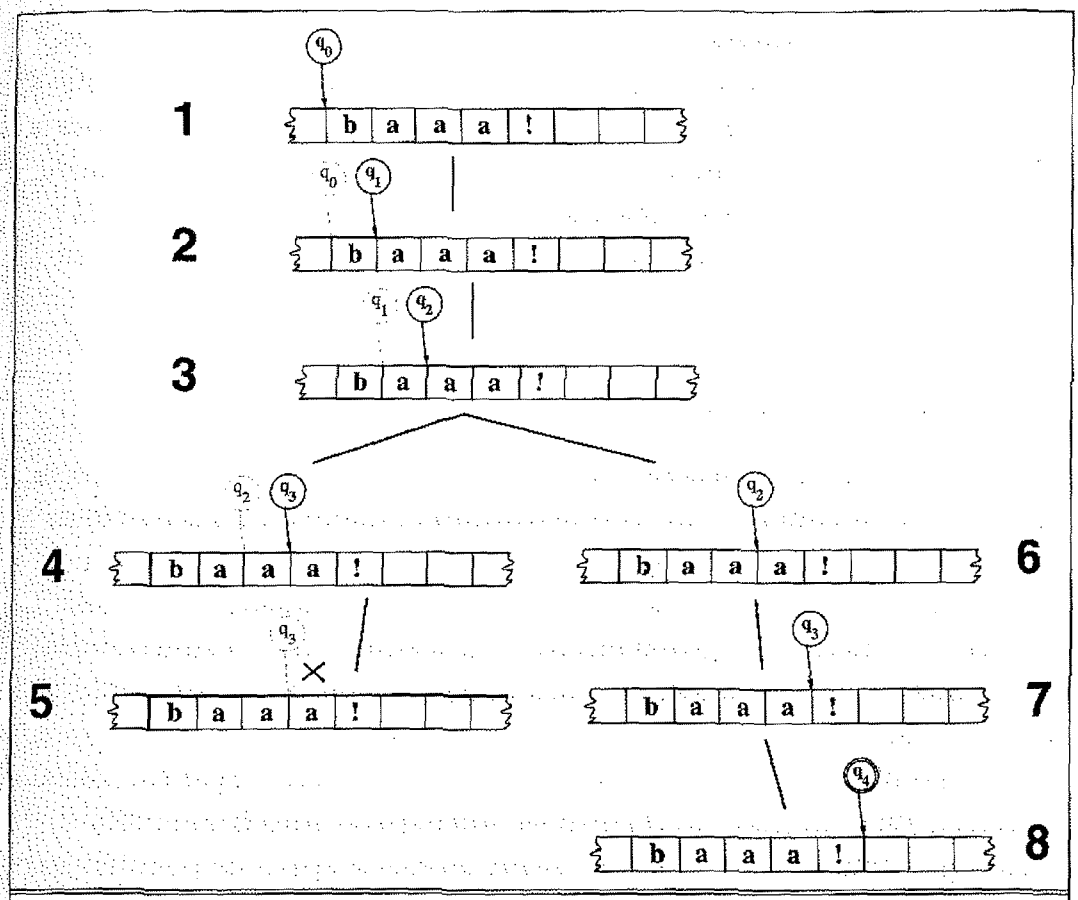


Figure 2.22 Tracing the execution of NFSA #1 (Figure 2.18) on some sheeptalk.

Figure 2.22 illustrates the progress of ND-RECOGNIZE as it attempts to handle the input `baaa!`. Each strip illustrates the state of the algorithm at a given point in its processing. The *current-search-state* variable is captured by the solid bubbles representing the machine-state along with the arrow representing progress on the tape. Each strip lower down in the figure represents progress from one *current-search-state* to the next.

Little of interest happens until the algorithm finds itself in state q_2 while looking at the second a on the tape. An examination of the entry for transition-table[q_2,a] returns both q_2 and q_3. Search states are created for each of these choices and placed on the agenda. Unfortunately, our algorithm chooses to move to state q_3, a move that results in neither an accept state nor any new states since the entry for transition-table[q_3, a] is empty. At this point, the algorithm simply asks the agenda for a new state to pursue. Since the choice of returning to q_2 from q_2 is the only unexamined choice on the agenda it is returned with the tape pointer advanced to the next a. Some-

what diabolically, ND-RECOGNIZE finds itself faced with the same choice. The entry for transition-table[q_2,a] still indicates that looping back to q_2 or advancing to q_3 are valid choices. As before, states representing both are placed on the agenda. These search states are not the same as the previous ones since their tape index values have advanced. This time the agenda provides the move to q_3 as the next move. The move to q_4, and success, is then uniquely determined by the tape and the transition-table.

Recognition as Search

ND-RECOGNIZE accomplishes the task of recognizing strings in a regular language by providing a way to systematically explore all the possible paths through a machine. If this exploration yields a path ending in an accept state, it accepts the string, otherwise it rejects it. This systematic exploration is made possible by the agenda mechanism, which on each iteration selects a partial path to explore and keeps track of any remaining, as yet unexplored, partial paths.

STATE-SPACE SEARCH

Algorithms such as ND-RECOGNIZE, which operate by systematically searching for solutions, are known as **state-space search** algorithms. In such algorithms, the problem definition creates a space of possible solutions; the goal is to explore this space, returning an answer when one is found or rejecting the input when the space has been exhaustively explored. In ND-RECOGNIZE, search states consist of pairings of machine-states with positions on the input tape. The state-space consists of all the pairings of machine-state and tape positions that are possible given the machine in question. The goal of the search is to navigate through this space from one state to another looking for a pairing of an accept state with an end of tape position.

The key to the effectiveness of such programs is often the *order* in which the states in the space are considered. A poor ordering of states may lead to the examination of a large number of unfruitful states before a successful solution is discovered. Unfortunately, it is typically not possible to tell a good choice from a bad one, and often the best we can do is to insure that each possible solution is eventually considered.

Careful readers may have noticed that the ordering of states in ND-RECOGNIZE has been left unspecified. We know only that unexplored states are added to the agenda as they are created and that the (undefined) function NEXT returns an unexplored state from the agenda when asked. How should the function NEXT be defined? Consider an ordering strategy where the states that are considered next are the most recently created ones. Such

a policy can be implemented by placing newly created states at the front of the agenda and having NEXT return the state at the front of the agenda when called. Thus the agenda is implemented by a **stack**. This is commonly referred to as a **depth-first search** or **Last In First Out (LIFO)** strategy. DEPTH-FIRST

Such a strategy dives into the search space following newly developed leads as they are generated. It will only return to consider earlier options when progress along a current lead has been blocked. The trace of the execution of ND-RECOGNIZE on the string `baaa!` as shown in Figure 2.22 illustrates a depth-first search. The algorithm hits the first choice point after seeing `ba` when it has to decide whether to stay in q_2 or advance to state q_3. At this point, it chooses one alternative and follows it until it is sure it's wrong. The algorithm then backs up and tries another older alternative.

Depth first strategies have one major pitfall: under certain circumstances they can enter an infinite loop. This is possible either if the search space happens to be set up in such a way that a search-state can be accidentally re-visited, or if there are an infinite number of search states. We will revisit this question when we turn to more complicated search problems in parsing in Chapter 10.

The second way to order the states in the search space is to consider states in the order in which they are created. Such a policy can be implemented by placing newly created states at the back of the agenda and still have NEXT return the state at the front of the agenda. Thus the agenda is implemented via a **queue**. This is commonly referred to as a **breadth-first search** or **First In First Out (FIFO)** strategy. Consider a different trace of the execution of ND-RECOGNIZE on the string `baaa!` as shown in Figure 2.23. Again, the algorithm hits its first choice point after seeing `ba` when it had to decide whether to stay in q_2 or advance to state q_3. But now rather than picking one choice and following it up, we imagine examining all possible choices, expanding one ply of the search tree at a time. BREADTH-FIRST

Like depth-first search, breadth-first search has its pitfalls. As with depth-first if the state-space is infinite, the search may never terminate. More importantly, due to growth in the size of the agenda if the state-space is even moderately large, the search may require an impractically large amount of memory. For small problems, either depth-first or breadth-first search strategies may be adequate, although depth-first is normally preferred for its more efficient use of memory. For larger problems, more complex search techniques such as **dynamic programming** or **A*** must be used, as we will see in Chapters 7 and 10.

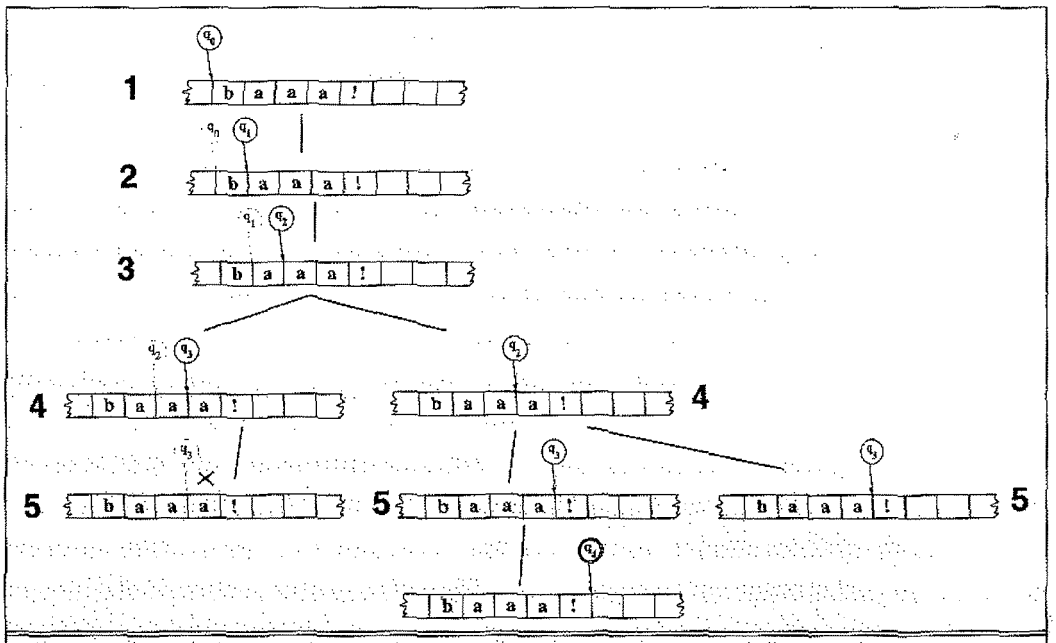


Figure 2.23 A breadth-first trace of FSA #1 on some sheeptalk.

Relating Deterministic and Non-Deterministic Automata

It may seem that allowing NFSAs to have non-deterministic features like ε-transitions would make them more powerful than DFSAs. In fact this is not the case; for any NFSA, there is an exactly equivalent DFSA. In fact there is a simple algorithm for converting an NFSA to an equivalent DFSA, although the number of states in this equivalent deterministic automaton may be much larger. See Lewis and Papadimitriou (1981) or Hopcroft and Ullman (1979) for the proof of the correspondence. The basic intuition of the proof is worth mentioning, however, and builds on the way NFSAs parse their input. Recall that the difference between NFSAs and DFSAs is that in an NFSA a state q_i may have more than one possible next state given an input i (for example q_a and q_b). The algorithm in Figure 2.21 dealt with this problem by choosing either q_a or q_b and then *backtracking* if the choice turned out to be wrong. We mentioned that a parallel version of the algorithm would follow both paths (toward q_a and q_b) simultaneously.

The algorithm for converting a NFSA to a DFSA is like this parallel algorithm; we build an automaton that has a deterministic path for every path our parallel recognizer might have followed in the search space. We imagine following both paths simultaneously, and group together into an equivalence class all the states we reach on the same input symbol (i.e., q_a and q_b). We now give a new state label to this new equivalence class state (for example

q_{ab}). We continue doing this for every possible input for every possible group of states. The resulting DFSA can have as many states as there are distinct sets of states in the original NFSA. The number of different subsets of a set with N elements is 2^N, hence the new DFSA can have as many as 2^N states.

2.3 Regular Languages and FSAs

As we suggested above, the class of languages that are definable by regular expressions is exactly the same as the class of languages that are characterizable by finite-state automata (whether deterministic or non-deterministic). Because of this, we call these languages the **regular languages**. In order to give a formal definition of the class of regular languages, we need to refer back to two earlier concepts: the alphabet Σ, which is the set of all symbols in the language, and the *empty string* ε, which is conventionally not included in Σ. In addition, we make reference to the *empty set* $\emptyset$ (which is distinct from ε). The class of regular languages (or **regular sets**) over Σ is then formally defined as follows: [1]

REGULAR LANGUAGES

1. $\emptyset$ is a regular language
2. $\forall a \in \Sigma \cup \varepsilon$, $\{a\}$ is a regular language
3. If L_1 and L_2 are regular languages, then so are:
   (a) $L_1 \cdot L_2 = \{xy \mid x \in L_1, y \in L_2\}$, the **concatenation** of L_1 and L_2
   (b) $L_1 \cup L_2$, the **union** or **disjunction** of L_1 and L_2
   (c) L_1^*, the **Kleene closure** of L_1

All and only the sets of languages which meet the above properties are regular languages. Since the regular languages are the set of languages characterizable by regular expressions, it must be the case that all the regular expression operators introduced in this chapter (except memory) can be implemented by the three operations which define regular languages: concatenation, disjunction/union (also called "|"), and Kleene closure. For example all the counters (*,+, `{n,m}`) are just a special case of repetition plus Kleene *. All the anchors can be thought of as individual special symbols. The square braces `[]` are a kind of disjunction (i.e., `[ab]` means "a or b", or the disjunction of a and b). Thus it is true that any regular expression can be turned into a (perhaps larger) expression which only makes use of the three primitive operations.

[1] Following van Santen and Sproat (1998), Kaplan and Kay (1994), and Lewis and Papadimitriou (1981).

Regular languages are also closed under the following operations (Σ^* means the infinite set of all possible strings formed from the alphabet Σ):

- **intersection**: if L_1 and L_2 are regular languages, then so is $L_1 \cap L_2$, the language consisting of the set of strings that are in both L_1 and L_2.
- **difference**: if L_1 and L_2 are regular languages, then so is $L_1 - L_2$, the language consisting of the set of strings that are in L_1 but not L_2.
- **complementation**: If L_1 is a regular language, then so is $\Sigma^* - L_1$, the set of all possible strings that aren't in L_1.
- **reversal**: If L_1 is a regular language, then so is L_1^R, the language consisting of the set of reversals of all the strings in L_1.

The proof that regular expressions are equivalent to finite-state automata can be found in Hopcroft and Ullman (1979), and has two parts: showing that an automaton can be built for each regular language, and conversely that a regular language can be built for each automaton. We won't give the proof, but we give the intuition by showing how to do the first part: take any regular expression and build an automaton from it. The intuition is inductive: for the base case we build an automaton to correspond to regular expressions of a single symbol (e.g., the expression a) by creating an initial state and an accepting final state, with an arc between them labeled a. For the inductive step, we show that each of the primitive operations of a regular expression (concatenation, union, closure) can be imitated by an automaton:

- **concatenation**: We just string two FSAs next to each other by connecting all the final states of FSA_1 to the initial state of FSA_2 by an ε-transition.

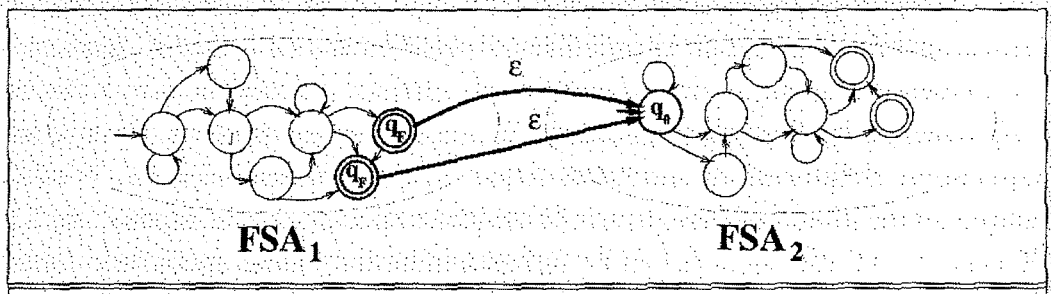


Figure 2.24 The concatenation of two FSAs.

- **closure**: We connect all the final states of the FSA back to the initial states by ε-transitions (this implements the repetition part of the Kleene *), and then put direct links between the initial and final states by ε-

transitions (this implements the possibly of having *zero* occurrences). We'd leave out this last part to implement Kleene-plus instead.

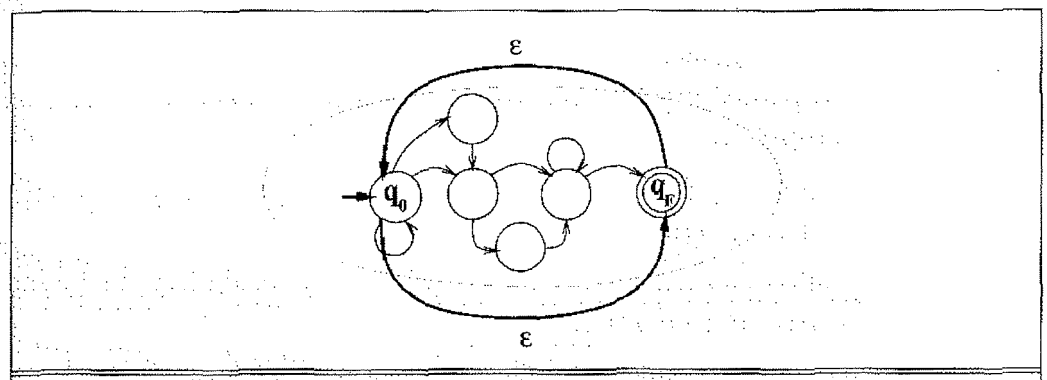


Figure 2.25 The closure (Kleene *) of an FSA.

- **union**: We add a single new initial state q_0', and add new transitions from it to all the former initial states of the two machines to be joined.

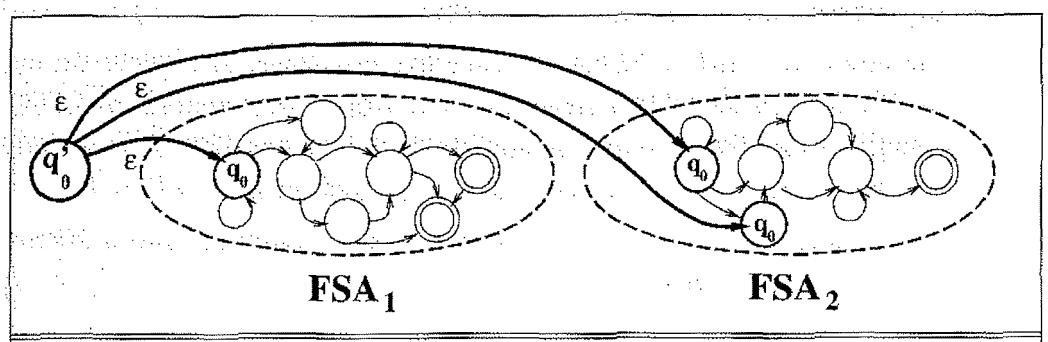


Figure 2.26 The union (|) of two FSAs.

2.4 SUMMARY

This chapter introduced the most important fundamental concept in language processing, the **finite automaton**, and the practical tool based on automaton, the **regular expression**. Here's a summary of the main points we covered about these ideas:

- The **regular expression** language is a powerful tool for pattern-matching.
- Basic operations in regular expressions include **concatenation** of symbols, **disjunction** of symbols ([], |, and .), **counters** (*, +, and

{n,m}), **anchors** (^, $) and precedence operators ((,)).

- Any regular expression can be realized as a **finite state automaton** (**FSA**).
- Memory (\1 together with ()) is an advanced operation that is often considered part of regular expressions, but which cannot be realized as a finite automaton.
- An automaton implicitly defines a **formal language** as the set of strings the automaton **accepts**.
- An automaton can use any set of symbols for its vocabulary, including letters, words, or even graphic images.
- The behavior of a **deterministic** automaton (**DFSA**) is fully determined by the state it is in.
- A **non-deterministic** automaton (**NFSA**) sometimes has to make a choice between multiple paths to take given the same current state and next input.
- Any **NFSA** can be converted to a **DFSA**.
- The order in which a **NFSA** chooses the next state to explore on the agenda defines its **search strategy**. The **depth-first search** or **LIFO** strategy corresponds to the agenda-as-stack; the **breadth-first search** or **FIFO** strategy corresponds to the agenda-as-queue.
- Any regular expression can be automatically compiled into a **NFSA** and hence into a **FSA**.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

Finite automata arose in the 1950s out of Turing's (1936) model of algorithmic computation, considered by many to be the foundation of modern computer science. The Turing machine was an abstract machine with a finite control and an input/output tape. In one move, the Turing machine could read a symbol on the tape, write a different symbol on the tape, change state, and move left or right. (Thus the Turing machine differs from a finite-state automaton mainly in its ability to change the symbols on its tape).

MCCULLOCH-PITTS NEURON

Inspired by Turing's work, McCulloch and Pitts built an automata-like model of the neuron (see von Neumann, 1963, p. 319). Their model, which is now usually called the **McCulloch-Pitts neuron** (McCulloch and Pitts, 1943), was a simplified model of the neuron as a kind of "computing ele-

ment" that could be described in terms of propositional logic. The model was a binary device, at any point either active or not, which took excitatory and inhibitatory input from other neurons and fired if its activation passed some fixed threshold. Based on the McCulloch-Pitts neuron, Kleene (1951) and (1956) defined the finite automaton and regular expressions, and proved their equivalence. Non-deterministic automata were introduced by Rabin and Scott (1959), who also proved them equivalent to deterministic ones.

Ken Thompson was one of the first to build regular expressions compilers into editors for text searching (Thompson, 1968). His editor *ed* included a command "g/regular expression/p", or Global Regular Expression Print, which later became the UNIX `grep` utility.

There are many general-purpose introductions to the mathematics underlying automata theory; such as Hopcroft and Ullman (1979) and Lewis and Papadimitriou (1981). These cover the mathematical foundations the simple automata of this chapter, as well as the finite-state transducers of Chapter 3, the context-free grammars of Chapter 9, and the Chomsky hierarchy of Chapter 13. Friedl (1997) is a very useful comprehensive guide to the advanced use of regular expressions.

The metaphor of problem-solving as search is basic to Artificial Intelligence (AI); more details on search can be found in any AI textbook such as Russell and Norvig (1995).

EXERCISES

2.1 Write regular expressions for the following languages: You may use either Perl notation or the minimal "algebraic" notation of Section 2.3, but make sure to say which one you are using. By "word", we mean an alphabetic string separated from other words by white space, any relevant punctuation, line breaks, and so forth.

a. the set of all alphabetic strings.

b. the set of all lowercase alphabetic strings ending in a *b*.

c. the set of all strings with two consecutive repeated words (e.g., "Humbert Humbert" and "the the" but not "the bug" or "the big bug").

d. the set of all strings from the alphabet a, b such that each a is immediately preceded and immediately followed by a b.

e. all strings which start at the beginning of the line with an integer (i.e., 1,2,3,...,10,...,10000,...) and which end at the end of the line with a word.

f. all strings which have both the word *grotto* and the word *raven* in them. (but not, for example, words like *grottos* that merely *contain* the word *grotto*).

g. write a pattern which places the first word of an English sentence in a register. Deal with punctuation.

2.2 Implement an ELIZA-like program, using substitutions such as those described on page 32. You may choose a different domain than a Rogerian psychologist, if you wish, although keep in mind that you would need a domain in which your program can legitimately do a lot of simple repeating-back.

2.3 Complete the FSA for English money expressions in Figure 2.16 as suggested in the text following the figure. You should handle amounts up to $100,000, and make sure that "cent" and "dollar" have the proper plural endings when appropriate.

2.4 Design an FSA that recognizes simple date expressions like *March 15, the 22nd of November, Christmas*. You should try to include all such "absolute" dates, (e.g. not "deictic" ones relative to the current day like *the day before yesterday*). Each edge of the graph should have a word or a set of words on it. You should use some sort of shorthand for classes of words to avoid drawing too many arcs (e.g., furniture → desk, chair, table).

2.5 Now extend your date FSA to handle deictic expressions like *yesterday, tomorrow, a week from tomorrow, the day before yesterday, Sunday, next Monday, three weeks from Saturday*.

2.6 Write an FSA for time-of-day expressions like *eleven o'clock, twelve-thirty, midnight*, or *a quarter to ten* and others.

2.7 (Due to Pauline Welby; this problem probably requires the ability to knit.) Write a regular expression (or draw an FSA) which matches all knitting patterns for scarves with the following specification: *32 stitches wide, K1P1 ribbing on both ends, stockinette stitch body, exactly two raised stripes*. All knitting patterns must include a cast-on row (to put the correct number of

stitches on the needle) and a bind-off row (to end the pattern and prevent unraveling). Here's a sample pattern for one possible scarf matching the above description:[2]

1.	Cast on 32 stitches.	*cast on; puts stitches on needle*
2.	K1 P1 across row (i.e. do (K1 P1) 16 times).	*K1P1 ribbing*
3.	Repeat instruction 2 seven more times.	*adds length*
4.	K32, P32.	*stockinette stitch*
5.	Repeat instruction 4 an additional 13 times.	*adds length*
6.	P32, P32.	*raised stripe stitch*
7.	K32, P32.	*stockinette stitch*
8.	Repeat instruction 7 an additional 251 times.	*adds length*
9.	P32, P32.	*raised stripe stitch*
10.	K32, P32.	*stockinette stitch*
11.	Repeat instruction 10 an additional 13 times.	*adds length*
12.	K1 P1 across row.	*K1P1 ribbing*
13.	Repeat instruction 12 an additional 7 times.	*adds length*
14.	Bind off 32 stitches.	*binds off row: ends pattern*

2.8 Write a regular expression for the language accepted by the NFSA in Figure 2.27.

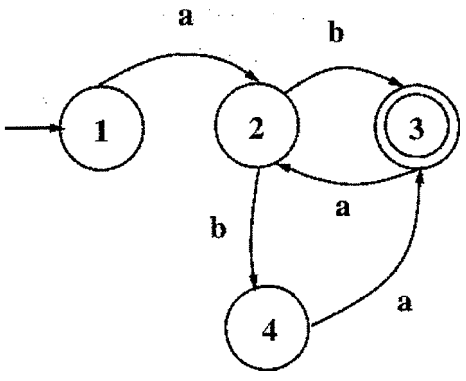


Figure 2.27 A mystery language

2.9 Currently the function D-RECOGNIZE in Figure 2.13 only solves a subpart of the important problem of finding a string in some text. Extend the algorithm to solve the following two deficiencies: (1) D-RECOGNIZE currently assumes that it is already pointing at the string to be checked, and (2)

[2] *Knit* and *purl* are two different types of stitches. The notation K*n* means do *n* knit stitches. Similarly for purl stitches. Ribbing has a striped texture—most sweaters have ribbing at the sleeves, bottom, and neck. Stockinette stitch is a series of knit and purl rows that produces a plain pattern— socks or stockings are knit with this basic pattern, hence the name.

D-RECOGNIZE fails if the string it is pointing includes as a proper substring a legal string for the FSA. That is, D-RECOGNIZE fails if there is an extra character at the end of the string.

2.10 Give an algorithm for negating a deterministic FSA. The negation of an FSA accepts exactly the set of strings that the original FSA rejects (over the same alphabet), and rejects all the strings that the original FSA accepts.

2.11 Why doesn't your previous algorithm work with NFSAs? Now extend your algorithm to negate an NFSA.

3 MORPHOLOGY AND FINITE-STATE TRANSDUCERS

A writer is someone who writes, and a stinger is something that stings. But fingers don't fing, grocers don't groce, haberdashers don't haberdash, hammers don't ham, and humdingers don't humding.

Richard Lederer, *Crazy English*

Chapter 2 introduced the regular expression, showing for example how a single search string could help a web search engine find both *woodchuck* and *woodchucks*. Hunting for singular or plural woodchucks was easy; the plural just tacks an *s* on to the end. But suppose we were looking for another fascinating woodland creatures; let's say a *fox*, and a *fish*, that surly *peccary* and perhaps a Canadian *wild goose*. Hunting for the plurals of these animals takes more than just tacking on an *s*. The plural of *fox* is *foxes*; of *peccary*, *peccaries*; and of *goose*, *geese*. To confuse matters further, fish don't usually change their form when they are plural (as Dr. Seuss points out: *one fish two fish, red fish, blue fish*).

It takes two kinds of knowledge to correctly search for singulars and plurals of these forms. **Spelling rules** tell us that English words ending in *-y* are pluralized by changing the *-y* to *-i-* and adding an *-es*. **Morphological rules** tell us that *fish* has a null plural, and that the plural of *goose* is formed by changing the vowel.

The problem of recognizing that *foxes* breaks down into the two morphemes *fox* and *-es* is called **morphological parsing**.

Key Concept #2. Parsing means taking an input and producing some sort of structure for it. PARSING

We will use the term parsing very broadly throughout this book, including many kinds of structures that might be produced; morphological, syntactic,

semantic, pragmatic; in the form of a string, or a tree, or a network. In the information retrieval domain, the similar (but not identical) problem of
STEMMING mapping from *foxes* to *fox* is called **stemming**. Morphological parsing or stemming applies to many affixes other than plurals; for example we might need to take any English verb form ending in *-ing* (*going*, *talking*, *congratulating*) and parse it into its verbal stem plus the *-ing* morpheme. So given
SURFACE the **surface** or **input form** *going*, we might want to produce the parsed form `VERB-go + GERUND-ing`. This chapter will survey the kinds of morphological knowledge that needs to be represented in different languages and introduce the main component of an important algorithm for morphological parsing: the **finite-state transducer**.

Why don't we just list all the plural forms of English nouns, and all the *-ing* forms of English verbs in the dictionary? The major reason is that *-ing*
PRODUCTIVE is a **productive** suffix; by this we mean that it applies to every verb. Similarly *-s* applies to almost every noun. So the idea of listing every noun and verb can be quite inefficient. Furthermore, productive suffixes even apply to new words (so the new word *fax* automatically can be used in the *-ing* form: *faxing*). Since new words (particularly acronyms and proper nouns) are created every day, the class of nouns in English increases constantly, and we need to be able to add the plural morpheme *-s* to each of these. Additionally, the plural form of these new nouns depends on the spelling/pronunciation of the singular form; for example if the noun ends in *-z* then the plural form is *-es* rather than *-s*. We'll need to encode these rules somewhere. Finally, we certainly cannot list all the morphological variants of every word in morphologically complex languages like Turkish, which has words like the following:

(3.1) uygarlaştıramadıklarımızdanmışsınızcasına

uygar *+laş* *+tır* *+ama* *+dık* *+lar* *+ımız*
civilized +BEC +CAUS +NEGABLE +PPART +PL +P1PL
+dan *+mış* *+sınız* *+casına*
+ABL +PAST +2PL +AsIf
"(behaving) as if you are among those whom we could not civilize/cause to become civilized"

The various pieces of this word (the **morphemes**) have these meanings:

+BEC	is "become" in English
+CAUS	is the causative voice marker on a verb
+NEGABLE	is "not able" in English

+PPart	marks a past participle form
+P1PL	is 1st person pl possessive agreement
+2PL	is 2nd person pl
+ABL	is the ablative (from/among) case marker
+AsIf	is a derivational marker that forms an adverb from a finite verb form

In such languages we clearly need to parse the input since it is impossible to store every possible word. Kemal Oflazer (personal communication), who came up with this example, notes that verbs in Turkish have 40,000 forms not counting derivational suffixes; adding derivational suffixes allows a theoretically infinite number of words. This is true because, for example, any verb can be "causativized" like the example above, and multiple instances of causativization can be embedded in a single word (*You cause X to cause Y to ... do W*). Not all Turkish words look like this; Oflazer finds that the average Turkish word has about three morphemes (a root plus two suffixes). Even so, the fact that such words are possible means that it will be difficult to store all possible Turkish words in advance.

Morphological parsing is necessary for more than just information retrieval. We will need it in machine translation to realize that the French words *va* and *aller* should both translate to forms of the English verb *go*. We will also need it in spell checking; as we will see, it is morphological knowledge that will tell us that *misclam* and *antiundoggingly* are not words.

The next sections will summarize morphological facts about English and then introduce the **finite-state transducer**.

3.1 Survey of (Mostly) English Morphology

Morphology is the study of the way words are built up from smaller meaning-bearing units, **morphemes**. A morpheme is often defined as the minimal meaning-bearing unit in a language. So for example the word *fox* consists of a single morpheme (the morpheme *fox*) while the word *cats* consists of two: the morpheme *cat* and the morpheme *-s*. MORPHEMES

As this example suggests, it is often useful to distinguish two broad classes of morphemes: **stems** and **affixes**. The exact details of the distinction vary from language to language, but intuitively, the stem is the "main" morpheme of the word, supplying the main meaning, while the affixes add "additional" meanings of various kinds. STEMS AFFIXES

Affixes are further divided into **prefixes**, **suffixes**, **infixes**, and **circumfixes**. Prefixes precede the stem, suffixes follow the stem, circumfixes do

both, and infixes are inserted inside the stem. For example, the word *eats* is composed of a stem *eat* and the suffix *-s*. The word *unbuckle* is composed of a stem *buckle* and the prefix *un-*. English doesn't have any good examples of circumfixes, but many other languages do. In German, for example, the past participle of some verbs formed by adding *ge-* to the beginning of the stem and *-t* to the end; so the past participle of the verb *sagen* (to say) is *gesagt* (said). Infixes, in which a morpheme is inserted in the middle of a word, occur very commonly for example in the Philipine language Tagalog. For example the affix *um*, which marks the agent of an action, is infixed to the Tagalog stem *hingi* "borrow" to produce *humingi*. There is one infix that occurs in some dialects of English in which taboo morpheme like "f**king" or "bl**dy" or others like it are inserted in the middle of other words ("Man-f**king-hattan", "abso-bl**dy-lutely"[1]) (McCawley, 1978).

Prefixes and suffixes are often called **concatenative morphology** since a word is composed of a number of morphemes concatenated together. A number of languages have extensive **non-concatenative morphology**, in which morphemes are combined in more complex ways. The Tagalog infixation example above is one example of non-concatenative morphology, since two morphemes (*hingi* and *um*) are intermingled. Another kind of non-concatenative morphology is called **templatic morphology** or **root-and-pattern** morphology. This is very common in Arabic, Hebrew, and other Semitic languages. In Hebrew, for example, a verb is constructed using two components: a root, consisting usually of three consonants (CCC) and carrying the basic meaning, and a template, which gives the ordering of consonants and vowels and specifies more semantic information about the resulting verb, such as the semantic voice (e.g., active, passive, middle). For example the Hebrew tri-consonantal root *lmd*, meaning 'learn' or 'study', can be combined with the active voice CaCaC template to produce the word *lamad*, 'he studied', or the intensive CiCeC template to produce the word *limed*, 'he taught', or the intensive passive template CuCaC to produce the word *lumad*, 'he was taught'.

A word can have more than one affix. For example, the word *rewrites* has the prefix *re-*, the stem *write*, and the suffix *-s*. The word *unbelievably* has a stem (*believe*) plus three affixes (*un-*, *-able*, and *-ly*). While English doesn't tend to stack more than four or five affixes, languages like Turkish can have words with nine or ten affixes, as we saw above. Languages

[1] Alan Jay Lerner, the lyricist of My Fair Lady, bowdlerized the latter to *abso-bloomin'lutely* in the lyric to "Wouldn't It Be Loverly?" (Lerner, 1978, p. 60).

that tend to string affixes together like Turkish does are called **agglutinative** languages.

There are two broad (and partially overlapping) classes of ways to form words from morphemes: **inflection** and **derivation**. Inflection is the combination of a word stem with a grammatical morpheme, usually resulting in a word of the same class as the original stem, and usually filling some syntactic function like agreement. For example, English has the inflectional morpheme *-s* for marking the **plural** on nouns, and the inflectional morpheme *-ed* for marking the past tense on verbs. Derivation is the combination of a word stem with a grammatical morpheme, usually resulting in a word of a *different* class, often with a meaning hard to predict exactly. For example the verb *computerize* can take the derivational suffix *-ation* to produce the noun *computerization*.

INFLECTION
DERIVATION

Inflectional Morphology

English has a relatively simple inflectional system; only nouns, verbs, and sometimes adjectives can be inflected, and the number of possible inflectional affixes is quite small.

English nouns have only two kinds of inflection: an affix that marks **plural** and an affix that marks **possessive**. For example, many (but not all) English nouns can either appear in the bare stem or **singular** form, or take a plural suffix. Here are examples of the regular plural suffix *-s*, the alternative spelling *-es*, and irregular plurals:

PLURAL
SINGULAR

	Regular Nouns		Irregular Nouns	
Singular	cat	thrush	mouse	ox
Plural	cats	thrushes	mice	oxen

While the regular plural is spelled *-s* after most nouns, it is spelled *-es* after words ending in *-s* (*ibis/ibises*) , *-z*, (*waltz/waltzes*) *-sh*, (*thrush/thrushes*) *-ch*, (*finch/finches*) and sometimes *-x* (*box/boxes*). Nouns ending in *-y* preceded by a consonant change the *-y* to *-i* (*butterfly/butterflies*).

The possessive suffix is realized by apostrophe + *-s* for regular singular nouns (*llama's*) and plural nouns not ending in *-s* (*children's*) and often by a lone apostrophe after regular plural nouns (*llamas'*) and some names ending in *-s* or *-z* (*Euripides' comedies*).

English verbal inflection is more complicated than nominal inflection. First, English has three kinds of verbs; **main verbs**, (*eat, sleep, impeach*), **modal verbs** (*can, will, should*), and **primary verbs** (*be, have, do*) (using

the terms of Quirk et al., 1985). In this chapter we will mostly be concerned with the main and primary verbs, because it is these that have inflectional
REGULAR endings. Of these verbs a large class are **regular**, that is to say all verbs of this class have the same endings marking the same functions. These regular verbs (e.g. *walk*, or *inspect*), have four morphological forms, as follow:

Morphological Form Classes	Regularly Inflected Verbs			
stem	walk	merge	try	map
-s form	walks	merges	tries	maps
-ing participle	walking	merging	trying	mapping
Past form or *-ed* participle	walked	merged	tried	mapped

These verbs are called regular because just by knowing the stem we can predict the other forms, by adding one of three predictable endings, and making some regular spelling changes (and as we will see in Chapter 4, regular pronunciation changes). These regular verbs and forms are significant in the morphology of English first because they cover a majority of the verbs, and second because the regular class is **productive**. As discussed earlier, a productive class is one that automatically includes any new words that enter the language. For example the recently-created verb *fax* (*My mom faxed me the note from cousin Everett*), takes the regular endings *-ed*, *-ing*, *-es*. (Note that the *-s* form is spelled *faxes* rather than *faxs*; we will discuss spelling rules below).

IRREGULAR VERBS The **irregular verbs** are those that have some more or less idiosyncratic forms of inflection. Irregular verbs in English often have five different forms, but can have as many as eight (e.g., the verb *be*) or as few as three (e.g. *cut* or *hit*). While constituting a much smaller class of verbs (Quirk et al. (1985) estimate there are only about 250 irregular verbs, not counting auxiliaries), this class includes most of the very frequent verbs of the language.[2] The table below shows some sample irregular forms. Note that an irregular
PRETERITE verb can inflect in the past form (also called the **preterite**) by changing its vowel (*eat/ate*), or its vowel and some consonants (*catch/caught*), or with no ending at all (*cut/cut*).

[2] In general, the more frequent a word form, the more likely it is to have idiosyncratic properties; this is due to a fact about language change; very frequent words preserve their form even if other words around them are changing so as to become more regular.

Morphological Form Classes	Irregularly Inflected Verbs		
stem	eat	catch	cut
-s form	eats	catches	cuts
-ing participle	eating	catching	cutting
Past form	ate	caught	cut
-ed participle	eaten	caught	cut

The way these forms are used in a sentence will be discussed in Chapters 8–12 but is worth a brief mention here. The *-s* form is used in the "habitual present" form to distinguish the third-person singular ending (*She jogs every Tuesday*) from the other choices of person and number (*I/you/we/they jog every Tuesday*). The stem form is used in the infinitive form, and also after certain other verbs (*I'd rather walk home, I want to walk home*). The *-ing* participle is used when the verb is treated as a noun; this particular kind of nominal use of a verb is called a **gerund** use: *Fishing is fine if you live near water.* The *-ed* participle is used in the **perfect** construction (*He's eaten lunch already*) or the passive construction (*The verdict was overturned yesterday.*).

GERUND

PERFECT

In addition to noting which suffixes can be attached to which stems, we need to capture the fact that a number of regular spelling changes occur at these morpheme boundaries. For example, a single consonant letter is doubled before adding the *-ing* and *-ed* suffixes (*beg/begging/begged*). If the final letter is "c", the doubling is spelled "ck" (*picnic/picnicking/picnicked*). If the base ends in a silent *-e*, it is deleted before adding *-ing* and *-ed* (*merge/merging/merged*). Just as for nouns, the *-s* ending is spelled *-es* after verb stems ending in *-s* (*toss/tosses*) , *-z*, (*waltz/waltzes*) *-sh*, (*wash/washes*) *-ch*, (*catch/catches*) and sometimes *-x* (*tax/taxes*). Also like nouns, verbs ending in *-y* preceded by a consonant change the *-y* to *-i* (*try/tries*).

The English verbal system is much simpler than for example the European Spanish system, which has as many as fifty distinct verb forms for each regular verb. Figure 3.1 shows just a few of the examples for the verb *amar*, 'to love'. Other languages can have even more forms than this Spanish example.

Derivational Morphology

While English inflection is relatively simple compared to other languages, derivation in English is quite complex. Recall that derivation is the combi-

Present Indicative	Imper.	Imperfect Indicative	Future	Preterite	Present Subjnct.	Conditional	Imperfect Subjnct.	Future Subjnct.
amo		amaba	amaré	amé	ame	amaría	amara	amare
amas	ama ames	amabas	amarás	amaste	ames	amarías	amaras	amares
ama		amaba	amará	amó	ame	amaría	amara	amáreme
amamos		amábamos	amaremos	amamos	amemos	amaríamos	amáramos	amáremos
amáis	amad amáis	amabais	amaréis	amasteis	améis	amaríais	amarais	amareis
aman		amaban	amarán	amaron	amen	amarían	amaran	amaren

Figure 3.1 To love in Spanish.

nation of a word stem with a grammatical morpheme, usually resulting in a word of a *different* class, often with a meaning hard to predict exactly.

NOMINALIZATION

A very common kind of derivation in English is the formation of new nouns, often from verbs or adjectives. This process is called **nominalization**. For example, the suffix *-ation* produces nouns from verbs ending often in the suffix *-ize* (*computerize* → *computerization*). Here are examples of some particularly productive English nominalizing suffixes.

Suffix	Base Verb/Adjective	Derived Noun
-ation	computerize (V)	computerization
-ee	appoint (V)	appointee
-er	kill (V)	killer
-ness	fuzzy (A)	fuzziness

Adjectives can also be derived from nouns and verbs. Here are examples of a few suffixes deriving adjectives from nouns or verbs.

Suffix	Base Noun/Verb	Derived Adjective
-al	computation (N)	computational
-able	embrace (V)	embraceable
-less	clue (N)	clueless

Derivation in English is more complex than inflection for a number of reasons. One is that it is generally less productive; even a nominalizing suffix like *-ation*, which can be added to almost any verb ending in *-ize*, cannot be added to absolutely every verb. Thus we can't say **eatation* or **spellation* (we use an asterisk (*) to mark "non-examples" of English). Another is that there are subtle and complex meaning differences among nominaliz-

ing suffixes. For example *sincerity* has a subtle difference in meaning from *sincereness*.

3.2 FINITE-STATE MORPHOLOGICAL PARSING

Let's now proceed to the problem of parsing English morphology. Consider a simple example: parsing just the productive nominal plural (*-s*) and the verbal progressive (*-ing*). Our goal will be to take input forms like those in the first column below and produce output forms like those in the second column.

Input	Morphological Parsed Output
cats	`cat +N +PL`
cat	`cat +N +SG`
cities	`city +N +PL`
geese	`goose +N +PL`
goose	(`goose +N +SG`) or (`goose +V`)
gooses	`goose +V +3SG`
merging	`merge +V +PRES-PART`
caught	(`catch +V +PAST-PART`) or (`catch +V +PAST`)

The second column contains the stem of each word as well as assorted morphological **features**. These features specify additional information about the stem. For example the feature `+N` means that the word is a noun; `+SG` means it is singular, `+PL` that it is plural. We will discuss features in Chapter 11; for now, consider `+SG` to be a primitive unit that means "singular". Note that some of the input forms (like *caught* or *goose*) will be ambiguous between different morphological parses. FEATURES

In order to build a morphological parser, we'll need at least the following:

1. **lexicon:** the list of stems and affixes, together with basic information about them (whether a stem is a Noun stem or a Verb stem, etc.). LEXICON
2. **morphotactics:** the model of morpheme ordering that explains which classes of morphemes can follow other classes of morphemes inside a word. For example, the rule that the English plural morpheme follows the noun rather than preceding it. MORPHOTACTICS
3. **orthographic rules:** these **spelling rules** are used to model the changes that occur in a word, usually when two morphemes combine (e.g., the

$y \rightarrow ie$ spelling rule discussed above that changes *city* + *-s* to *cities* rather than *citys*).

The next part of this section will discuss how to represent a simple version of the lexicon just for the sub-problem of morphological recognition, including how to use FSAs to model morphotactic knowledge. We will then introduce the finite-state transducer (FST) as a way of modeling morphological features in the lexicon, and addressing morphological parsing. Finally, we show how to use FSTs to model orthographic rules.

The Lexicon and Morphotactics

A lexicon is a repository for words. The simplest possible lexicon would consist of an explicit list of every word of the language (*every* word, i.e., including abbreviations ("AAA") and proper names ("Jane" or "Beijing") as follows:

a
AAA
AA
Aachen
aardvark
aardwolf
aba
abaca
aback
...

Since it will often be inconvenient or impossible, for the various reasons we discussed above, to list every word in the language, computational lexicons are usually structured with a list of each of the stems and affixes of the language together with a representation of the morphotactics that tells us how they can fit together. There are many ways to model morphotactics; one of the most common is the finite-state automaton. A very simple finite-state model for English nominal inflection might look like Figure 3.2.

The FSA in Figure 3.2 assumes that the lexicon includes regular nouns (**reg-noun**) that take the regular *-s* plural (e.g., *cat*, *dog*, *fox*, *aardvark*). These are the vast majority of English nouns since for now we will ignore the fact that the plural of words like *fox* have an inserted *e*: *foxes*. The lexicon also includes irregular noun forms that don't take *-s*, both singular **irreg-sg-noun** (*goose*, *mouse*) and plural **irreg-pl-noun** (*geese*, *mice*).

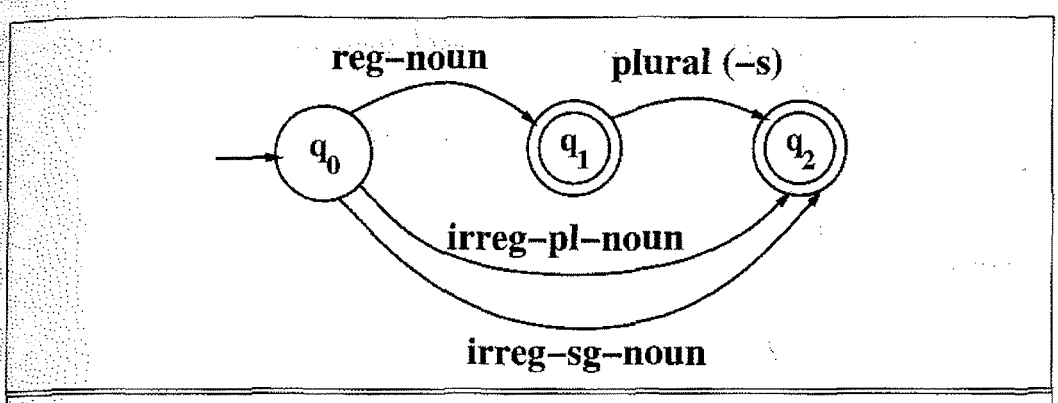


Figure 3.2 A finite-state automaton for English nominal inflection.

reg-noun	irreg-pl-noun	irreg-sg-noun	plural
fox	geese	goose	-s
cat	sheep	sheep	
dog	mice	mouse	
aardvark			

A similar model for English verbal inflection might look like Figure 3.3.

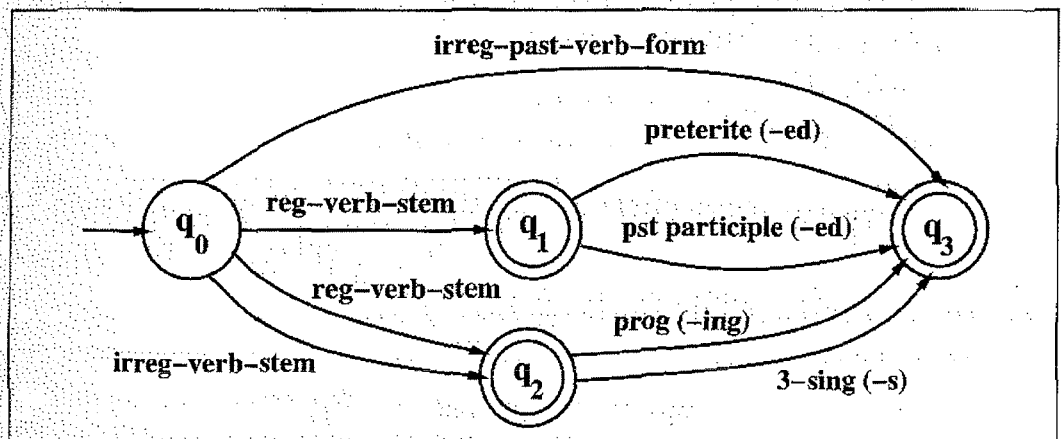


Figure 3.3 A finite-state automaton for English verbal inflection

This lexicon has three stem classes (reg-verb-stem, irreg-verb-stem, and irreg-past-verb-form), plus four more affix classes (*-ed* past, *-ed* participle, *-ing* participle, and third singular *-s*):

reg-verb-stem	irreg-verb-stem	irreg-past-verb	past	past-part	pres-part	3sg
walk	cut	caught	-ed	-ed	-ing	-s
fry	speak	ate				
talk	sing	eaten				
impeach	sang					
	spoken					

English derivational morphology is significantly more complex than English inflectional morphology, and so automata for modeling English derivation tend to be quite complex. Some models of English derivation, in fact, are based on the more complex context-free grammars of Chapter 9 (Sproat, 1993; Orgun, 1995).

As a preliminary example, though, of the kind of analysis it would require, we present a small part of the morphotactics of English adjectives, taken from Antworth (1990). Antworth offers the following data on English adjectives:

big, bigger, biggest
cool, cooler, coolest, coolly
red, redder, reddest
clear, clearer, clearest, clearly, unclear, unclearly
happy, happier, happiest, happily
unhappy, unhappier, unhappiest, unhappily
real, unreal, really

An initial hypothesis might be that adjectives can have an optional prefix (*un-*), an obligatory root (*big, cool,* etc) and an optional suffix (*-er, -est,* or *-ly*). This might suggest the the FSA in Figure 3.4.

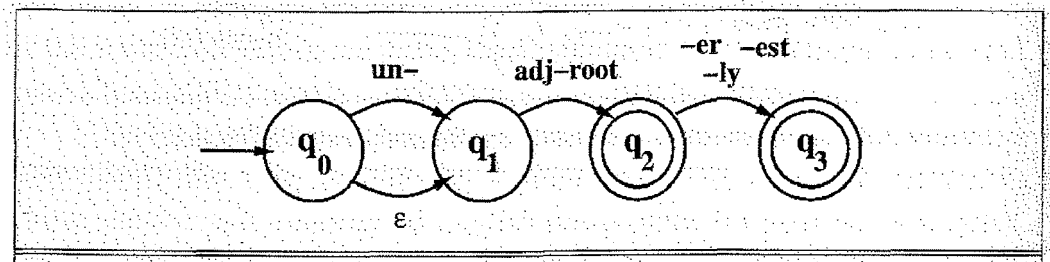


Figure 3.4 An FSA for a fragment of English adjective morphology: Antworth's Proposal #1.

Alas, while this FSA will recognize all the adjectives in the table above, it will also recognize ungrammatical forms like *unbig*, *redly*, and *realest*. We need to set up classes of roots and specify which can occur with which suffixes. So **adj-root**$_1$ would include adjectives that can occur with *un-* and *-ly* (*clear*, *happy*, and *real*) while **adj-root**$_2$ will include adjectives that can't (*big*, *cool*, and *red*). Antworth (1990) presents Figure 3.5 as a partial solution to these problems.

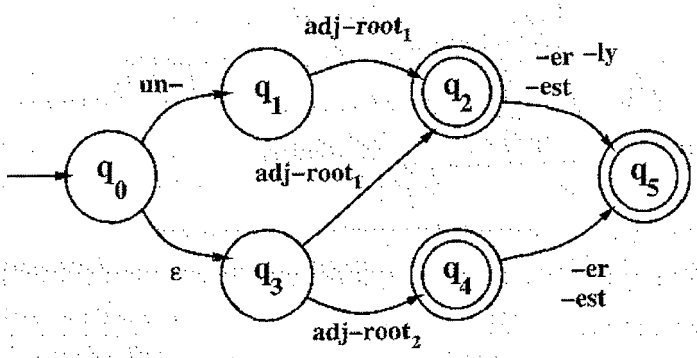


Figure 3.5 An FSA for a fragment of English adjective morphology: Antworth's Proposal #2.

This gives an idea of the complexity to be expected from English derivation. For a further example, we give in Figure 3.6 another fragment of an FSA for English nominal and verbal derivational morphology, based on Sproat (1993), Bauer (1983), and Porter (1980). This FSA models a number of derivational facts, such as the well known generalization that any verb ending in *-ize* can be followed by the nominalizing suffix *-ation* (Bauer, 1983; Sproat, 1993)). Thus since there is a word *fossilize*, we can predict the word *fossilization* by following states q_0, q_1, and q_2. Similarly, adjectives ending in *-al* or *-able* at q_5 (*equal*, *formal*, *realizable*) can take the suffix *-ity*, or sometimes the suffix *-ness* to state q_6 (*naturalness*, *casualness*). We leave it as an exercise for the reader (Exercise 3.2) to discover some of the individual exceptions to many of these constraints, and also to give examples of some of the various noun and verb classes.

We can now use these FSAs to solve the problem of **morphological recognition**; that is, of determining whether an input string of letters makes up a legitimate English word or not. We do this by taking the morphotactic FSAs, and plugging in each "sub-lexicon" into the FSA. That is, we expand each arc (e.g., the **reg-noun-stem** arc) with all the morphemes that make up the set of **reg-noun-stem**. The resulting FSA can then be defined at the level of the individual letter.

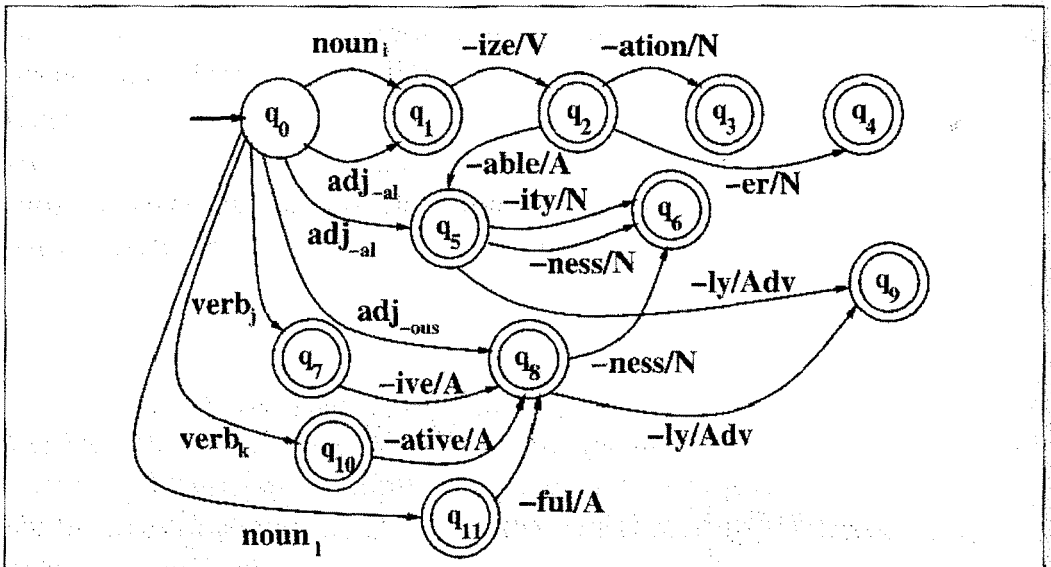


Figure 3.6 An FSA for another fragment of English derivational morphology.

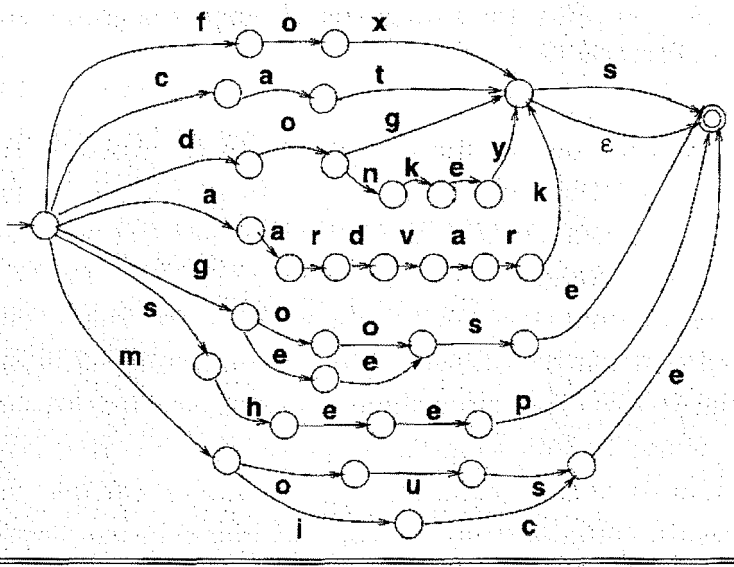


Figure 3.7 Compiled FSA for a few English nouns with their inflection. Note that this automaton will incorrectly accept the input *foxs*. We will see beginning on page 76 how to correctly deal with the inserted *e* in *foxes*.

Figure 3.7 shows the noun-recognition FSA produced by expanding the Nominal Inflection FSA of Figure 3.2 with sample regular and irregular nouns for each class. We can use Figure 3.7 to recognize strings like *aardvarks* by simply starting at the initial state, and comparing the input letter

by letter with each word on each outgoing arc, and so on, just as we saw in Chapter 2.

Morphological Parsing with Finite-State Transducers

Now that we've seen how to use FSAs to represent the lexicon and incidentally do morphological recognition, let's move on to morphological parsing. For example, given the input *cats*, we'd like to output `cat +N +PL`, telling us that cat is a plural noun. We will do this via a version of **two-level morphology**, first proposed by Koskenniemi (1983). Two-level morphology represents a word as a correspondence between a **lexical level**, which represents a simple concatenation of morphemes making up a word, and the **surface** level, which represents the actual spelling of the final word. Morphological parsing is implemented by building mapping rules that map letter sequences like *cats* on the surface level into morpheme and features sequences like `cat +N +PL` on the lexical level. Figure 3.8 shows these two levels for the word *cats*. Note that the lexical level has the stem for a word, followed by the morphological information `+N +PL` which tells us that *cats* is a plural noun.

TWO-LEVEL

SURFACE

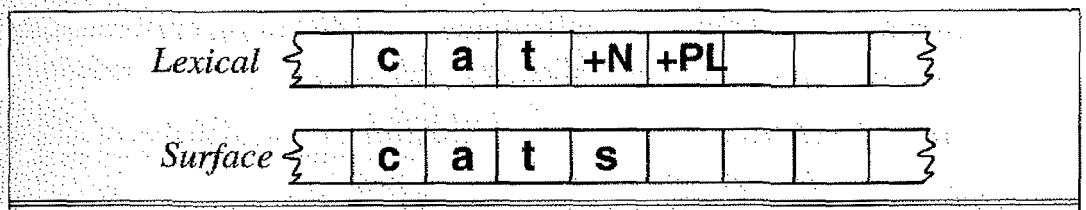


Figure 3.8 Example of the lexical and surface tapes.

The automaton that we use for performing the mapping between these two levels is the **finite-state transducer** or **FST**. A transducer maps between one set of symbols and another; a finite-state transducer does this via a finite automaton. Thus we usually visualize an FST as a two-tape automaton which recognizes or generates *pairs* of strings. The FST thus has a more general function than an FSA; where an FSA defines a formal language by defining a set of strings, an FST defines a *relation* between sets of strings. This relates to another view of an FST; as a machine that reads one string and generates another. Here's a summary of this four-fold way of thinking about transducers:

FST

- **FST as recognizer:** a transducer that takes a pair of strings as input and outputs *accept* if the string-pair is in the string-pair language, and a *reject* if it is not.
- **FST as generator:** a machine that outputs pairs of strings of the language. Thus the output is a yes or no, and a pair of output strings.
- **FST as translator:** a machine that reads a string and outputs another string
- **FST as set relater:** a machine that computes relations between sets.

MEALY MACHINE

An FST can be formally defined in a number of ways; we will rely on the following definition, based on what is called the **Mealy machine** extension to a simple FSA:

- Q: a finite set of N states $q_0, q_1, \ldots, q_N$
- Σ: a finite alphabet of complex symbols. Each complex symbol is composed of an input-output pair $i:o$; one symbol i from an input alphabet I, and one symbol o from an output alphabet O, thus $\Sigma \subseteq I \times O$. I and O may each also include the epsilon symbol ε.
- q_0: the start state
- F: the set of final states, $F \subseteq Q$
- $\delta(q, i:o)$: the transition function or transition matrix between states. Given a state $q \in Q$ and complex symbol $i:o \in \Sigma$, $\delta(q, i:o)$ returns a new state $q' \in Q$. δ is thus a relation from $Q \times \Sigma$ to Q.

Where an FSA accepts a language stated over a finite alphabet of single symbols, such as the alphabet of our sheep language:

$$\Sigma = \{b, a, !\} \tag{3.2}$$

an FST accepts a language stated over *pairs* of symbols, as in:

$$\Sigma = \{a:a,\ b:b,\ !:!,\ a:!,\ a:\varepsilon,\ \varepsilon:!\} \tag{3.3}$$

FEASIBLE PAIRS

In two-level morphology, the pairs of symbols in Σ are also called **feasible pairs**.

REGULAR RELATIONS

Where FSAs are isomorphic to regular languages, FSTs are isomorphic to **regular relations**. Regular relations are sets of pairs of strings, a natural extension of the regular languages, which are sets of strings. Like FSAs and regular languages, FSTs and regular relations are closed under union, although in general they are not closed under difference, complementation and intersection (although some useful subclasses of FSTs *are* closed under these operations; in general FSTs that are not augmented with the ε

are more likely to have such closure properties). Besides union, FSTs have two additional closure properties that turn out to be extremely useful:

- **inversion**: The inversion of a transducer T (T^{-1}) simply switches the input and output labels. Thus if T maps from the input alphabet I to the output alphabet O, T^{-1} maps from O to I. INVERSION
- **composition**: If T_1 is a transducer from I_1 to O_1 and T_2 a transducer from I_2 to O_2, then $T_1 \circ T_2$ maps from I_1 to O_2. COMPOSITION

Inversion is useful because it makes it easy to convert a FST-as-parser into an FST-as-generator. Composition is useful because it allows us to take two transducers that run in series and replace them with one more complex transducer. Composition works as in algebra; applying $T_1 \circ T_2$ to an input sequence S is identical to applying T_1 to S and then T_2 to the result; thus $T_1 \circ T_2(S) = T_2(T_1(S))$. We will see examples of composition below.

We mentioned that for two-level morphology it's convenient to view an FST as having two tapes. The **upper** or **lexical tape**, is composed from characters from the left side of the $a:b$ pairs; the **lower** or **surface** tape, is composed of characters from the right side of the $a:b$ pairs. Thus each symbol $a:b$ in the transducer alphabet Σ expresses how the symbol a from one tape is mapped to the symbol b on the another tape. For example $a:\varepsilon$ means that an a on the upper tape will correspond to *nothing* on the lower tape. Just as for an FSA, we can write regular expressions in the complex alphabet Σ. Since it's most common for symbols to map to themselves, in two-level morphology we call pairs like $a:a$ **default pairs**, and just refer to them by the single letter a. LEXICAL TAPE DEFAULT PAIRS

We are now ready to build an FST morphological parser out of our earlier morphotactic FSAs and lexica by adding an extra "lexical" tape and the appropriate morphological features. Figure 3.9 shows an augmentation of Figure 3.2 with the nominal morphological features (+SG and +PL) that correspond to each morpheme. Note that these features map to the empty string ε or the word/morpheme boundary symbol # since there is no segment corresponding to them on the output tape.

In order to use Figure 3.9 as a morphological noun parser, it needs to be augmented with all the individual regular and irregular noun stems, replacing the labels **regular-noun-stem** etc. In order to do this we need to update the lexicon for this transducer, so that irregular plurals like *geese* will parse into the correct stem goose +N +PL. We do this by allowing the lexicon to also have two levels. Since surface *geese* maps to underlying goose, the new lexical entry will be "g:g o:e o:e s:s e:e". Regular forms are

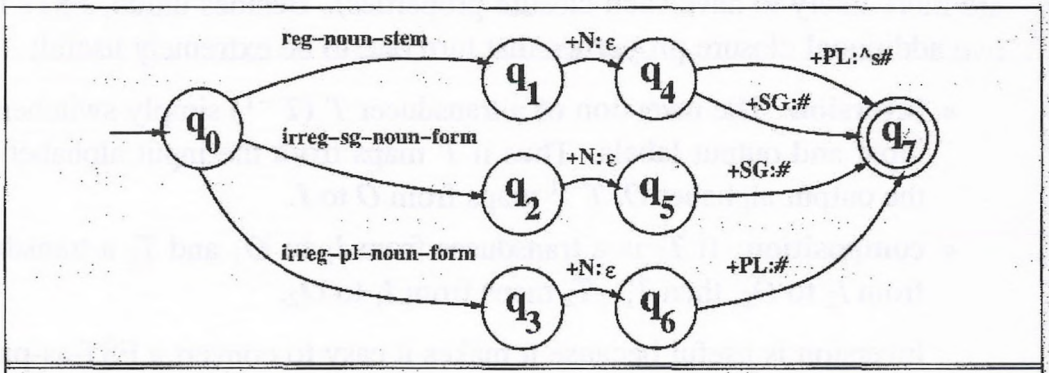


Figure 3.9 A transducer for English nominal number inflection T_{num}. Since both q_1 and q_2 are accepting states, regular nouns can have the plural suffix or not. The morpheme-boundary symbol ^ and word-boundary marker # will be discussed below.

simpler; the two-level entry for *fox* will now be "`f:f o:o x:x`", but by relying on the orthographic convention that `f` stands for `f:f` and so on, we can simply refer to it as `fox` and the form for *geese* as "`g o:e o:e s e`". Thus the lexicon will look only slightly more complex:

reg-noun	irreg-pl-noun	irreg-sg-noun
fox	g o:e o:e s e	goose
cat	sheep	sheep
dog	m o:i u:ɛ s:c e	mouse
aardvark		

Our proposed morphological parser needs to map from surface forms like *geese* to lexical forms like `goose +N +SG`. We could do this by **cascading** the lexicon above with the singular/plural automaton of Figure 3.9. Cascading two automata means running them in series with the output of the first feeding the input to the second. We would first represent the lexicon of stems in the above table as the FST T_{stems} of Figure 3.10. This FST maps e.g. *dog* to **reg-noun-stem**. In order to allow possible suffixes, T_{stems} in Figure 3.10 allows the forms to be followed by the wildcard @ **symbol**; `@:@` stands for "any feasible pair". A pair of the form `@:x`, for example will mean "any feasible pair which has `x` on the surface level", and correspondingly for the form `x:@`. The output of this FST would then feed the number automaton T_{num}.

@ SYMBOL

Instead of cascading the two transducers, we can **compose** them using the composition operator defined above. Composing is a way of taking a

cascade of transducers with many different levels of inputs and outputs and converting them into a single "two-level" transducer with one input tape and one output tape. The algorithm for composition bears some resemblance to the algorithm for determinization of FSAs from page 48; given two automata T_1 and T_2 with state sets Q_1 and Q_2 and transition functions δ_1 and δ_2, we create a new possible state (x,y) for every pair of states $x \in Q_1$ and $y \in Q_2$. Then the new automaton has the transition function:

$$\begin{aligned}\delta_3((x_a,y_a),i:o) &= (x_b,y_b) \text{ if} \\ &\exists\, c \text{ s.t. } \delta_1(x_a,i:c) = x_b \\ &\text{and } \delta_2(y_a,c:o) = y_b \end{aligned} \tag{3.4}$$

The resulting composed automaton, $T_{lex} = T_{num} \circ T_{stems}$, is shown in Figure 3.11 (compare this with the FSA lexicon in Figure 3.7 on page 70).[3] Note that the final automaton still has two levels separated by the :. Because the colon was reserved for these levels, we had to use the | symbol in T_{stems} in Figure 3.10 to separate the upper and lower tapes.

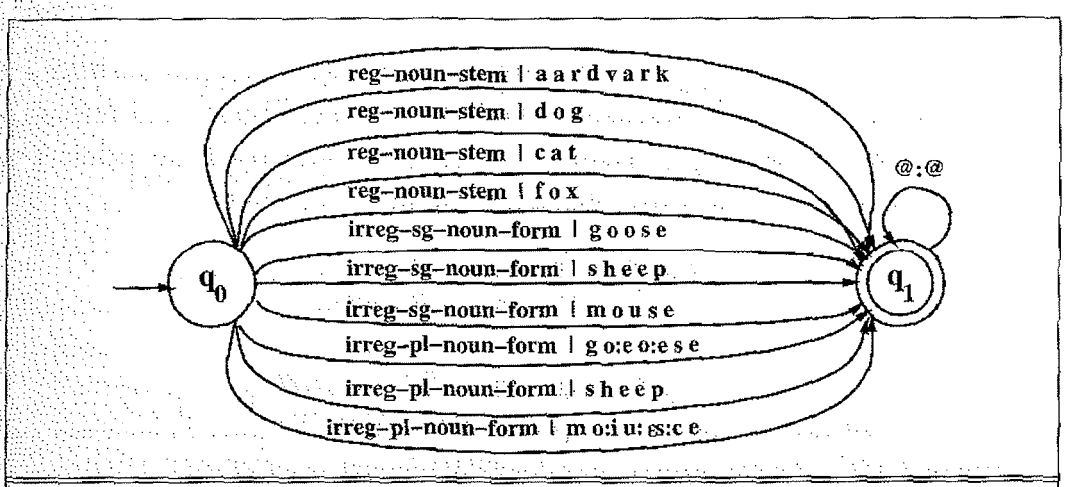


Figure 3.10 The transducer T_{stems}, which maps roots to their root-class.

This transducer will map plural nouns into the stem plus the morphological marker +PL, and singular nouns into the stem plus the morpheme +SG. Thus a surface *cats* will map to cat +N +PL as follows:

c:c a:a t:t +N:ε +PL:^s#

That is, c maps to itself, as do a and t, while the morphological feature +N (recall that this means "noun") maps to nothing (ε), and the feature +PL

[3] Note that for the purposes of clear exposition, Figure 3.11 has not been minimized in the way that Figure 3.7 has.

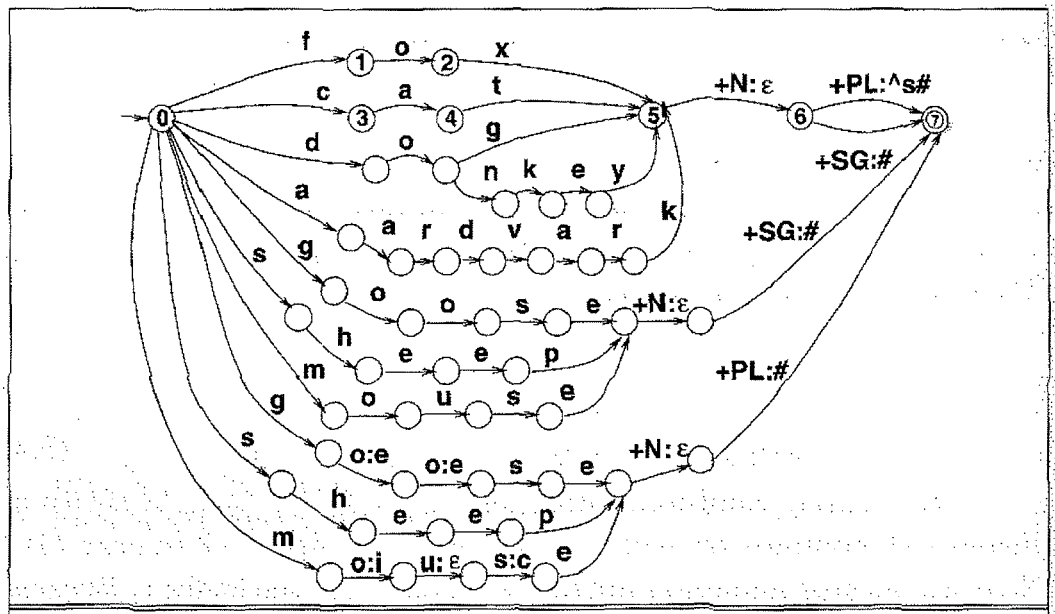


Figure 3.11 A fleshed-out English nominal inflection FST $T_{lex} = T_{num} \circ T_{stems}$.

(meaning "plural") maps to ^s. The symbol ^ indicates a **morpheme boundary**, while the symbol **#** indicates a **word boundary**, Figure 3.12 refers to tapes with these morpheme boundary markers as **intermediate** tapes; the next section will show how the boundary marker is removed.

MORPHEME BOUNDARY

#

WORD BOUNDARY

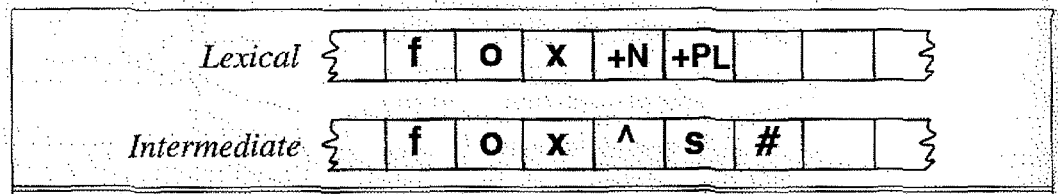


Figure 3.12 An example of the lexical and intermediate tapes.

Orthographic Rules and Finite-State Transducers

The method described in the previous section will successfully recognize words like *aardvarks* and *mice*. But just concatenating the morphemes won't work for cases where there is a spelling change; it would incorrectly reject an input like *foxes* and accept an input like *foxs*. We need to deal with the fact that English often requires spelling changes at morpheme boundaries by introducing **spelling rules** (or **orthographic rules**). This section introduces a number of notations for writing such rules and shows how to implement the rules as transducers. Some of these spelling rules:

SPELLING RULES

Name	Description of Rule	Example
Consonant doubling	1-letter consonant doubled before *-ing/-ed*	beg/begging
E deletion	Silent e dropped before *-ing* and *-ed*	make/making
E insertion	e added after *-s,-z,-x,-ch, -sh* before *-s*	watch/watches
Y replacement	*-y* changes to *-ie* before *-s*, *-i* before *-ed*	try/tries
K insertion	verbs ending with *vowel + -c* add *-k*	panic/panicked

We can think of these spelling changes as taking as input a simple concatenation of morphemes (the "intermediate output" of the lexical transducer in Figure 3.11) and producing as output a slightly-modified, (correctly-spelled) concatenation of morphemes. Figure 3.13 shows the three levels we are talking about: lexical, intermediate, and surface. So for example we could write an E-insertion rule that performs the mapping from the intermediate to surface levels shown in Figure 3.13. Such a rule might say some-

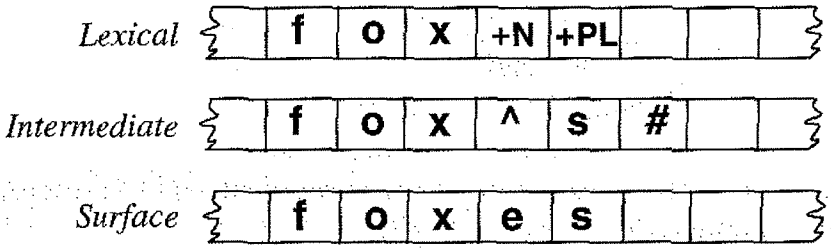


Figure 3.13 An example of the lexical, intermediate, and surface tapes. Between each pair of tapes is a two-level transducer; the lexical transducer of Figure 3.11 between the lexical and intermediate levels, and the E-insertion spelling rule between the intermediate and surface levels. The E-insertion spelling rule inserts an *e* on the surface tape when the intermediate tape has a morpheme boundary ^ followed by the morpheme *-s*.

thing like "insert an *e* on the surface tape just when the lexical tape has a morpheme ending in *x* (or *z*, etc) and the next morpheme is *-s*". Here's a formalization of the rule:

$$\varepsilon \rightarrow \text{e} / \left\{ \begin{matrix} x \\ s \\ z \end{matrix} \right\} \hat{} \_\_ \text{s\#} \tag{3.5}$$

This is the rule notation of Chomsky and Halle (1968); a rule of the form $a \rightarrow b / c \_\_ d$ means "rewrite a as b when it occurs between c and

d". Since the symbol ε means an empty transition, replacing it means inserting something. The symbol ^ indicates a morpheme boundary. These boundaries are deleted by including the symbol ^:ε in the default pairs for the transducer; thus morpheme boundary markers are deleted on the surface level by default. (Recall that the colon is used to separate symbols on the intermediate and surface forms). The # symbol is a special symbol that marks a word boundary. Thus (3.5) means "insert an *e* after a morpheme-final *x*, *s*, or *z*, and before the morpheme *s*". Figure 3.14 shows an automaton that corresponds to this rule.

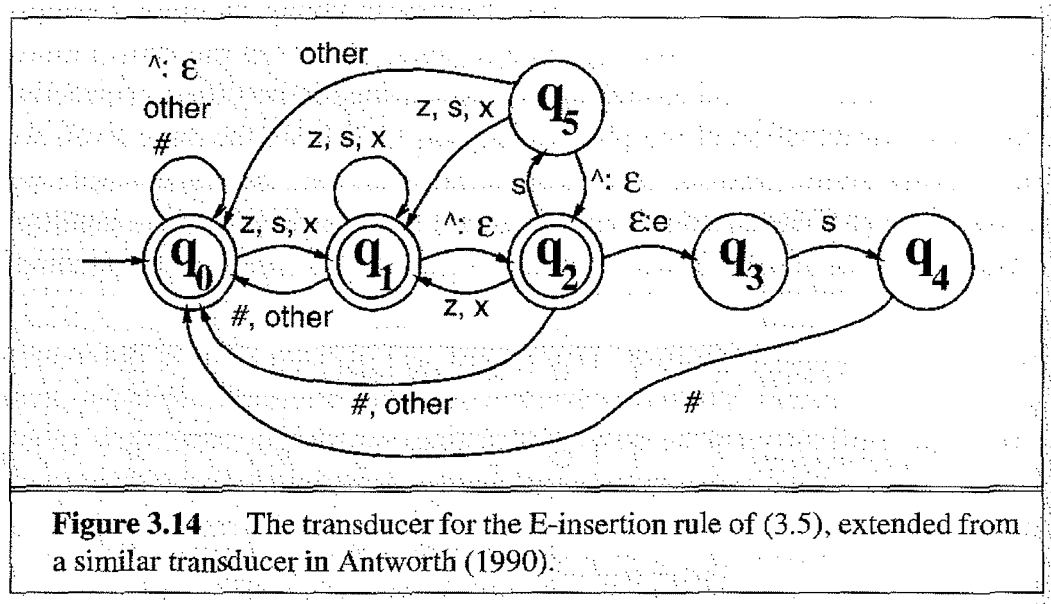


Figure 3.14 The transducer for the E-insertion rule of (3.5), extended from a similar transducer in Antworth (1990).

The idea in building a transducer for a particular rule is to express only the constraints necessary for that rule, allowing any other string of symbols to pass through unchanged. This rule is used to insure that we can only see the ε:*e* pair if we are in the proper context. So state q_0, which models having seen only default pairs unrelated to the rule, is an accepting state, as is q_1, which models having seen a *z*, *s*, or *x*. q_2 models having seen the morpheme boundary after the *z*, *s*, or *x*, and again is an accepting state. State q_3 models having just seen the E-insertion; it is not an accepting state, since the insertion is only allowed if it is followed by the *s* morpheme and then the end-of-word symbol #.

The *other* symbol is used in Figure 3.14 to safely pass through any parts of words that don't play a role in the E-insertion rule. *other* means "any feasible pair that is not in this transducer"; it is thus a version of *@:@* which is context-dependent in a transducer-by-transducer way. So for example when leaving state q_0, we go to q_1 on the *z*, *s*, or *x* symbols, rather than

following the *other* arc and staying in q_0. The semantics of *other* depends on what symbols are on other arcs; since # is mentioned on some arcs, it is (by definition) not included in *other*, and thus, for example, is explicitly mentioned on the arc from q_2 to q_0.

A transducer needs to correctly reject a string that applies the rule when it shouldn't. One possible bad string would have the correct environment for the E-insertion, but have no insertion. State q_5 is used to insure that the *e* is always inserted whenever the environment is appropriate; the transducer reaches q_5 only when it has seen an *s* after an appropriate morpheme boundary. If the machine is in state q_5 and the next symbol is #, the machine rejects the string (because there is no legal transition on # from q_5). Figure 3.15 shows the transition table for the rule which makes the illegal transitions explicit with the "–" symbol.

State\Input	s:s	x:x	z:z	^:ε	ε:e	#	other
q_0:	1	1	1	0	-	0	0
q_1:	1	1	1	2	-	0	0
q_2:	5	1	1	0	3	0	0
q_3	4	-	-	-	-	-	-
q_4	-	-	-	-	-	0	-
q_5	1	1	1	2	-	-	0

Figure 3.15 The state-transition table for E-insertion rule of Figure 3.14, extended from a similar transducer in Antworth (1990).

The next section will show a trace of this E-insertion transducer running on a sample input string.

3.3 Combining FST Lexicon and Rules

We are now ready to combine our lexicon and rule transducers for parsing and generating. Figure 3.16 shows the architecture of a two-level morphology system, whether used for parsing or generating. The lexicon transducer maps between the lexical level, with its stems and morphological features, and an intermediate level that represents a simple concatenation of morphemes. Then a host of transducers, each representing a single spelling rule constraint, all run in parallel so as to map between this intermediate level and the surface level. Putting all the spelling rules in parallel is a design choice;

we could also have chosen to run all the spelling rules in series (as a long cascade), if we slightly changed each rule.

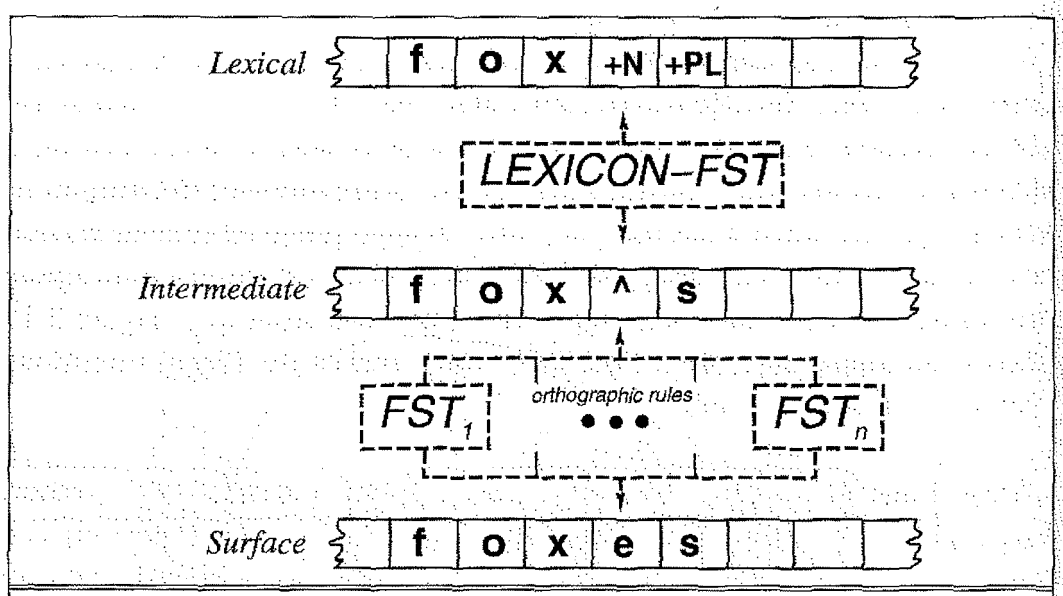


Figure 3.16 Generating or parsing with FST lexicon and rules

The architecture in Figure 3.16 is a two-level cascade of transducers. Recall that a cascade is a set of transducers in series, in which the output from one transducer acts as the input to another transducer; cascades can be of arbitrary depth, and each level might be built out of many individual transducers. The cascade in Figure 3.16 has two transducers in series: the transducer mapping from the lexical to the intermediate levels, and the collection of parallel transducers mapping from the intermediate to the surface level. The cascade can be run top-down to generate a string, or bottom-up to parse it; Figure 3.17 shows a trace of the system *accepting* the mapping from *fox^s* to *foxes*.

The power of finite-state transducers is that the exact same cascade with the same state sequences is used when the machine is generating the surface tape from the lexical tape, or when it is parsing the lexical tape from the surface tape. For example, for generation, imagine leaving the Intermediate and Surface tapes blank. Now if we run the lexicon transducer, given `fox +N +PL`, it will produce *fox^s#* on the Intermediate tape via the same states that it accepted the Lexical and Intermediate tapes in our earlier example. If we then allow all possible orthographic transducers to run in parallel, we will produce the same surface tape.

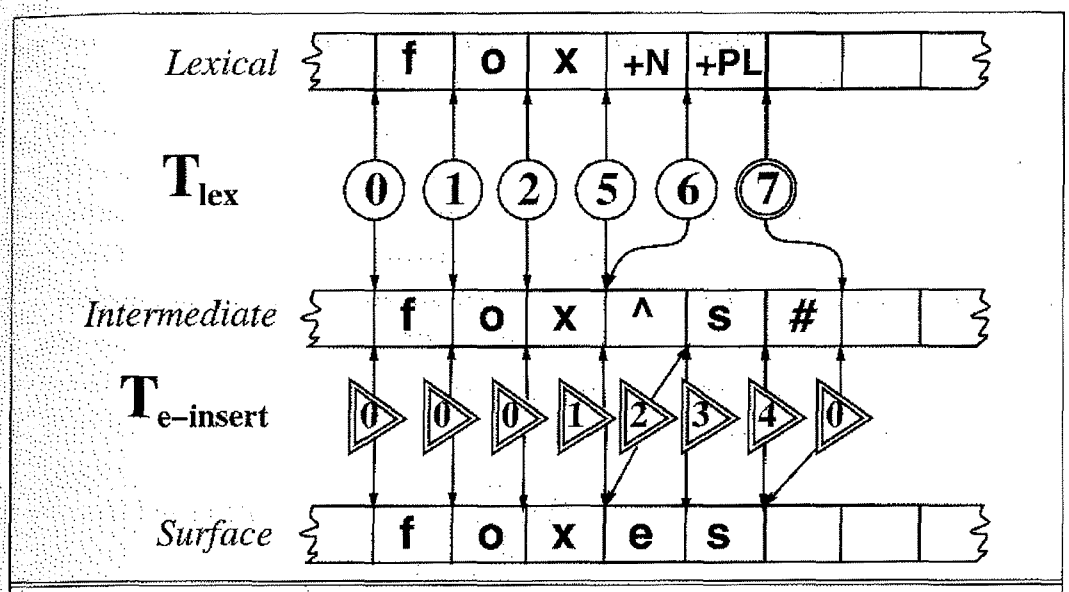


Figure 3.17 Accepting *foxes*: The lexicon transducer T_{lex} from Figure 3.11 cascaded with the E-insertion transducer in Figure 3.14.

Parsing can be slightly more complicated than generation, because of the problem of **ambiguity**. For example, *foxes* can also be a verb (albeit a rare one, meaning "to baffle or confuse"), and hence the lexical parse for *foxes* could be `fox +V +3SG` as well as `fox +N +PL`. How are we to know which one is the proper parse? In fact, for ambiguous cases of this sort, the transducer is not capable of deciding. **Disambiguating** will require some external evidence such as the surrounding words. Thus *foxes* is likely to be a noun in the sequence *I saw two foxes yesterday*, but a verb in the sequence *That trickster foxes me every time!*. We will discuss such disambiguation algorithms in Chapters 8 and 17. Barring such external evidence, the best our transducer can do is just enumerate the possible choices; so we can transduce *fox^s#* into both `fox +V +3SG` and `fox +N +PL`.

AMBIGUITY

DISAMBIGUATING

There is a kind of ambiguity that we need to handle: local ambiguity that occurs during the process of parsing. For example, imagine parsing the input verb *assess*. After seeing *ass*, our E-insertion transducer may propose that the *e* that follows is inserted by the spelling rule (for example, as far as the transducer is concerned, we might have been parsing the word *asses*). It is not until we don't see the *#* after *asses*, but rather run into another *s*, that we realize we have gone down an incorrect path.

Because of this non-determinism, FST-parsing algorithms need to incorporate some sort of search algorithm. Exercise 3.8 asks the reader to modify the algorithm for non-deterministic FSA recognition in Figure 2.21 in Chapter 2 to do FST parsing.

Running a cascade, particularly one with many levels, can be unwieldy. Luckily, we've already seen how to compose a cascade of transducers in series into a single more complex transducer. Transducers in parallel can be combined by **automaton intersection**. The automaton intersection algorithm just takes the Cartesian product of the states, i.e., for each state q_i in machine 1 and state q_j in machine 2, we create a new state q_{ij}. Then for any input symbol a, if machine 1 would transition to state q_n and machine 2 would transition to state q_m, we transition to state q_{nm}.

INTERSECTION

Figure 3.18 sketches how this intersection ($\wedge$) and composition ($\circ$) process might be carried out.

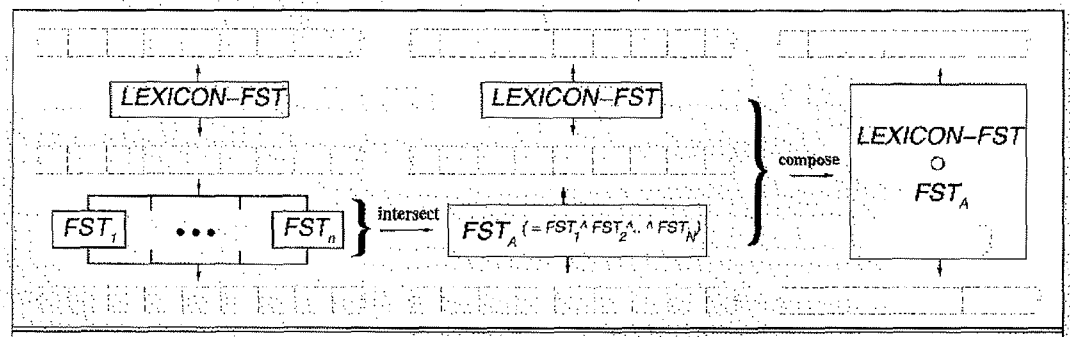


Figure 3.18 Intersection and composition of transducers.

Since there are a number of rule→FST compilers, it is almost never necessary in practice to write an FST by hand. Kaplan and Kay (1994) give the mathematics that define the mapping from rules to two-level relations, and Antworth (1990) gives details of the algorithms for rule compilation. Mohri (1997) gives algorithms for transducer minimization and determinization.

3.4 Lexicon-Free FSTs: The Porter Stemmer

While building a transducer from a lexicon plus rules is the standard algorithm for morphological parsing, there are simpler algorithms that don't require the large on-line lexicon demanded by this algorithm. These are used especially in Information Retrieval (IR) tasks (Chapter 17) in which a user needs some information, and is looking for relevant documents (perhaps on the web, perhaps in a digital library database). She gives the system a query with some important characteristics of documents she desires, and the IR system retrieves what it thinks are the relevant documents. One common

type of query is Boolean combinations of relevant **keywords** or phrases, e.g. (*marsupial OR kangaroo OR koala*). The system then returns documents that have these words in them. Since a document with the word *marsupials* might not match the keyword *marsupial*, some IR systems first run a stemmer on the keywords and on the words in the document. Since morphological parsing in IR is only used to help form equivalence classes, the details of the suffixes are irrelevant; what matters is determining that two words have the same stem. KEYWORDS

One of the most widely used such **stemming** algorithms is the simple and efficient Porter (1980) algorithm, which is based on a series of simple cascaded rewrite rules. Since cascaded rewrite rules are just the sort of thing that could be easily implemented as an FST, we think of the Porter algorithm as a lexicon-free FST stemmer (this idea will be developed further in the exercises (Exercise 3.7). The algorithm contains rules like: STEMMING

(3.6) ATIONAL → ATE (e.g., relational → relate)

(3.7) ING → ε if stem contains vowel (e.g., motoring → motor)

The algorithm is presented in detail in Appendix B.

Do stemmers really improve the performance of information retrieval engines? One problem is that stemmers are not perfect. For example Krovetz (1993) summarizes the following kinds of errors of omission and of commission in the Porter algorithm:

Errors of Commission		Errors of Omission	
organization	organ	European	Europe
doing	doe	analysis	analyzes
generalization	generic	matrices	matrix
numerical	numerous	noise	noisy
policy	police	sparse	sparsity
university	universe	explain	explanation
negligible	negligent	urgency	urgent

Krovetz also gives the results of a number of experiments testing whether the Porter stemmer actually improved IR performance. Overall he found some improvement, especially with smaller documents (the larger the document, the higher the chance the keyword will occur in the exact form used in the query). Since any improvement is quite small, IR engines often don't use stemming.

3.5 HUMAN MORPHOLOGICAL PROCESSING

In this section we look at psychological studies to learn how multi-morphemic words are represented in the minds of speakers of English. For example, consider the word *walk* and its inflected forms *walks*, and *walked*. Are all three in the human lexicon? Or merely *walk* plus as well as *-ed* and *-s*? How about the word *happy* and its derived forms *happily* and *happiness*? We can imagine two ends of a theoretical spectrum of representations. The **full listing** hypothesis proposes that all words of a language are listed in the mental lexicon without any internal morphological structure. On this view, morphological structure is simply an epiphenomenon, and *walk*, *walks*, *walked*, *happy*, and *happily* are all separately listed in the lexicon. This hypothesis is certainly untenable for morphologically complex languages like Turkish (Hankamer (1989) estimates Turkish as 200 billion possible words). The **minimum redundancy** hypothesis suggests that only the constituent morphemes are represented in the lexicon, and when processing *walks*, (whether for reading, listening, or talking) we must access both morphemes (*walk* and *-s*) and combine them.

FULL LISTING

MINIMUM REDUNDANCY

Most modern experimental evidence suggests that neither of these is completely true. Rather, some kinds of morphological relationships are mentally represented (particularly inflection and certain kinds of derivation), but others are not, with those words being fully listed. Stanners et al. (1979), for example, found that derived forms (*happiness*, *happily*) are stored separately from their stem (*happy*), but that regularly inflected forms (*pouring*) are not distinct in the lexicon from their stems (*pour*). They did this by using a repetition priming experiment. In short, repetition priming takes advantage of the fact that a word is recognized faster if it has been seen before (if it is **primed**). They found that *lifting* primed *lift*, and *burned* primed *burn*, but for example *selective* didn't prime *select*. Figure 3.19 sketches one possible representation of their finding:

PRIMED

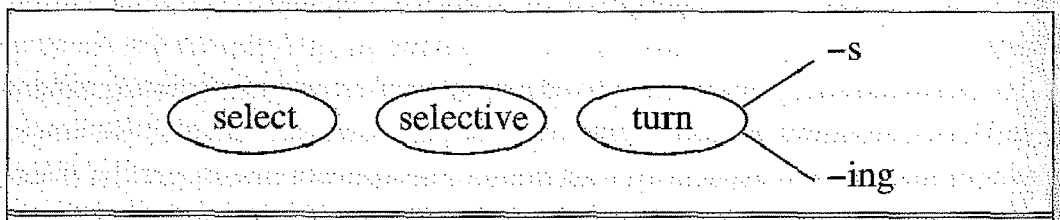


Figure 3.19 Stanners et al. (1979) result: Different representations of inflection and derivation.

In a more recent study, Marslen-Wilson et al. (1994) found that *spoken* derived words can prime their stems, but only if the meaning of the derived form is closely related to the stem. For example *government* primes *govern*, but *department* does not prime *depart*. Grainger et al. (1991) found similar results with prefixed words (but not with suffixed words). Marslen-Wilson et al. (1994) represent a model compatible with their own findings as follows:

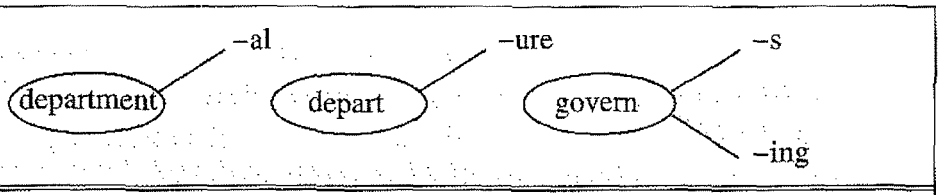


Figure 3.20 Marslen-Wilson et al. (1994) result: Derived words are linked to their stems only if semantically related

Other evidence that the human lexicon represents some morphological structure comes from **speech errors**, also called **slips of the tongue**. In normal conversation, speakers often mix up the order of the words or initial sounds:

if you break it it'll drop
I don't have time to work to watch television because I have to work

But inflectional and derivational affixes can also appear separately from their stems, as these examples from Fromkin and Ratner (1998) and Garrett (1975) show:

it's not only us who have screw looses (for "screws loose")
words of rule formation (for "rules of word formation")
easy enoughly (for "easily enough")
which by itself is the most unimplausible sentence you can imagine

The ability of these affixes to be produced separately from their stem suggests that the mental lexicon must contain some representation of the morphological structure of these words.

In summary, these results suggest that morphology does play a role in the human lexicon, especially productive morphology like inflection. They also emphasize the important of semantic generalizations across words, and suggest that the human auditory lexicon (representing words in terms of their sounds) and the orthographic lexicon (representing words in terms of letters)

may have similar structures. Finally, it seems that many properties of language processing, like morphology, may apply equally (or at least similarly) to language **comprehension** and language **production**.

3.6 SUMMARY

This chapter introduced **morphology**, the arena of language processing dealing with the subparts of words, and the **finite-state transducer**, the computational device that is commonly used to model morphology. Here's a summary of the main points we covered about these ideas:

- **Morphological parsing** is the process of finding the constituent **morphemes** in a word (e.g., `cat +N +PL` for *cats*).
- English mainly uses **prefixes** and **suffixes** to express **inflectional** and **derivational** morphology.
- English **inflectional** morphology is relatively simple and includes person and number agreement (*-s*) and tense markings (*-ed* and *-ing*).
- English **derivational** morphology is more complex and includes suffixes like *-ation*, *-ness*, *-able* as well as prefixes like *co-* and *re-*.
- Many constraints on the English **morphotactics** (allowable morpheme sequences) can be represented by finite automata.
- **Finite-state transducers** are an extension of finite-state automata that can generate output symbols.
- **Two-level morphology** is the application of finite-state transducers to morphological representation and parsing.
- **Spelling rules** can be implemented as transducers.
- There are automatic transducer-compilers that can produce a transducer for any simple rewrite rule.
- The lexicon and spelling rules can be combined by **composing** and **intersecting** various transducers.
- The **Porter algorithm** is a simple and efficient way to do **stemming**, stripping off affixes. It is not as accurate as a transducer model that includes a lexicon, but may be preferable for applications like **information retrieval** in which exact morphological structure is not needed.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

Despite the close mathematical similarity of finite-state transducers to finite-state automata, the two models grew out of somewhat different traditions. Chapter 2 described how the finite automaton grew out of Turing's (1936) model of algorithmic computation, and McCulloch and Pitts finite-state-like models of the neuron. The influence of the Turing machine on the transducer was somewhat more indirect. Huffman (1954) proposed what was essentially a state-transition table to model the behavior of sequential circuits, based on the work of Shannon (1938) on an algebraic model of relay circuits. Based on Turing and Shannon's work, and unaware of Huffman's work, Moore (1956) introduced the term **finite automaton** for a machine with a finite number of states with an alphabet of input symbols and an alphabet of output symbols. Mealy (1955) extended and synthesized the work of Moore and Huffman.

The finite automata in Moore's original paper, and the extension by Mealy differed in an important way. In a Mealy machine, the input/output symbols are associated with the transitions between states. The finite-state transducers in this chapter are Mealy machines. In a Moore machine, the input/output symbols are associated with the state; we will see examples of Moore machines in Chapter 5 and Chapter 7. The two types of transducers are equivalent; any Moore machine can be converted into an equivalent Mealy machine and vice versa.

Many early programs for morphological parsing used an **affix-stripping** approach to parsing. For example Packard's (1973) parser for ancient Greek iteratively stripped prefixes and suffixes off the input word, making note of them, and then looked up the remainder in a lexicon. It returned any root that was compatible with the stripped-off affixes. This approach is equivalent to the **bottom-up** method of parsing that we will discuss in Chapter 10.

AMPLE (A Morphological Parser for Linguistic Exploration) (Weber and Mann, 1981; Weber et al., 1988; Hankamer and Black, 1991) is another early bottom-up morphological parser. It contains a lexicon with all possible surface variants of each morpheme (these are called **allomorphs**), together with constraints on their occurrence (for example in English the *-es* allomorph of the plural morpheme can only occur after s, x, z, sh, or ch). The system finds every possible sequence of morphemes which match the input and then filters out all the sequences which have failing constraints.

An alternative approach to morphological parsing is called *generate-and-test* or *analysis-by-synthesis* approach. Hankamer's (1986) keCi is a morphological parser for Turkish which is guided by a finite-state representation of Turkish morphemes. The program begins with a morpheme that might match the left edge of the word, and applies every possible phonological rule to it, checking each result against the input. If one of the outputs succeeds, the program then follows the finite-state morphotactics to the next morpheme and tries to continue matching the input.

The idea of modeling spelling rules as finite-state transducers is really based on Johnson's (1972) early idea that phonological rules (to be discussed in Chapter 4) have finite-state properties. Johnson's insight unfortunately did not attract the attention of the community, and was independently discovered by Roland Kaplan and Martin Kay, first in an unpublished talk (Kaplan and Kay, 1981) and then finally in print (Kaplan and Kay, 1994) (see page 15 for a discussion of multiple independent discoveries). Kaplan and Kay's work was followed up and most fully worked out by Koskenniemi (1983), who described finite-state morphological rules for Finnish. Karttunen (1983) built a program called KIMMO based on Koskenniemi's models. Antworth (1990) gives many details of two-level morphology and its application to English. Besides Koskenniemi's work on Finnish and that of Antworth (1990) on English, two-level or other finite-state models of morphology have been worked out for many languages, such as Turkish (Oflazer, 1993) and Arabic (Beesley, 1996). Antworth (1990) summarizes a number of issues in finite-state analysis of languages with morphologically complex processes like infixation and reduplication (e.g., Tagalog) and gemination (e.g., Hebrew). Karttunen (1993) is a good summary of the application of two-level morphology specifically to phonological rules of the sort we will discuss in Chapter 4. Barton et al. (1987) bring up some computational complexity problems with two-level models, which are responded to by Koskenniemi and Church (1988).

Students interested in further details of the fundamental mathematics of automata theory should see Hopcroft and Ullman (1979) or Lewis and Papadimitriou (1981). Mohri (1997) and Roche and Schabes (1997b) give additional algorithms and mathematical foundations for language applications, including, for example, the details of the algorithm for transducer minimization. Sproat (1993) gives a broad general introduction to computational morphology.

EXERCISES

3.1 Add some adjectives to the adjective FSA in Figure 3.5.

3.2 Give examples of each of the noun and verb classes in Figure 3.6, and find some exceptions to the rules.

3.3 Extend the transducer in Figure 3.14 to deal with `sh` and `ch`.

3.4 Write a transducer(s) for the K insertion spelling rule in English.

3.5 Write a transducer(s) for the consonant doubling spelling rule in English.

3.6 The Soundex algorithm (Odell and Russell, 1922; Knuth, 1973) is a method commonly used in libraries and older Census records for representing people's names. It has the advantage that versions of the names that are slightly misspelled or otherwise modified (common, for example, in handwritten census records) will still have the same representation as correctly-spelled names. (e.g., Jurafsky, Jarofsky, Jarovsky, and Jarovski all map to J612).

a. Keep the first letter of the name, and drop all occurrences of non-initial a, e, h, i, o, u, w, y

b. Replace the remaining letters with the following numbers:

b, f, p, v $\rightarrow$ 1
c, g, j, k, q, s, x, z $\rightarrow$ 2
d, t $\rightarrow$ 3
l $\rightarrow$ 4
m, n $\rightarrow$ 5
r $\rightarrow$ 6

c. Replace any sequences of identical numbers with a single number (i.e., 666 $\rightarrow$ 6)

d. Convert to the form `Letter Digit Digit Digit` by dropping digits past the third (if necessary) or padding with trailing zeros (if necessary).

The exercise: write a FST to implement the Soundex algorithm.

3.7 Implement one of the steps of the Porter Stemmer as a transducer.

3.8 Write the algorithm for parsing a finite-state transducer, using the pseudo-code introduced in Chapter 2. You should do this by modifying the algorithm ND-RECOGNIZE in Figure 2.21 in Chapter 2.

3.9 Write a program that takes a word and, using an on-line dictionary, computes possible anagrams of the word, each of which is a legal word.

3.10 In Figure 3.14, why is there a *z, s, x* arc from q_5 to q_1?

4 COMPUTATIONAL PHONOLOGY AND TEXT-TO-SPEECH

You like po-tay-to and I like po-tah-to.
You like to-may-to and I like to-mah-to.
Po-tay-to, po-tah-to,
To-may-to, to-mah-to,
Let's call the whole thing off!

George and Ira Gershwin, *Let's Call the Whole Thing Off* from *Shall We Dance*, 1937

The debate between the "whole language" and "phonics" methods of teaching reading to children seems at very glance like a purely modern educational debate. Like many modern debates, however, this one recapitulates an important historical dialectic, in this case in writing systems. The earliest independently-invented writing systems (Sumerian, Chinese, Mayan) were mainly logographic: one symbol represented a whole word. But from the earliest stages we can find, most such systems contain elements of syllabic or phonemic writing systems, in which symbols are used to represent the sounds that make up the words. Thus the Sumerian symbol pronounced *ba* and meaning "ration" could also function purely as the sound /ba/. Even modern Chinese, which remains primarily logographic, uses sound-based characters to spell out foreign words and especially geographical names. Purely sound-based writing systems, whether syllabic (like Japanese *hiragana* or *katakana*), alphabetic (like the Roman alphabet used in this book), or consonantal (like Semitic writing systems), can generally be traced back to these early logo-syllabic systems, often as two cultures came together. Thus the Arabic, Aramaic, Hebrew, Greek, and Roman systems all derive from a West Semitic script that is presumed to have been modified by Western Semitic mercenaries from a cursive form of Egyptian hieroglyphs. The

Japanese syllabaries were modified from a cursive form of a set of Chinese characters which were used to represent sounds. These Chinese characters themselves were used in Chinese to phonetically represent the Sanskrit in the Buddhist scriptures that were brought to China in the Tang dynasty.

Whatever its origins, the idea implicit in a sound-based writing system, that the spoken word is composed of smaller units of speech, is the Ur-theory that underlies all our modern theories of phonology. In the next four chapters we begin our exploration of these ideas, as we introduce the fundamental insights and algorithms necessary to understand modern speech recognition and speech synthesis technology, and the related branch of linguistics called **computational phonology**.

Let's begin by defining these areas. The core task of automatic speech recognition is take an acoustic waveform as input and produce as output a string of words. Conversely, the core task of text-to-speech synthesis is to take a sequence of text words and produce as output an acoustic waveform. The uses of speech recognition and synthesis are manifold, including automatic dictation/transcription, speech-based interfaces to computers and telephones, voice-based input and output for the disabled, and many others that will be discussed in greater detail in Chapter 7.

This chapter will focus on an important part of both speech recognition and text-to-speech systems: how words are pronounced in terms of individual speech units called **phones**. A speech recognition system needs to have a pronunciation for every word it can recognize, and a text-to-speech system needs to have a pronunciation for every word it can say. The first section of this chapter will introduce **phonetic alphabets** for describing pronunciation,

PHONETICS

ARTICULATORY PHONETICS

part of the field of **phonetics**. We then introduce **articulatory phonetics**, the study of how speech sounds are produced by articulators in the mouth.

Modeling pronunciation would be much simpler if a given phone was always pronounced the same in every context. Unfortunately this is not the case. As we will see, the phone [t] is pronounced very differently in different phonetic environments. **Phonology** is the area of linguistics that describes the systematic way that sounds are differently realized in different environments, and how this system of sounds is related to the rest of the grammar. The next section of the chapter will describe the way we write **phonological rules** to describe these different realizations.

COMPUTATIONAL PHONOLOGY

We next introduce an area known as **computational phonology**. One important part of computational phonology is the study of computational mechanisms for modeling phonological rules. We will show how the spelling-rule transducers of Chapter 3 can be used to model phonology. We then