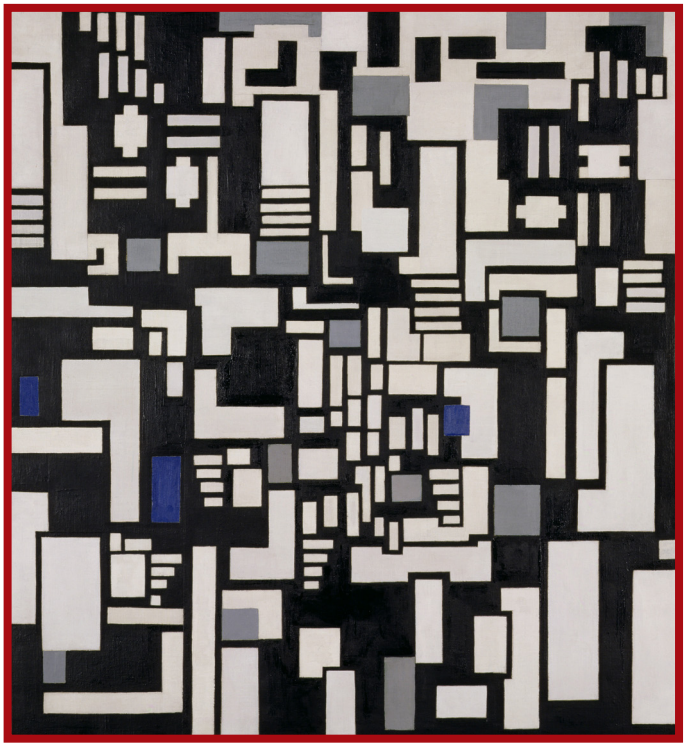
The Handbook of Computational Linguistics and Natural Language Processing



Edited by

Alexander Clark, Chris Fox, and Shalom Lappin

WILEY-BLACKWELL

Praise for *The Handbook of Computational Linguistics and Natural Language Processing*

"All in all, this is very well compiled book, which effectively balances the width and depth of theories and applications in two very diverse yet closely related fields of language research."

Machine Translation

"This *Handbook* is exceptionally broad and exceptionally deep in its coverage. The contributions, by noted experts, cover all aspects of the field, from fundamental theory to concrete applications. Clark, Fox and Lappin have performed a great service by compiling this volume."

Richard Sproat, Oregon Health & Science University

Blackwell Handbooks in Linguistics

This outstanding multi-volume series covers all the major subdisciplines within linguistics today and, when complete, will offer a comprehensive survey of linguistics as a whole.

Already published:

The Handbook of Child Language
Edited by Paul Fletcher and Brian MacWhinney

The Handbook of Phonological Theory, Second Edition
Edited by John A. Goldsmith, Jason Riggle, and Alan C. L. Yu

The Handbook of Contemporary Semantic Theory
Edited by Shalom Lappin

The Handbook of Sociolinguistics
Edited by Florian Coulmas

The Handbook of Phonetic Sciences, Second Edition
Edited by William J. Hardcastle and John Laver

The Handbook of Morphology
Edited by Andrew Spencer and Arnold Zwicky

The Handbook of Japanese Linguistics
Edited by Natsuko Tsujimura

The Handbook of Linguistics
Edited by Mark Aronoff and Janie Rees-Miller

The Handbook of Contemporary Syntactic Theory
Edited by Mark Baltin and Chris Collins

The Handbook of Discourse Analysis
Edited by Deborah Schiffrin, Deborah Tannen, and Heidi E. Hamilton

The Handbook of Language Variation and Change
Edited by J. K. Chambers, Peter Trudgill, and Natalie Schilling-Estes

The Handbook of Historical Linguistics
Edited by Brian D. Joseph and Richard D. Janda

The Handbook of Language and Gender
Edited by Janet Holmes and Miriam Meyerhoff

The Handbook of Second Language Acquisition
Edited by Catherine J. Doughty and Michael H. Long

The Handbook of Bilingualism and Multilingualism, Second Edition
Edited by Tej K. Bhatia and William C. Ritchie

The Handbook of Pragmatics
Edited by Laurence R. Horn and Gregory Ward

The Handbook of Applied Linguistics
Edited by Alan Davies and Catherine Elder

The Handbook of Speech Perception
Edited by David B. Pisoni and Robert E. Remez

The Handbook of the History of English
Edited by Ans van Kemenade and Bettelou Los

The Handbook of English Linguistics
Edited by Bas Aarts and April McMahon

The Handbook of World Englishes
Edited by Braj B. Kachru; Yamuna Kachru, and Cecil L. Nelson

The Handbook of Educational Linguistics
Edited by Bernard Spolsky and Francis M. Hult

The Handbook of Clinical Linguistics
Edited by Martin J. Ball, Michael R. Perkins, Nicole Müller, and Sara Howard

The Handbook of Pidgin and Creole Studies
Edited by Silvia Kouwenberg and John Victor Singler

The Handbook of Language Teaching
Edited by Michael H. Long and Catherine J. Doughty

The Handbook of Language Contact
Edited by Raymond Hickey

The Handbook of Language and Speech Disorders
Edited by Jack S. Damico, Nicole Müller, Martin J. Ball

The Handbook of Computational Linguistics and Natural Language Processing
Edited by Alexander Clark, Chris Fox, and Shalom Lappin

The Handbook of Language and Globalization
Edited by Nikolas Coupland

The Handbook of Hispanic Linguistics
Edited by Manuel Díaz-Campos

The Handbook of Language Socialization
Edited by Alessandro Duranti, Elinor Ochs, and Bambi B. Schieffelin

The Handbook of Intercultural Discourse and Communication
Edited by Christina Bratt Paulston, Scott F. Kiesling, and Elizabeth S. Rangel

The Handbook of Historical Sociolinguistics
Edited by Juan Manuel Hernández-Campoy and Juan Camilo Conde-Silvestre

The Handbook of Hispanic Linguistics
Edited by José Ignacio Hualde, Antxon Olarrea, and Erin O'Rourke

The Handbook of Conversation Analysis
Edited by Jack Sidnell and Tanya Stivers

The Handbook of English for Specific Purposes
Edited by Brian Paltridge and Sue Starfield

The Handbook of Computational Linguistics and Natural Language Processing

Edited by

Alexander Clark, Chris Fox, and Shalom Lappin

WILEY-BLACKWELL
A John Wiley & Sons, Ltd., Publication

This paperback edition first published 2013

Edition History: Blackwell Publishing Ltd (hardback, 2010)

Blackwell Publishing was acquired by John Wiley & Sons in February 2007. Blackwell's publishing program has been merged with Wiley's global Scientific, Technical, and Medical business to form Wiley-Blackwell.

Registered Office
John Wiley & Sons Ltd, The Atrium, Southern Gate, Chichester, West Sussex, PO19 8SQ, UK

Editorial Offices
350 Main Street, Malden, MA 02148-5020, USA
9600 Garsington Road, Oxford, OX4 2DQ, UK
The Atrium, Southern Gate, Chichester, West Sussex, PO19 8SQ, UK

For details of our global editorial offices, for customer services, and for information about how to apply for permission to reuse the copyright material in this book please see our website at www.wiley.com/wiley-blackwell.

Library of Congress Cataloging-in-Publication Data

The handbook of computational linguistics and natural language processing / edited by Alexander Clark, Chris Fox, and Shalom Lappin.
p. cm. – (Blackwell handbooks in linguistics)
Includes bibliographical references and index.
ISBN 978-1-4051-5581-6 (hardcover : alk. paper) ISBN 978-1-118-34718-8 (paperback : alk. paper)
1. Computational linguistics. 2. Natural language processing (Computer science).
I. Clark, Alexander (Alexander Simon) II. Fox, Chris, 1965– III. Lappin, Shalom.
P98.H346 2010
410_.285–dc22
2010003116

A catalogue record for this book is available from the British Library.

Cover image: Theo van Doesburg, Composition IX, opus 18, 1917. Haags Gemeentemuseum, The Hague, Netherlands / The Bridgeman Art Library.
Cover design by Workhaus.

Set in 10/12pt Palatino by SPi Publisher Services, Pondicherry, India

1 2013

For Camilla

לאחיי דוד ודניאל, ולאחותי נעמי באהבה ובהומור

Contents

List of Figures

List of Tables

Notes on Contributors

Ciprian Chelba is a Research Scientist with Google. Between 2000 and 2006 he worked as a Researcher in the Speech Technology Group at Microsoft Research.

He received his Diploma Engineer degree in 1993 from the Faculty of Electronics and Telecommunications at "Politehnica" University, Bucuresti, Romania, M.S. in 1996 and PhD in 2000 from the Electrical and Computer Engineering Department at the Johns Hopkins University.

His research interests are in statistical modeling of natural language and speech, as well as related areas such as machine learning and information theory as applied to natural language problems.

Recent projects include language modeling for large-vocabulary speech recognition (discriminative model estimation, compact storage for large models), search in spoken document collections (spoken content indexing, ranking and snipeting), as well as speech and text classification.

Alexander Clark is an Honorary Research Fellow in the Department of Computer Science at Royal Holloway, University of London. His first degree was in Mathematics from the University of Cambridge, and his PhD is from the University of Sussex. He did postdoctoral research at the University of Geneva. In 2007 he was a *Professeur invité* at the University of Marseille. He is on the editorial board of the journal *Research on Language and Computation*, and a member of the steering committee of the International Colloquium on Grammatical Inference. His research is on unsupervised learning in computational linguistics, and in grammatical inference; he has won several prizes and competitions for his research. He has co-authored with Shalom Lappin a book entitled *Linguistic Nativism and the Poverty of the Stimulus*, which is being published by Wiley-Blackwell in 2010.

Stephen Clark is a Senior Lecturer at the University of Cambridge Computer Laboratory where he is a member of the Natural Language and Information Processing Research Group. From 2004 to 2008 he was a University Lecturer at the Oxford University Computing Laboratory, and before that spent four years as a postdoctoral researcher at the University of Edinburgh's School of Informatics,

working with Prof. Mark Steedman. He has a PhD in Artificial Intelligence from the University of Sussex and a first degree in Philosophy from the University of Cambridge. His main research interest is statistical parsing, with a focus on the grammar formalism combinatory categorial grammar. In 2009 he led a team at the Johns Hopkins University Summer Workshop working on "Large Scale Syntactic Processing: Parsing the Web." He is on the editorial boards of *Computational Linguistics* and the *Journal of Natural Language Engineering*, and is a Program Co-Chair for the 2010 Annual Meeting of the Association for Computational Linguistics.

Matthew W. Crocker obtained his PhD in Artificial Intelligence from the University of Edinburgh in 1992, where he subsequently held appointments as Lecturer in Artificial Intelligence and Cognitive Science and as an ESRC Research Fellow. In January 2000, Dr Crocker was appointed to a newly established Chair in Psycholinguistics, in the Department of Computational Linguistics at Saarland University, Germany. His current research brings together the experimental investigation of real-time human language processing and situated cognition in the development of computational cognitive models.

Matthew Crocker co-founded the annual conference on Architectures and Mechanisms for Language Processing (AMLaP) in 1995. He is currently an associate editor for *Cognition*, on the editorial board of Springer's *Studies in Theoretical Psycholinguistics*, and has been a member of the editorial board for *Computational Linguistics*.

Walter Daelemans (MA, University of Leuven, Belgium, 1982; PhD, Computational Linguistics, University of Leuven, 1987) held research and teaching positions at the Radboud University Nijmegen, the AI-LAB at the University of Brussels, and Tilburg University, where he founded the ILK (Induction of Linguistic Knowledge) research group, and where he remained part-time Full Professor until 2006. Since 1999, he has been a Full Professor at the University of Antwerp (UA), teaching Computational Linguistics and Artificial Intelligence courses and co-directing the CLiPS research center. His current research interests are in machine learning of natural language, computational psycholinguistics, and text mining. He was elected fellow of ECCAI in 2003 and graduated 11 PhD students as supervisor.

Raquel Fernández is a Postdoctoral Researcher at the Institute for Logic, Language and Computation, University of Amsterdam. She holds a PhD in Computer Science from King's College London for work on formal and computational modeling of dialogue and has published numerous peer-review articles on dialogue research. She has worked as Research Fellow in the Center for the Study of Language and Information (CSLI) at Stanford University and in the Linguistics Department at the University of Potsdam.

Dr **Chris Fox** is a Reader in the School of Computer Science and Electronic Engineering at the University of Essex. He started his research career as a Senior Research Officer in the Department of Language and Linguistics at the University of Essex. He subsequently worked in the Computer Science Department where he

obtained his PhD in 1993. After that he spent a brief period as a Visiting Researcher at Saarbruecken before becoming a Lecturer at Goldsmiths College, University of London, and then King's College London. He returned to Essex in 2003. At the time of writing, he is serving as Deputy Mayor of Wivenhoe.

Much of his research is in the area of logic and formal semantics, with a particular emphasis on issues of formal expressiveness, and proof-theoretic approaches to characterizing intuitions about natural language semantic phenomena.

Jonathan Ginzburg is a Senior Lecturer in the Department of Computer Science at King's College London. He has previously held posts in Edinburgh and Jerusalem. He is one of the managing editors of the journal *Dialogue and Discourse*. He has published widely on formal semantics and dialogue. His monograph *The Interactive Stance: Meaning for Conversation* was published in 2009.

John A. Goldsmith is Edward Carson Waller Distinguished Service Professor in the Departments of Linguistics and Computer Science at the University of Chicago, where he has been since 1984. He received his PhD in Linguistics in 1976 from MIT, and taught from 1976 to 1984 at Indiana University. His primary interests are computational learning of natural language, phonological theory, and the history of linguistics.

Ralph Grishman is Professor of Computer Science at New York University. He has been involved in research in natural language processing since 1969, and since 1985 has directed the Proteus Project, with funding from DARPA, NSF, and other government agencies. The Proteus Project has conducted research in natural language text analysis, with a focus on information extraction, and has been involved in the creation of a number of major lexical and syntactic resources, including Comlex, Nomlex, and NomBank. He is a past President of the Association for Computational Linguistics and the author of the text *Computational Linguistics: An Introduction*.

Thomas Hain holds the degree Dipl.-Ing. with honors from the University of Technology, Vienna and a PhD from Cambridge University. In 1994 he joined Philips Speech Processing, which he left as Senior Technologist in 1997. He took up a position as Research Associate at the Speech, Vision and Robotics Group and Machine Intelligence Lab at the Cambridge University Engineering Department where he also received an appointment as Lecturer in 2001. In 2004 he joined the Department of Computer Science at the University of Sheffield where he is now a Senior Lecturer. Thomas Hain has a well established track record in automatic speech recognition, in particular involvement in best-performing ASR systems for participation in NIST evaluations. His main research interests are in speech recognition, speech and audio processing, machine learning, optimisation of large-scale statistical systems, and modeling of machine/machine interfaces. He is a member of the IEEE Speech and Language Technical Committee.

James B. Henderson is an MER (Research Professor) in the Department of Computer Science of the University of Geneva, where he is co-head of the interdisciplinary research group Computational Learning and Computational

Linguistics. His research bridges the topics of machine learning methods for structure-prediction tasks and the modeling and exploitation of such tasks in NLP, particularly syntactic and semantic parsing. In machine learning his current interests focus on latent variable models inspired by neural networks. Previously, Dr Henderson was a Research Fellow in ICCS at the University of Edinburgh, and a Lecturer in CS at the University of Exeter, UK. Dr Henderson received his PhD and MSc from the University of Pennsylvania, and his BSc from the Massachusetts Institute of Technology, USA.

Shalom Lappin is Professor of Computational Linguistics at King's College London. He does research in computational semantics, and in the application of machine learning to issues in natural language processing and the cognitive basis of language acquisition. He has taught at SOAS, Tel Aviv University, the University of Haifa, the University of Ottawa, and Ben Gurion University of the Negev. He was also a Research Staff member in the Natural Language group of the Computer Science Department at IBM T.J. Watson Research Center. He edited the *Handbook of Contemporary Semantic Theory* (1996, Blackwell), and, with Chris Fox, he co-authored *Foundations of Intensional Semantics* (2005, Blackwell). His most recent book, *Linguistic Nativism and the Poverty of the Stimulus*, co-authored with Alexander Clark, is being published by Wiley-Blackwell in 2010.

Jimmy Lin is an Associate Professor in the iSchool at the University of Maryland, affiliated with the Department of Computer Science and the Institute for Advanced Computer Studies. He graduated with a PhD in Computer Science from MIT in 2004. Lin's research lies at the intersection of information retrieval and natural language processing, and he has done work in a variety of areas, including question answering, medical informatics, bioinformatics, evaluation metrics, and knowledge-based retrieval techniques. Lin's current research focuses on "cloud computing," in particular, massively distributed text processing in cluster-based environments.

Robert Malouf is an Associate Professor in the Department of Linguistics and Asian/Middle Eastern Languages at San Diego State University. Before coming to SDSU, Robert held a postdoctoral fellowship in the Humanities Computing Department, University of Groningen (1999–2002). He received a PhD in Linguistics from Stanford University (1998) and BA in linguistics and computer science from SUNY Buffalo (1992). His research focuses on the application of computational techniques to understanding how language works, particularly in the domains of morphology and syntax. He is currently investigating the use of evolutionary simulation for explaining linguistic universals.

Prof. **Ruslan Mitkov** has been working in (applied) natural language processing, computational linguistics, corpus linguistics, machine translation, translation technology, and related areas since the early 1980s. His extensively cited research covers areas such as anaphora resolution, automatic generation of

multiple-choice tests, machine translation, natural language generation, automatic summarization, computer-aided language processing, centering, translation memory, evaluation, corpus annotation, bilingual term extraction, question answering, automatic identification of cognates and false friends, and an NLP-driven corpus-based study of translation universals.

Mitkov is author of the monograph *Anaphora Resolution* (2002, Longman) and sole editor of *The Oxford Handbook of Computational Linguistics* (2005, Oxford University Press). Current prestigious projects include his role as Executive Editor of the *Journal of Natural Language Engineering* (Cambridge University Press) and Editor-in-Chief of the *Natural Language Processing* book series (John Benjamins Publishing). Ruslan Mitkov received his MSc from the Humboldt University in Berlin, his PhD from the Technical University in Dresden and he worked as a Research Professor at the Institute of Mathematics, Bulgarian Academy of Sciences, Sofia. Prof. Mitkov is Professor of Computational Linguistics and Language Engineering at the School of Humanities, Languages and Social Sciences at the University of Wolverhampton which he joined in 1995, where he set up the Research Group in Computational Linguistics. In addition to being Head of the Research Group in Computational Linguistics, Prof. Mitkov is also Director of the Research Institute in Information and Language Processing.

Dr **Mark-Jan Nederhof** is a Lecturer in the School of Computer Science at the University of St Andrews. He holds a PhD (1994) and MSc (1990) in computer science from the University of Nijmegen. Before coming to St Andrews in 2006, he was Senior Researcher at DFKI in Saarbrücken and Lecturer in the Faculty of Arts at the University of Groningen. He has served on the editorial board of *Computational Linguistics* and has been a member of the programme committees of EACL, HLT/EMNLP, and COLING-ACL.

His research covers areas of computational linguistics and computer languages, with an emphasis on formal language theory and computational complexity. He is also developing tools for use in philological research, and especially the study of Ancient Egyptian.

Martha Palmer is an Associate Professor in the Linguistics Department and the Computer Science Department of the University of Colorado at Boulder, as well as a Faculty Fellow of the Institute of Cognitive Science. She was formerly an Associate Professor in Computer and Information Sciences at the University of Pennsylvania. She has been actively involved in research in natural language processing and knowledge representation for 30 years and did her PhD in Artificial Intelligence at the University of Edinburgh in Scotland. She has a life-long interest in the use of semantic representations in natural language processing and is dedicated to the development of community-wide resources. She was the leader of the English, Chinese, and Korean PropBanks and the Pilot Arabic PropBank. She is now the PI for the Hindi/Urdu Treebank Project and is leading the English, Chinese, and Arabic sense-tagging and PropBanking efforts for the DARPA-GALE OntoNotes project. In addition to building state-of-the-art word-sense taggers and semantic role labelers, she and her students have also developed VerbNet, a public-domain

rich lexical resource that can be used in conjunction with WordNet, and SemLink, a mapping from the PropBank generic arguments to the more fine-grained VerbNet semantic roles as well as to FrameNet Frame Elements. She is a past President of the Association for Computational Linguistics, and a past Chair of SIGHAN and SIGLEX, where she was instrumental in getting the Senseval/Semeval evaluations under way.

Ian Pratt-Hartmann studied Mathematics and Philosophy at Brasenose College, Oxford, and Philosophy at Princeton and Stanford Universities, gaining his PhD from Princeton in 1987. He is currently Senior Lecturer in the Department of Computer Science at the University of Manchester.

Ehud Reiter is a Reader in Computer Science at the University of Aberdeen in Scotland. He completed a PhD in natural language generation at Harvard in 1990 and worked at the University of Edinburgh and at CoGenTex (a small US NLG company) before coming to Aberdeen in 1995. He has published over 100 papers, most of which deal with natural language generation, including the first book ever written on applied NLG. In recent years he has focused on data-to-text systems and related "language and the world" research challenges.

Steve Renals received a BSc in Chemistry from the University of Sheffield in 1986, an MSc in Artificial Intelligence in 1987, and a PhD in Speech Recognition and Neural Networks in 1990, both from the University of Edinburgh. He is a Professor in the School of Informatics, University of Edinburgh, where he is the Director of the Centre for Speech Technology Research. From 1991 to 1992, he was a Postdoctoral Fellow at the International Computer Science Institute, Berkeley, CA, and was then an EPSRC Postdoctoral Fellow in Information Engineering at the University of Cambridge (1992–4). From 1994 to 2003, he was a Lecturer then Reader at the University of Sheffield, moving to the University of Edinburgh in 2003. His research interests are in the area of signal-based approaches to human communication, in particular speech recognition and machine learning approaches to modeling multi-modal data. He has over 150 publications in these areas.

Philip Resnik is an Associate Professor at the University of Maryland, College Park, with joint appointments in the Department of Linguistics and the Institute for Advanced Computer Studies. He completed his PhD in Computer and Information Science at the University of Pennsylvania in 1993. His research focuses on the integration of linguistic knowledge with data-driven statistical modeling, and he has done work in a variety of areas, including computational psycholinguistics, word-sense disambiguation, cross-language information retrieval, machine translation, and sentiment analysis.

Giorgio Satta received a PhD in Computer Science in 1990 from the University of Padua, Italy. He is currently a Full Professor at the Department of Information Engineering, University of Padua. His main research interests are in computational linguistics, mathematics of language and formal language theory.

For the years 2009–10 he is serving as Chair of the European Chapter of the Association for Computational Linguistics (EACL). He has joined the standing

committee of the Formal Grammar conference (FG) and the editorial boards of the journals *Computational Linguistics*, *Grammars* and *Research on Language and Computation*. He has also served as Program Committee Chair for the Annual Meeting of the Association for Computational Linguistics (ACL) and for the International Workshop on Parsing Technologies (IWPT).

Helmut Schmid works as a Senior Scientist at the Institute for Natural Language Processing in Stuttgart with a focus on statistical methods for NLP. He developed a range of tools for tokenization, POS tagging, parsing, computational morphology, and statistical clustering, and he frequently used decision trees in his work.

Antal van den Bosch (MA, Tilburg University, The Netherlands, 1992; PhD, Computer Science, Universiteit Maastricht, The Netherlands, 1997) held Research Assistant positions at the experimental psychology labs of Tilburg University and the Université Libre de Bruxelles (Belgium) in 1993 and 1994. After his PhD project at the Universiteit Maastricht (1994–7), he returned to Tilburg University in 1997 as a postdoc researcher. In 1999 he was awarded a Royal Dutch Academy of Arts and Sciences fellowship, followed in 2001 and 2006 by two consecutively awarded Innovational Research funds of the Netherlands Organisation for Scientific Research. Tilburg University appointed him as Assistant Professor (2001), Associate Professor (2006), and Full Professor in Computational Linguistics and AI (2008). He is also a Guest Professor at the University of Antwerp (Belgium). He currently supervises five PhD students, and has graduated seven PhD students as co-supervisor. His research interests include memory-based natural language processing and modeling, machine translation, and proofing tools.

Prof. **Andy Way** obtained his BSc (Hons) in 1986, MSc in 1989, and PhD in 2001 from the University of Essex, Colchester, UK. From 1988 to 1991 he worked at the University of Essex, UK, on the Eurotra Machine Translation project. He joined Dublin City University (DCU) as a Lecturer in 1991 and was promoted to Senior Lecturer in 2001 and Associate Professor in 2006. He was a DCU Senior Albert College Fellow from 2002 to 2003, and has been an IBM Centers for Advanced Studies Scientist since 2003, and a Science Foundation Ireland Fellow since 2005. He has published over 160 peer-reviewed papers. He has been awarded grants totaling over €6.15 million since 2000, and over €6.6 million in total. He is the Centre for Next Generation Localisation co-ordinator for Integrated Language Technologies (ILT). He currently supervises eight students on PhD programs of study, all of whom are externally funded, and has in addition graduated 10 PhD and 11 MSc students. He is currently the Editor of the journal *Machine Translation*, President of the European Association for Machine Translation, and President-Elect of the International Association for Machine Translation.

Nick Webb is a Senior Research Scientist in the Institute for Informatics, Logics and Security Studies, at the University at Albany, SUNY, USA. Previously he was a Research Fellow in the Natural Language Processing Group at the University of Sheffield, UK, and a Research Officer at the University of Essex, UK, where he obtained a BSc in Computer Science (with a focus on Artificial Intelligence)

and an MSc (in Computational Linguistics). His PhD from Sheffield concerns the analysis of dialogue corpora to build computational models of dialogue-act classification, and his research interests concern intelligent information access, including interactive question answering and dialogue systems.

Bonnie Webber was a Researcher at Bolt Beranek and Newman while working on the PhD she received from Harvard University in 1978. She then taught in the Department of Computer and Information Science at the University of Pennsylvania for 20 years before joining the School of Informatics at the University of Edinburgh. Known for research on discourse and on question answering, she is a Past President of the Association for Computational Linguistics, co-developer (with Aravind Joshi, Rashmi Prasad, Alan Lee, and Eleni Miltsakaki) of the Penn Discourse TreeBank, and co-editor (with Annie Zaenen and Martha Palmer) of the new electronic journal, *Linguistic Issues in Language Technology*.

Shuly Wintner is a Senior Lecturer at the Department of Computer Science, University of Haifa, Israel. His research spans various areas in computational linguistics, including formal grammars, morphology, syntax, development of language resources and machine translation, with a focus on Semitic languages. He has published over 60 scientific papers in computational linguistics. Dr Wintner is the Editor-in-Chief of the journal *Research in Language and Computation*.

Nianwen Xue is an Assistant Professor of Languages & Linguistics and Computer Science at Brandeis University. His research interests include syntactic and semantic parsing, machine translation, temporal representation and inference, Chinese-language processing, and linguistic annotation (Chinese Treebank, Chinese Proposition Bank, OntoNotes). He serves on the ACL SIGANN committee and co-organized the Linguistic Annotation Workshops (LAW II and LAW III) and the 2009 CoNLL Shared Task on Syntactic and Semantic Dependencies in Multiple Languages. He got his PhD in linguistics from the University of Delaware.

Preface

We started work on this handbook three years ago and, while bringing it to fruition has involved a great deal of work, we have enjoyed the process. We are grateful to our colleagues who have contributed chapters to the volume. Its quality is due to their labor and commitment. We appreciate the considerable time and effort that they have invested in making this venture a success. It has been a pleasure working with them.

We owe a debt of gratitude to our editors at Wiley-Blackwell, Danielle Descoteaux and Julia Kirk, for their unstinting support and encouragement throughout this project. We wish that all scientific-publishing projects were blessed with publishers of their professionalism and good nature.

Finally, we must thank our families for enduring the long period of time that we have been engaged in working on this volume. Their patience and good will has been a necessary ingredient for its completion.

The best part of compiling this handbook has been the opportunity that it has given each of us to observe in detail and in perspective the wonderful burst of creativity that has taken hold of our field in recent years.

Alexander Clark, Chris Fox, and Shalom Lappin
London and Wivenhoe
September 2009

Introduction

The field of computational linguistics (CL), together with its engineering domain of natural language processing (NLP), has exploded in recent years. It has developed rapidly from a relatively obscure adjunct of both AI and formal linguistics into a thriving scientific discipline. It has also become an important area of industrial development. The focus of research in CL and NLP has shifted over the past three decades from the study of small prototypes and theoretical models to robust learning and processing systems applied to large corpora. This handbook is intended to provide an introduction to the main areas of CL and NLP, and an overview of current work in these areas. It is designed as a reference and source text for graduate students and researchers from computer science, linguistics, psychology, philosophy, and mathematics who are interested in this area.

The volume is divided into four main parts. Part I contains chapters on the formal foundations of the discipline. Part II introduces the current methods that are employed in CL and NLP, and it divides into three subsections. The first section describes several influential approaches to Machine Learning (ML) and their application to NLP tasks. The second section presents work in the annotation of corpora. The last section addresses the problem of evaluating the performance of NLP systems. Part III of the handbook takes up the use of CL and NLP procedures within particular linguistic domains. Finally, Part IV discusses several leading engineering tasks to which these procedures are applied.

In Chapter 1 Shuly Wintner gives a detailed introductory account of the main concepts of formal language theory. This subdiscipline is one of the primary formal pillars of computational linguistics, and its results continue to shape theoretical and applied work. Wintner offers a remarkably clear guide through the classical language classes of the Chomsky hierarchy, and he exhibits the relations between these classes and the automata or grammars that generate (recognize) their members.

While formal language theory identifies classes of languages and their decidability (or lack of such), complexity theory studies the computational resources

The Handbook of Computational Linguistics and Natural Language Processing, First Edition.
Edited by Alexander Clark, Chris Fox and Shalom Lappin.

in time and space required to compute the elements of these classes. Ian Pratt-Hartmann introduces this central area of computer science in Chapter 2, and he takes up its significance for CL and NLP. He describes a series of important complexity results for several prominent language classes and NLP tasks. He also extends the treatment of complexity in CL/NLP from classical problems, like syntactic parsing, to the relatively unexplored area of computing sentence meaning and logical relations among sentences.

Statistical modeling has become one of the primary tools in CL and NLP for representing natural language properties and processes. In Chapter 3 Ciprian Chelba offers a clear and concise account of the basic concepts involved in the construction of statistical language models. He reviews probabilistic n-gram models and their relation to Markov systems. He defines and clarifies the notions of perplexity and entropy in terms of which the predictive power of a language model can be measured. Chelba compares n-gram models with structured language models generated by probabilistic context-free grammars, and he discusses their applications in several NLP tasks.

Part I concludes with Mark-Jan Nederhof and Giorgio Satta's discussion of the formal foundations of parsing in Chapter 4. They illustrate the problem of recognizing and representing syntactic structure with an examination of (non-lexicalized and lexicalized) context-free grammars (CFGs) and tabular (chart) parsing. They present several CFG parsing algorithms, and they consider probabilistic CFG parsing. They then extend their study to dependency grammar parsers and tree adjoining grammars (TAGs). The latter are mildly context sensitive, and so more formally powerful than CFGs. This chapter provides a solid introduction to the central theoretical concepts and results of a core CL domain.

Robert Malouf opens the first section of Part II with an examination of maximum entropy models in Chapter 5. These constitute an influential machine learning technique that involves minimizing the bias in a probability model for a set of events to the minimal set of constraints required to accommodate the data. Malouf gives a rigorous account of the formal properties of MaxEnt model selection, and exhibits its role in describing natural languages. He compares MaxEnt to support vector machines (SVMs), another ML technique, and he looks at its usefulness in part of speech tagging, parsing, and machine translation.

In Chapter 6 Walter Daelemans and Antal van den Bosch give a detailed overview of memory-based learning (MBL), an ML classification model that is widely used in NLP. MBL invokes a similarity measure to evaluate the distance between the feature vectors of stored training data and those of new events or entities in order to construct classification classes. It is a highly versatile and efficient learning framework that constitutes an alternative to statistical language modeling methods. Daelemans and van den Bosch consider modified and extended versions of MBL, and they review its application to a wide variety of NLP tasks. These include phonological and morphological analysis, part of speech tagging, shallow parsing, word disambiguation, phrasal chunking, named entity recognition, generation, machine translation, and dialogue-act recognition.

Helmut Schmid surveys decision trees in Chapter 7. These provide an efficient procedure for classifying data into descending binary branching subclasses, and they can be quickly induced from large data samples. Schmid points out that simple decision trees often exhibit instability because of their sensitivity to small changes in feature patterns of the data. He considers several modifications of decision trees that overcome this limitation, specifically bagging, boosting, and random forests. These methods combine sets of trees induced for a data set to achieve a more robust classifier. Schmid illustrates the application of decision trees to natural language tasks with discussions of grapheme conversion to phonemes, and POS tagging.

Alex Clark and Shalom Lappin characterize grammar induction as a problem in unsupervised learning in Chapter 8. They compare supervised and unsupervised grammar inference, from both engineering and cognitive perspectives. They consider the costs and benefits of both learning approaches as a way of solving NLP tasks. They conclude that, while supervised systems are currently more accurate than unsupervised ones, the latter will become increasingly influential because of the enormous investment in resources required to annotate corpora for training supervised classifiers. By contrast, large quantities of raw text are readily available online for unsupervised learning. In modeling human language acquisition, unsupervised grammar induction is a more appropriate framework, given that the primary linguistic data available to children is not annotated with sample classifications to be learned. Clark and Lappin discuss recent work in unsupervised POS tagging and grammar inference, and they observe that the most successful of these procedures are beginning to approach the performance levels achieved by state-of-the-art supervised taggers and parsers.

Neural networks are one of the earliest and most influential paradigms of machine learning. James B. Henderson concludes the first section of Part II with an overview in Chapter 9 of neural networks and their application to NLP problems. He considers multi-layered perceptrons (MLPs), which contain hidden units between their inputs and outputs, and recurrent MLPs, which have cyclic links to hidden units. These cyclic links allow the system to process unbounded sequences by storing copies of hidden unit states and feeding them back as input to units when they are processing successive positions in the sequence. In effect, they provide the system with a memory for processing sequences of inputs. Henderson shows how a neural network can be used to calculate probability values for its outputs. He also illustrates the application of neural networks to the tasks of generating statistical language models for a set of data, learning different sorts of syntactic parsing, and identifying semantic roles. He compares them to other machine learning methods and indicates certain equivalence relations that hold between neural networks and these methods.

In the second section (Chapter 10), Martha Palmer and Nianwen Xue address the central issue of corpus annotation. They compare alternative systems for marking corpora and propose clear criteria for achieving adequate results across distinct annotation tasks. They look at a number of important types of linguistic information that annotation encodes including, *inter alia*, POS tagging, deep and

shallow syntactic parsing, coreference and anaphora relations, lexical meanings, semantic roles, temporal connections among propositions, logical entailments among propositions, and discourse structure. Palmer and Xue discuss the problems of securing reasonable levels of annotator agreement. They show how a sound and well-motivated annotation scheme is crucial for the success of supervised machine learning procedures in NLP, as well as for the rigorous evaluation of their performance.

Philip Resnik and Jimmy Lin conclude Part II with a discussion in the last section (Chapter 11) of methods for evaluating NLP systems. They consider both intrinsic evaluation of a procedure's performance for a specified task, and external assessment of its contribution to the quality of a larger engineering system in which it is a component. They present several ways to formulate precise quantitative metrics for grading the output of an NLP device, and they review testing sequences through which these metrics can be applied. They illustrate the issues of evaluation by considering in some detail what is involved in assessing systems for word-sense disambiguation and for question answering. This chapter extends and develops some of the concerns raised in the previous chapter on annotation. It also factors out and addresses evaluation problems that emerged in earlier chapters on the application of machine learning methods to NLP tasks.

Part III opens with Steve Renals and Thomas Hain's comprehensive account in chapter 12 of current work in automatic speech recognition (ASR). They observe that ASR plays a central role in NLP applications involving spoken language, including speech-to-speech translation, dictation, and spoken dialogue systems. Renals and Hain focus on the general task of transcribing natural conversational speech to text, and present the problem in terms of a statistical framework in which the problem of the speech recogniser is to find the most likely word sequence given the observed acoustics. The focus of the chapter is acoustic modeling based on hidden Markov models (HMMs) and Gaussian mixture models. In the first part of the chapter they develop the basic acoustic modeling framework that underlies current speech recognition systems, including refinements to include discriminative training and the adaptation to particular speakers using only small amounts of data. These components are drawn together in the description of a state-of-the-art system for the automatic transcription of multiparty meetings. The final part of the chapter discusses approaches that enable robustness for noisier or less constrained acoustic environments, the incorporation of multiple sources of knowledge, the development of sequence models that are richer than HMMs, and issues that arise when developing large-scale ASR systems.

In Chapter 13 Stephen Clark discusses statistical parsing as the probabilistic syntactic analysis of sentences in a corpus, through supervised learning. He traces the development of this area from generative parsing models to discriminative frameworks. Clark studies Collins' lexicalized probabilistic context-free grammars (PCFGs) as a particularly successful instance of these models. He examines the parsing algorithms, procedures for parse ranking, and methods for parse optimization that are commonly used in generative parse models like PCFG. Discriminative parsing does not model sentences, but provides a way of modeling

parses directly. It discards some of the independence assumptions encoded in generative parsing, and it allows for complex dependencies among syntactic features. Clark examines log-linear (maximum entropy) models as instantiations of this approach. He applies them to parsers driven by combinatory categorial grammar (CCG). He gives a detailed description of recent work on statistical CCG parsing, focusing on the efficiency with which such grammars can be learned, and the impressive accuracy which CCG parsing has recently achieved.

John A. Goldsmith offers a detailed overview in Chapter 14 of computational approaches to morphology. He looks at unsupervised learning of word segmentation for a corpus in which word boundaries have been eliminated, and he identifies two main problems in connection with this task. The first involves identifying the correct word boundaries for a stripped corpus on the basis of prior knowledge of the lexicon of the language. The second, and significantly more difficult, problem is to devise a procedure for constructing the lexicon of the language from the stripped corpus. Goldsmith describes a variety of approaches to word segmentation, highlighting probabilistic modeling techniques, such as minimum description length and hierarchical Bayesian models. He reviews distributional methods for unsupervised morphological learning which have their origins in Zellig Harris' work, and gives a very clear account of finite state transducers and their central role in morphological induction.

In Chapter 15 Chris Fox discusses the major questions driving work in logic-based computational semantics. He focuses on formalized theories of meaning, and examines what properties a semantic representation language must possess in order to be sufficiently expressive while sustaining computational viability. Fox proposes that implementability and tractability be taken as conditions of adequacy on semantic theories. Specifically, these theories must permit efficient computation of the major semantic properties of sentences, phrases, and discourse sequences. He surveys work on type theory, intensionality, the relation between proof theory and model theory, and the dynamic representation of scope and anaphora in leading semantic frameworks. Fox also summarizes current research on corpus-based semantics, specifically the use of latent semantic analysis to identify lexical semantic clusters, methods for word-sense disambiguation, and current work on textual entailment. He reflects on possible connections between the corpus-based approach to semantics and logic-based formal theories of meaning, and he concludes with several interesting suggestions for pursuing these connections.

Jonathan Ginzburg and Raquel Fernández present a comprehensive account in Chapter 16 of recent developments in the computational modeling of dialogue. They first examine a range of central phenomena that an adequate formal theory of dialogue must handle. These include non-sentential fragments, which play an important role in conversation; meta-communicative expressions, which serve as crucial feedback and clarification devices to speakers and hearers; procedures for updating shared information and common ground; and mechanisms for adapting a dialogue to a particular conversational domain. Ginzburg and Fernández propose a formal model of dialogue, KoS, which they formulate in the type theoretic framework of type theory with records. This type theory has the full

power of functional application and abstraction, but it permits the specification of recursively dependent type structures that correspond to re-entrant typed feature structures. They compare their dialogue model to other approaches current in the literature. They conclude by examining some of the issues involved in constructing a robust, wide-coverage dialogue management system, and they consider the application of machine learning methods to facilitate certain aspects of this task.

In Chapter 17 Matthew W. Crocker characterizes the major questions and theoretical developments shaping contemporary work in computational psycholinguistics. He observes that this domain of inquiry shares important objectives with both theoretical linguistics and psycholinguistics. In common with the former, it seeks to explain the way in which humans recognize sentence structure and meaning. Together with the latter, it is concerned to describe the cognitive processing mechanisms through which they achieve these tasks. However, in contrast to both theoretical linguistics and psycholinguistics, computational psycholinguistics models language understanding by constructing systems that can be implemented and rigorously tested. Crocker focuses on syntactic processing, and he discusses the central problem of resolving structural ambiguity. He observes that a general consensus has emerged on the view that sentence processing is incremental, and a variety of constraints (syntactic, semantic, pragmatic, etc.) are available at each point in the processing sequence to resolve or reduce different sources of ambiguity. Crocker considers three main approaches. Symbolic methods use grammars to represent syntactic structure and parsing algorithms to exhibit the way in which humans apply a grammar to sentence recognition. Connectionists employ neural nets as non-symbolic systems of induction and processing. Probabilistic approaches model language interpretation as a stochastic procedure, where this involves generating a probability distribution for the strings produced by an automaton or a grammar of some formal class. Crocker concludes with the observation that computational psycholinguistics (like theoretical linguistics) still tends to view sentence processing in isolation from other cognitive activities. He makes the important suggestion that integrating language understanding into the wider range of human functions in which it figures is likely to yield more accurate accounts of processing and acquisition.

Ralph Grishman starts off Part IV of the handbook with a review, in Chapter 18, of information extraction (IE) from documents. He highlights name, entity, relation, and event extraction as primary IE tasks, and he addresses each in turn. Name extraction consists in identifying names in text and classifying them according to semantic (ontological) type. Entity extraction selects referring phrases, assigns them to semantic classes, and specifies coreference links among them. Relation extraction recognizes pairs of related entities and the semantic type of the relation that holds between them. Event extraction picks out cases of events described in a text, according to semantic type, and it locates the entities that appear in the event. For each of these tasks Grishman traces the development of IE approaches from manually crafted rule-based systems, through supervised machine learning, to semi- and unsupervised methods. He concludes the chapter with some reflections on the challenges and opportunities that the web, with its

enormous resources of online text in a variety of languages and formats, poses for future research in IE.

In Chapter 19 Andy Way presents a systematic overview of the current state of machine translation (MT). He discusses the evolution of statistical machine translation (SMT) from word-based n-gram language models specified for aligned multi-lingual corpora (originally developed by the IBM speech and language group in the 1990s) to the phrase-based SMT (PB-SMT) language models that currently dominate the field. He also looks at the use of both generative and discriminative language models in SMT, and he considers results achieved with both supervised and unsupervised learning methods. Way offers a systematic comparison of PB-SMT with other paradigms of MT, including hierarchical, tree-based, and example-based approaches, as well as traditional rule-based systems, that continue to figure prominently in commercial MT products. He concludes with a detailed discussion of the MT work that his research group is doing. This work applies a hybrid view in which syntactic, morphological, and lexical semantic information is combined with statistical language modeling techniques to maximize the accuracy and efficiency of the distinct components of an MT system. He also discusses the role of MT in contemporary online and spoken applications.

Ehud Reiter describes natural language generation (NLG) in Chapter 20. He characterizes the generation problem as mapping representations in one format (or language) into text in a given language. As he observes, NLG is distinguished from most other areas of NLP by the pervasive complexity of making choices from a large set of alternatives at each point in the generation process. The mapping of representations to text involves resolving numerous one-to-many selections. Reiter identifies three main subtasks for NLG. Document planning determines the content of the representation to be realized in NL text, and the general structure of the content. Microplanning specifies the organization and linguistic structure of the text. Realization produces the text itself. In the course of implementing this sequence of tasks, an NLG procedure must decide on the general format of the message to be realized, the nature of the syntactic units in which it will be encoded, the internal structure of these sentences, and a variety of lexical and stylistic choices. Reiter reviews a number of current NLG systems, and he discusses the central role of NLG in a variety of NLP applications. He concludes with some thoughtful proposals for future research directions in this domain.

Ruslan Mitkov reviews computational analysis of discourse structure in Chapter 21. He begins with algorithms for segmenting text into discourse elements. He then describes three major computational treatments of discourse coherence relations: Hobbs' coherence account, rhetorical structure theory, and centering. He follows this with an extended discussion of anaphora resolution. He points out that accurate anaphora resolution is a necessary condition for success in many tasks, such as MT, text summarization, NLG, and IE. He concludes by surveying some of the significant contributions that discourse modeling has made to a wide variety of NLP applications.

Bonnie Webber and Nick Webb conclude Part IV, and the volume, with a presentation of current work on question answering (QA) in Chapter 22. They

trace the development of QA from early procedures that mapped NL questions into queries in a standard database language for a closed data set, to contemporary open systems that seek answers to questions across a large set of documents, often the entire web. As with other NLP applications, this development has also involved a move from manually crafted rules to machine learning classifiers, and hybrid systems combining rule-based and probabilistic methods. They discuss the relation between QA and text retrieval. While the latter provides documents in response to user queries, the former seeks information expressed as natural language replies. They survey the design and performance of current QA procedures, focusing on the challenges involved in improving their coverage and extending their functionality. An important method for achieving such extension is to incorporate methods for identifying text entailments in order to move beyond simple word pattern matching. These entailments enrich the domain of possible answers that a QA system can consider by adding a set of semantic implications to a question and its range of possible answers. Webber and Webb also take up alternative ways of evaluating QA systems, and they consider issues for future research.

While we have tried to provide as broad and comprehensive a view of CL and NLP as possible, this handbook is, inevitably, not exhaustive. Many more chapters could have been added on a host of important issues, and the field would still not have been fully covered. Considerations of space and manageability have forced us to limit the volume to a subset of central research themes. One might take issue with our selection, or with the way that we have chosen to organize the chapters. We suspect that this would be true for any handbook of this size. In many cases, topics to which one might plausibly devote a separate chapter are treated from different perspectives in a number of chapters. So, for example, finite state methods are discussed in the chapters on formal language theory, complexity, morphology, and speech recognition. Therefore, we were able to forego a distinct chapter on this area. In other instances, important new research, like work on text entailment, is touched on lightly (see the brief discussions of text entailment in the chapters on semantics and QA), but pressures of space and timely production prevented us from including fuller treatments.

The survey of work provided here indicates that both symbolic and information theoretic methods continue to play a major role across a large variety of tasks and domains. Moreover, rather than these approaches being in conflict, there is a strong movement towards hybrid models that integrate different approaches. It seems likely that this trend will continue, as each method carries strengths and weaknesses that complement the other. Symbolic techniques offer compact representations of high level information that generally eludes statistical models, while information theoretic procedures achieve a level of robustness and wide coverage that symbolic systems rarely, if ever, achieve on their own.

Above all the chapters of this volume give a clear view of the remarkable diversity and vitality of research being done in CL and NLP, and the enormous progress that has been made in these areas over the past several decades. We hope that the handbook communicates some of the excitement and the satisfaction that we and our colleagues experience from our work in this amazing field.

Part I Formal Foundations

1 Formal Language Theory

SHULY WINTNER

1 Introduction

This chapter provides a gentle introduction to formal language theory, aimed at readers with little background in formal systems. The motivation is natural language processing (NLP), and the presentation is geared towards NLP applications, with linguistically motivated examples, but without compromising mathematical rigor.

The text covers elementary formal language theory, including: regular languages and regular expressions; languages vs. computational machinery; finite state automata; regular relations and finite state transducers; context-free grammars and languages; the Chomsky hierarchy; weak and strong generative capacity; and mildly context-sensitive languages.

2 Basic Notions

Formal languages are defined with respect to a given *alphabet*, which is a finite set of symbols, each of which is called a *letter*. This notation does not mean, however, that elements of the alphabet must be "ordinary" letters; they can be any symbol, such as numbers, or digits, or words. It is customary to use 'Σ' to denote the alphabet. A finite sequence of letters is called a *string*, or a *word*. For simplicity, we usually forsake the traditional sequence notation in favor of a more straightforward representation of strings.

Example 1 (Strings). Let $\Sigma = \{0, 1\}$ be an alphabet. Then all binary numbers are strings over Σ. Instead of $\langle 0, 1, 1, 0, 1 \rangle$ we usually write 01101. If $\Sigma = \{a, b, c, d, \ldots, y, z\}$ is an alphabet, then *cat*, *incredulous*, and *supercalifragilisticexpialidocious* are strings, as are *tac*, *qqq*, and *kjshdflkwjehr*.

The *length* of a string w is the number of letters in the sequence, and is denoted $|w|$. The unique string of length 0 is called the *empty string* and is usually denoted ϵ (but sometimes λ).

The Handbook of Computational Linguistics and Natural Language Processing, First Edition.
Edited by Alexander Clark, Chris Fox and Shalom Lappin.

Let $w_1 = \langle x_1, \ldots, x_n \rangle$ and $w_2 = \langle y_1, \ldots, y_m \rangle$ be two strings over the same alphabet Σ. The *concatenation* of w_1 and w_2, denoted $w_1 \cdot w_2$, is the string $\langle x_1, \ldots, x_n, y_1, \ldots, y_m \rangle$. Note that the length of $w_1 \cdot w_2$ is the sum of the lengths of w_1 and w_2: $|w_1 \cdot w_2| = |w_1| + |w_2|$. When it is clear from the context, we sometimes omit the '$\cdot$' symbol when depicting concatenation.

Example 2 (Concatenation). Let $\Sigma = \{a, b, c, d, \ldots, y, z\}$ be an alphabet. Then *master* $\cdot$ *mind* = *mastermind*, *mind* $\cdot$ *master* = *mindmaster*, and *master* $\cdot$ *master* = *mastermaster*. Similarly, *learn* $\cdot$ *s* = *learns*, *learn* $\cdot$ *ed* = *learned*, and *learn* $\cdot$ *ing* = *learning*.

Notice that when the empty string ϵ is concatenated with any string w, the resulting string is w. Formally, for every string w, $w \cdot \epsilon = \epsilon \cdot w = w$.

We define an *exponent* operator over strings in the following way: for every string w, w^0 (read: w raised to the power of zero) is defined as ϵ. Then, for $n > 0$, w^n is defined as $w^{n-1} \cdot w$. Informally, w^n is obtained by concatenating w with itself n times. In particular, $w^1 = w$.

Example 3 (Exponent). If $w =$ *go*, then $w^0 = \epsilon$, $w^1 = w =$ *go*, $w^2 = w^1 \cdot w = w \cdot w =$ *gogo*, $w^3 =$ *gogogo*, and so on.

A few other notions that will be useful in the sequel: the *reversal* of a string w is denoted w^R and is obtained by writing w in the reverse order. Thus, if $w = \langle x_1, x_2, \ldots, x_n \rangle$, $w^R = \langle x_n, x_{n-1}, \ldots, x_1 \rangle$.

Example 4 (Reversal). Let $\Sigma = \{a, b, c, d, \ldots, y, z\}$ be an alphabet. If w is the string *saw*, then w^R is the string *was*. If $w =$ *madam*, then $w^R =$ *madam* $= w$. In this case we say that w is a *palindrome*.

Given a string w, a *substring* of w is a sequence formed by taking contiguous symbols of w in the order in which they occur in w: w_c is a substring of w if and only if there exist (possibly empty) strings w_l and w_r such that $w = w_l \cdot w_c \cdot w_r$. Two special cases of substrings are *prefix* and *suffix*: if $w = w_l \cdot w_c \cdot w_r$ then w_l is a prefix of w and w_r is a suffix of w. Note that every prefix and every suffix is a substring, but not every substring is a prefix or a suffix.

Example 5 (Substrings). Let $\Sigma = \{a, b, c, d, \ldots, y, z\}$ be an alphabet and $w =$ *indistinguishable* a string over Σ. Then ϵ, *in*, *indis*, *indistinguish*, and *indistinguishable* are prefixes of w, while ϵ, *e*, *able*, *distinguishable* and *indistinguishable* are suffixes of w. Substrings that are neither prefixes nor suffixes include *distinguish*, *gui*, and *is*.

Given an alphabet Σ, the set of all strings over Σ is denoted by Σ^* (the reason for this notation will become clear presently). Notice that no matter what the Σ is, as long as it includes at least one symbol, Σ^* is always infinite. A *formal language* over an alphabet Σ is any subset of Σ^*. Since Σ^* is always infinite, the number of formal languages over Σ is also infinite.

As the following example demonstrates, formal languages are quite unlike what one usually means when one uses the term "language" informally. They

are essentially sets of strings of characters. Still, all natural languages are, at least superficially, such string sets. Higher-level notions, relating the strings to objects and actions in the world, are completely ignored by this view. While this is a rather radical idealization, it is a useful one.

Example 6 (Languages). Let $\Sigma = \{a, b, c, \ldots, y, z\}$. Then Σ^* is the set of all strings over the Latin alphabet. Any subset of this set is a language. In particular, the following are formal languages:

- Σ^*;
- the set of strings consisting of consonants only;
- the set of strings consisting of vowels only;
- the set of strings each of which contains at least one vowel and at least one consonant;
- the set of palindromes: strings that read the same from right to left and from left to right;
- the set of strings whose length is less than 17 letters;
- the set of single-letter strings;
- the set {*i, you, he, she, it, we, they*};
- the set of words occurring in Joyce's Ulysses (ignoring punctuation etc.);
- the empty set.

Note that the first five languages are infinite while the last five are finite.

We can now lift some of the string operations defined above to languages. If L is a language then the *reversal* of L, denoted L^R, is the language $\{w \mid w^R \in L\}$, that is, the set of reversed L-strings. *Concatenation* can also be lifted to languages: if L_1 and L_2 are languages, then $L_1 \cdot L_2$ is the language defined as $\{w_1 \cdot w_2 \mid w_1 \in L_1 \textit{ and } w_2 \in L_2\}$: the concatenation of two languages is the set of strings obtained by concatenating some word of the first language with some word of the second.

Example 7 (Language operations). Let $L_1 = \{$*i, you, he, she, it, we, they*$\}$ and $L_2 = \{$*smile, sleep*$\}$. Then $L_1^R = \{$*i, uoy, eh, ehs, ti, ew, yeht*$\}$ and $L_1 \cdot L_2 = \{$*ismile, yousmile, hesmile, shesmile, itsmile, wesmile, theysmile, isleep, yousleep, hesleep, shesleep, itsleep, wesleep, theysleep*$\}$.

In the same way we can define the *exponent* of a language: if L is a language then L^0 is the language containing the empty string only, $\{\epsilon\}$. Then, for $i > 0$, $L^i = L \cdot L^{i-1}$, that is, L^i is obtained by concatenating L with itself i times.

Example 8 (Language exponentiation). Let L be the set of words {*bau, haus, hof, frau*}. Then $L^0 = \{\epsilon\}$, $L^1 = L$ and $L^2 = \{$*baubau, bauhaus, bauhof, baufrau, hausbau, haushaus, haushof, hausfrau, hofbau, hofhaus, hofhof, hoffrau, fraubau, frauhaus, frauhof, fraufrau*$\}$.

The language obtained by considering any number of concatenations of words from L is called the *Kleene closure* of L and is denoted L^*. Formally, $L^* = \bigcup_{i=0}^{\infty} L^i$,

which is a terse notation for the union of L^0 with L^1, then with L^2, L^3 and so on ad infinitum. When one wants to leave L^0 out, one writes $L^+ = \bigcup_{i=1}^{\infty} L^i$.

Example 9 (Kleene closure). Let $L = \{dog, cat\}$. Observe that $L^0 = \{\epsilon\}$, $L^1 = \{dog, cat\}$, $L^2 = \{catcat, catdog, dogcat, dogdog\}$, etc. Thus L^* contains, among its infinite set of strings, the strings ϵ, *cat, dog, catcat, catdog, dogcat, dogdog, catcatcat, catdogcat, dogcatcat, dogdogcat*, etc.

As another example, consider the alphabet $\Sigma = \{a, b\}$ and the language $L = \{a, b\}$ defined over Σ. L^* is the set of all strings over a and b, which is exactly the definition of Σ^*. The notation for Σ^* should now become clear: it is simply a special case of L^*, where $L = \Sigma$.

3 Language Classes and Linguistic Formalisms

Formal languages are sets of strings, subsets of Σ^*, and they can be specified using any of the specification methods for sets (of course, since languages may be infinite, stipulation of their members is in the general case infeasible). When languages are fairly simple (not arbitrarily complex), they can be characterized by means of *rules*. In the following sections we define several mechanisms for defining languages, and focus on the *classes* of languages that can be defined with these mechanisms. A formal mechanism with which formal languages can be defined is a *linguistic formalism*. We use L (with or without subscripts) to denote languages, and $\mathcal{L}$ to denote classes of languages.

Example 10 (Language class). Let $\Sigma = \{a, b, c, \ldots, y, z\}$. Let $\mathcal{L}$ be the set of all the *finite* subsets of Σ^*. Then $\mathcal{L}$ is a language class.

When classes of languages are discussed, some of the interesting properties to be investigated are *closures* with respect to certain operators. The previous section defined several operators, such as concatenation, union, Kleene closure, etc., on languages. Given a particular (binary) operation, say union, it is interesting to know whether a class of languages is *closed under* this operation. A class of languages $\mathcal{L}$ is said to be closed under some operation '$\bullet$' if and only if, whenever two languages L_1 and L_2 are in the class ($L_1, L_2 \in \mathcal{L}$), the result of performing the operation on the two languages is also in this class: $L_1 \bullet L_2 \in \mathcal{L}$.

Closure properties have a theoretical interest in and by themselves, but they are especially important when one is interested in processing languages. Given an efficient computational implementation for a class of languages (for example, an algorithm that determines *membership*: whether a given string indeed belongs to a given language), one can use the operators that the class is closed under, and still preserve computational efficiency in processing. We will see such examples in the following sections.

The membership problem is one of the fundamental questions of interest concerned with language classes. As we shall see, the more expressive the class, the harder it is to determine membership in languages of this class. Algorithms that determine membership are called *recognition* algorithms; when a recognition

algorithm additionally provides the structure that the formalism induces on the string in question, it is called a *parsing* algorithm.

4 Regular Languages

4.1 Regular expressions

The first linguistic formalism we discuss is *regular expressions*. These are expressions over some alphabet Σ, augmented by some special characters. We define a mapping, called *denotation*, from regular expressions to sets of strings over Σ, such that every well-formed regular expression denotes a set of strings, or a language.

DEFINITION 1. *Given an alphabet Σ, the set of **regular expressions** over Σ is defined as follows:*

- *$\emptyset$ is a regular expression;*
- *ϵ is a regular expression;*
- *if $a \in \Sigma$ is a letter, then a is a regular expression;*
- *if r_1 and r_2 are regular expressions, then so are $(r_1 + r_2)$ and $(r_1 \cdot r_2)$;*
- *if r is a regular expression, then so is $(r)^*$;*
- *nothing else is a regular expression over Σ.*

Example 11 (Regular expressions). Let Σ be the alphabet $\{a, b, c, \ldots, y, z\}$. Some regular expressions over this alphabet are $\emptyset$, a, $((c \cdot a) \cdot t)$, $(((m \cdot e) \cdot (o)^*) \cdot w)$, $(a + (e + (i + (o + u))))$, $((a + (e + (i + (o + u)))))^*$, etc.

DEFINITION 2. *Given a regular expression r, its **denotation, $[\![r]\!]$**, is a set of strings defined as follows:*

- *$[\![\emptyset]\!] = \{\}$, the empty set;*
- *$[\![\epsilon]\!] = \{\epsilon\}$, the singleton set containing the empty string;*
- *if $a \in \Sigma$ is a letter, then $[\![a]\!] = \{a\}$, the singleton set containing a only;*
- *if r_1 and r_2 are two regular expressions whose denotations are $[\![r_1]\!]$ and $[\![r_2]\!]$, respectively, then $[\![(r_1 + r_2)]\!] = [\![r_1]\!] \cup [\![r_2]\!]$ and $[\![(r_1 \cdot r_2)]\!] = [\![r_1]\!] \cdot [\![r_2]\!]$;*
- *if r is a regular expression whose denotation is $[\![r]\!]$ then $[\![(r)^*]\!] = [\![r]\!]^*$.*

Example 12 (Regular expressions). Following are the denotations of the regular expressions of the previous example:

$\emptyset$	$\emptyset$
ϵ	$\{\epsilon\}$
a	$\{a\}$
$((c \cdot a) \cdot t)$	$\{c \cdot a \cdot t\}$
$(((m \cdot e) \cdot (o)^*) \cdot w)$	$\{mew, meow, meoow, meooow, meoooow, \ldots\}$
$(a + (e + (i + (o + u))))$	$\{a, e, i, o, u\}$
$((a + (e + (i + (o + u)))))^*$	*the set containing all strings of 0 or more vowels*

Regular expressions are useful because they facilitate specification of complex languages in a formal, concise way. Of course, finite languages can still be specified by enumerating their members; but infinite languages are much easier to specify with a regular expression, as the last instance of the above example shows.

For simplicity, we omit the parentheses around regular expressions when no confusion can be caused. Thus, the expression $((a + (e + (i + (o + u)))))^*$ is written as $(a + e + i + o + u)^*$. Also, if $\Sigma = \{a_1, a_2, \ldots, a_n\}$, we use Σ as a shorthand notation for $a_1 + a_2 + \cdots + a_n$. As in the case of string concatenation and language concatenation, we sometimes omit the '$\cdot$' operator in regular expressions, so that the expression $c \cdot a \cdot t$ can be written *cat*.

Example 13 (Regular expressions). Given the alphabet of all English letters, $\Sigma = \{a, b, c, \ldots, y, z\}$, the language Σ^* is denoted by the regular expression Σ^*. The set of all strings which contain a vowel is denoted by $\Sigma^* \cdot (a + e + i + o + u) \cdot \Sigma^*$. The set of all strings that begin in "*un*" is denoted by $(un)\Sigma^*$. The set of strings that end in either "*tion*" or "*sion*" is denoted by $\Sigma^* \cdot (s + t) \cdot (ion)$. Note that all these languages are infinite.

The class of languages which can be expressed as the denotation of regular expressions is called the class of *regular languages*.

DEFINITION 3. *A language L is **regular** iff there exists a regular expression r such that $L = [\![r]\!]$.*

It is a mathematical fact that some languages, subsets of Σ^*, are not regular. We will encounter such languages in the sequel.

4.2 Properties of regular languages

The class of regular languages is interesting because of its "nice" properties, which we review here. It should be fairly easy to see that regular languages are closed under union, concatenation, and Kleene closure. Given two regular languages, L_1 and L_2, there must exist two regular expressions, r_1 and r_2, such that $[\![r_1]\!] = L_1$ and $[\![r_2]\!] = L_2$. It is therefore possible to form new regular expressions based on r_1 and r_2, such as $r_1 \cdot r_2$, $r_1 + r_2$ and r_1^*. Now, by the definition of regular expressions and their denotations, it follows that the denotation of $r_1 \cdot r_2$ is $L_1 \cdot L_2$: $[\![r_1 \cdot r_2]\!] = L_1 \cdot L_2$. Since $r_1 \cdot r_2$ is a regular expression, its denotation is a regular language, and hence $L_1 \cdot L_2$ is a regular language. Hence the regular languages are closed under concatenation. In exactly the same way we can prove that the class of regular languages is closed under union and Kleene closure.

One of the reasons for the attractiveness of regular languages is that they are known to be closed under a wealth of useful operations: intersection, complementation, exponentiation, substitution, homomorphism, etc. These properties come in handy both in practical applications that use regular languages and in mathematical proofs that concern them. For example, several formalisms extend regular expressions by allowing one to express regular languages using not only the three

basic operations, but also a wealth of other operations (that the class of regular languages is closed under). It is worth noting that such "good behavior" is not exhibited by more complex classes of languages.

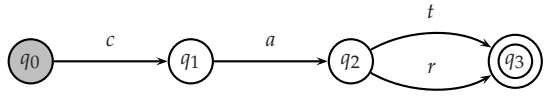
4.3 Finite state automata

Regular expressions are a declarative formalism for specifying (regular) languages. We now present languages as entities generated by a *computation*. This is a very common situation in formal language theory: many language classes are associated with computing machinery that generates them. The dual view of languages (as the denotation of some specifying formalism and as the output of a computational process) is central in formal language theory.

The computational device we define in this section is *finite state automata* (FSA). Informally, they consist of a finite set of *states* (sometimes called *nodes* or *vertices*), connected by a finite number of *transitions* (also called *edges* or *links*). Each of the transitions is labeled by a letter, taken from some finite alphabet Σ. A computation starts at a designated state, the *start* state or *initial* state, and it moves from one state to another along the labeled transitions. As it moves, it prints the letter which labels the transition. Thus, during a computation, a string of letters is printed out. Some of the states of the machine are designated *final* states, or *accepting* states. Whenever the computation reaches a final state, the string that was printed so far is said to be *accepted* by the machine. Since each computation defines a string, the set of all possible computations defines a set of strings or, in other words, a language. We say that this language is *accepted* or *generated* by the machine.

DEFINITION 4. *A **finite state automaton** is a five-tuple $\langle Q, q_0, \Sigma, \delta, F\rangle$, where Σ is a finite set of **alphabet** symbols, Q is a finite set of **states**, $q_0 \in Q$ is the **initial state**, $F \subseteq Q$ is a set of **final** states, and $\delta : Q \times \Sigma \times Q$ is a relation from states and alphabet symbols to states.*

Example 14 (Finite state automata). Finite state automata are depicted graphically, with circles for states and arrows for the transitions. The initial state is shaded and the final states are depicted by two concentric circles. The finite state automaton $A = \langle Q, \Sigma, q_0, \delta, F\rangle$, where $Q = \{q_0, q_1, q_2, q_3\}$, $\Sigma = \{c, a, t, r\}$, $F = \{q_3\}$, and $\delta = \{\langle q_0, c, q_1\rangle, \langle q_1, a, q_2\rangle, \langle q_2, t, q_3\rangle, \langle q_2, r, q_3\rangle\}$, is depicted graphically as follows:



To define the *language* generated by an FSA, we first extend the transition relation from single edges to *paths* by extending the transition relation δ to its reflexive transitive closure, $\hat{\delta}$. This relation assigns a string to each path (it also assumes that an empty path, decorated by ϵ, leads from each state to itself). We focus on paths that lead from the initial state to some final state. The strings that decorate these paths are said to be *accepted* by the FSA, and the *language* of the FSA is the set of all these strings. In other words, in order for a string to be in the

language of the FSA, there must be a path in the FSA which leads from the initial state to some final state decorated by the string. Paths that lead to non-final states do not define accepted strings.

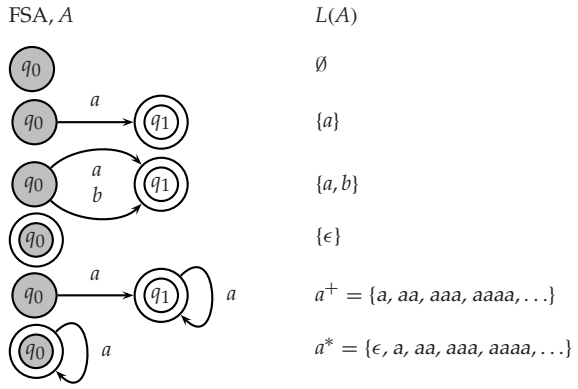
DEFINITION 5. *Given an FSA $A = \langle Q, q_0, \Sigma, \delta, F\rangle$, the reflexive transitive closure of the transition relation δ is $\hat{\delta}$, defined as follows:*

- *for every state $q \in Q$, $(q, \epsilon, q) \in \hat{\delta}$;*
- *for every string $w \in \Sigma^*$ and letter $a \in \Sigma$, if $(q, w, q') \in \hat{\delta}$ and $(q', a, q'') \in \delta$, then $(q, w \cdot a, q'') \in \hat{\delta}$.*

*A string w is **accepted** by A if and only if there exists a state $q_f \in F$ such that $\hat{\delta}(q_0, w) = q_f$. The **language** of A is the set of all the strings accepted by it: $L(A) = \{w \mid$ there exists $q_f \in F$ such that $\hat{\delta}(q_0, w) = q_f\}$.*

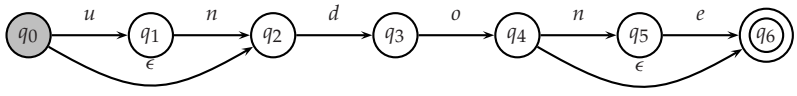
Example 15 (Language accepted by an FSA). For the finite state automaton of Example 14, $\hat{\delta}$ is the following set of triples: $\langle q_0, \epsilon, q_0\rangle$, $\langle q_1, \epsilon, q_1\rangle$, $\langle q_2, \epsilon, q_2\rangle$, $\langle q_3, \epsilon, q_3\rangle$, $\langle q_0, c, q_1\rangle$, $\langle q_1, a, q_2\rangle$, $\langle q_2, t, q_3\rangle$, $\langle q_2, r, q_3\rangle$, $\langle q_0, ca, q_2\rangle$, $\langle q_1, at, q_3\rangle$, $\langle q_1, ar, q_3\rangle$, $\langle q_0, cat, q_3\rangle$, $\langle q_0, car, q_3\rangle$. The language of the FSA is thus {*cat, car*}.

Example 16 (Finite state automata). Following are some simple FSA and the languages they generate.



We now slightly amend the definition of finite state automata to include what is called *ϵ-moves*. By our original definition, the transition relation δ is a relation from states and alphabet symbols to states. We extend δ such that its second coordinate is now $\Sigma \cup \{\epsilon\}$, that is, any edge in an automaton can be labeled either by some alphabet symbol or by the special symbol ϵ, which as usual denotes the empty word. The implication is that a computation can move from one state to another over an ϵ-transition without printing out any symbol.

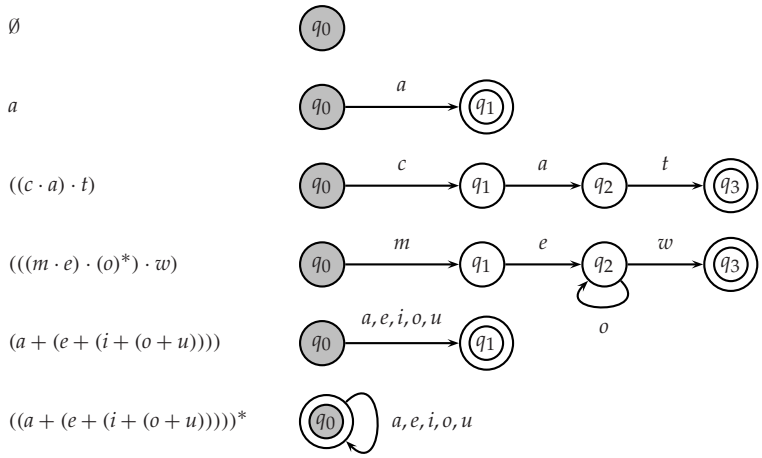
Example 17 (Automata with ϵ-moves). The language accepted by the following automaton is {*do, undo, done, undone*}:



Finite state automata, just like regular expressions, are devices for defining formal languages. The major theorem of regular languages states that the class of languages which can be generated by FSA is exactly the class of regular languages. Furthermore, there are simple and efficient algorithms for "translating" a regular expression to an equivalent automaton and vice versa.

THEOREM 1. *A language L is regular iff there exists an FSA A such that $L = L(A)$.*

Example 18 (Equivalence of finite state automata and regular expressions). For each of the regular expressions of Example 12 we depict an equivalent automaton below:



4.4 Minimization and determinization

The finite state automata presented above are non-deterministic. By this we mean that when the computation reaches a certain state, the next state is not uniquely determined by the next alphabet symbol to be printed. There might very well be more than one state that can be reached by a transition that is labeled by some symbol. This is because we defined automata using a transition relation, δ, which is not required to be functional. For some state q and alphabet symbol a, δ might include the two pairs $\langle q, a, q_1 \rangle$ and $\langle q, a, q_2 \rangle$ with $q_1 \neq q_2$. Furthermore, when we extended δ to allow ϵ-transitions, we added yet another dimension of non-determinism: when the machine is in a certain state q and an ϵ-arc leaves q, the computation must "guess" whether to traverse this arc.

DEFINITION 6. *An FSA $A = \langle Q, q_0, \Sigma, \delta, F \rangle$ is **deterministic** iff it has no ϵ-transitions and δ is a function from $Q \times \Sigma$ to Q.*

Much of the appeal of finite state automata lies in their efficiency; and their efficiency is in great part due to the fact that, given some deterministic FSA A and a string w, it is possible to determine whether or not $w \in L(A)$ by "walking" the path labeled w, starting with the initial state of A, and checking whether the walk leads to a final state. Such a walk takes time that is proportional to the length of w, and is completely independent of the number of states in A. We therefore say that

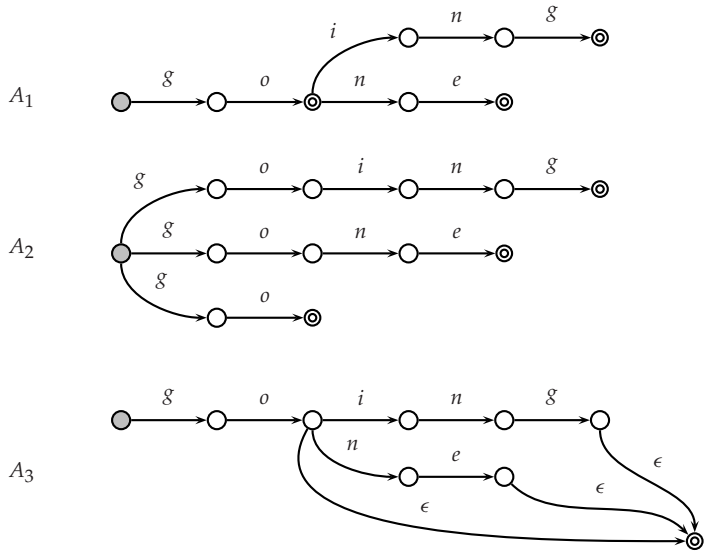
the *membership* problem for FSA can be solved in *linear* time. But when automata are non-deterministic, an element of guessing is introduced, which may impair the efficiency: no longer is there a single walk along a single path labeled w, and some control mechanism must be introduced to check that all possible paths are taken.

Non-determinism is important because it is sometimes much easier to construct a non-deterministic automaton for some language. Fortunately, we can rely on two very important results: every non-deterministic finite state automaton is equivalent to some deterministic one; and every finite state automaton is equivalent to one that has a minimum number of nodes, and the minimal automaton is unique. We now explain these results.

First, it is important to clarify what is meant by *equivalent*. We say that two finite state automata are equivalent if and only if they accept the same language.

DEFINITION 7. *Two FSA A_1 and A_2 are* ***equivalent*** *iff $L(A_1) = L(A_2)$.*

Example 19 (Equivalent automata). The following three finite state automata are equivalent: they all accept the set {*go, gone, going*}.



Note that A_1 is deterministic: for any state and alphabet symbol there is at most one possible transition. A_2 is not deterministic: the initial state has three outgoing arcs all labeled by g. The third automaton, A_3, has ϵ-arcs and hence is non-deterministic. While A_2 might be the most readable, A_1 is the most compact as it has the fewest nodes.

Given a non-deterministic FSA A, it is always possible to construct an equivalent deterministic automaton, one whose next state is fully determined by the current state and the alphabet symbol, and which contains no ϵ-moves. Sometimes this construction yields an automaton with more states than the original,

non-deterministic one (in the worst case, the number of states in the deterministic automaton can be exponential in the size of the non-deterministic one). However, the deterministic automaton can then be minimized such that it is guaranteed that no deterministic finite state automaton generating the same language is smaller. Thus, it is always possible to determinize and then minimize a given automaton without affecting the language it generates.

THEOREM 2. *For every FSA A (with n states) there exists a deterministic FSA A' (with at most 2^n states) such that $L(A) = L(A')$.*

THEOREM 3. *For every regular language L there exists a minimal FSA A such that no other FSA A' such that $L(A) = L(A')$ has fewer states than A. A is unique (up to isomorphism).*

4.5 Operations on finite state automata

We know from Section 4.3 that finite state automata are equivalent to regular expressions; we also know from Section 4.2 that the regular languages are closed under several operations, including union, concatenation, and Kleene closure. So, for example, if L_1 and L_2 are two regular languages, there exist automata A_1 and A_2 which accept them, respectively. Since we know that $L_1 \cup L_2$ is also a regular language, there must be an automaton which accepts it as well. The question is, can this automaton be constructed using the automata A_1 and A_2? In this section we show how simple operations on finite state automata correspond to some operators on languages.

We start with concatenation. Suppose that A_1 is a finite state automaton such that $L(A_1) = L_1$, and similarly that A_2 is an automaton such that $L(A_2) = L_2$. We describe an automaton A such that $L(A) = L_1 \cdot L_2$. A word w is in $L_1 \cdot L_2$ if and only if it can be broken into two parts, w_1 and w_2, such that $w = w_1 \cdot w_2$, and $w_1 \in L_1$, $w_2 \in L_2$. In terms of automata, this means that there is an accepting path for w_1 in A_1 and an accepting path for w_2 in A_2; so if we allow an ϵ-transition from all the final states of A_1 to the initial state of A_2, we will have accepting paths for words of $L_1 \cdot L_2$. The finite state automaton A is constructed by combining A_1 and A_2 in the following way: its set of states, Q, is the union of Q_1 and Q_2; its alphabet is the union of the two alphabets; its initial state is the initial state of A_1; its final states are the final states of A_2; and its transition relation is obtained by adding to $\delta_1 \cup \delta_2$ the set of ϵ-moves described above: $\{\langle q_f, \epsilon, q_{0_2}\rangle \mid q_f \in F_1\}$ where q_{0_2} is the initial state of A_2.

In a very similar way, an automaton A can be constructed whose languages is $L_1 \cup L_2$ by combining A_1 and A_2. Here, one should notice that for a word to be accepted by A it must be accepted either by A_1 or by A_2 (or by both). The combined automaton will have an accepting path for every accepting path in A_1 and in A_2. The idea is to add a new initial state to A, from which two ϵ-arcs lead to the initial states of A_1 and A_2. The states of A are the union of the states of A_1 and A_2, plus

the new initial state. The transition relation is the union of δ_1 with δ_2, plus the new ϵ-arcs. The final states are the union of F_1 and F_2.

An extension of the same technique to construct the Kleene closure of an automaton is rather straightforward. However, all these results are not surprising, as we have already seen in Section 4.2 that the regular languages are closed under these operations. Thinking of languages in terms of the automata that accept them comes in handy when one wants to show that the regular languages are closed under other operations, where the regular expression notation is not very suggestive of how to approach the problem. Consider the operation of *complementation*: if L is a regular language over an alphabet Σ, we say that the complement of L is the set of all the words (in Σ^*) that are not in L, and write $\overline{L}$ for this set. Formally, $\overline{L} = \Sigma^* \setminus L$. Given a regular expression r, it is not clear what regular expression r' is such that $[\![r']\!] = \overline{[\![r]\!]}$. However, with automata this becomes much easier.

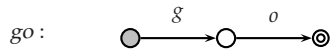
Assume that a finite state automaton A is such that $L(A) = L$. Assume also that A is deterministic. To construct an automaton for the complemented language, all one has to do is change all final states to non-final, and all non-final states to final. In other words, if $A = \langle Q, \Sigma, q_0, \delta, F\rangle$, then $\overline{A} = \langle Q, \Sigma, q_0, \delta, Q \setminus F\rangle$ is such that $L(\overline{A}) = \overline{L}$. This is because every accepting path in A is not accepting in $\overline{A}$, and vice versa.

Now that we know that the regular languages are closed under complementation, it is easy to show that they are closed under intersection: if L_1 and L_2 are regular languages, then $L_1 \cap L_2$ is also regular. This follows directly from fundamental theorems of set theory, since $L_1 \cap L_2$ can actually be written as $\overline{\overline{L_1} \cup \overline{L_2}}$, and we already know that the regular languages are closed under union and complementation. In fact, construction of an automaton for the intersection language is not very difficult, although it is less straightforward than the previous examples.

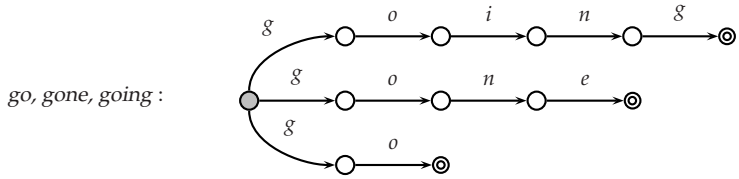
4.6 *Applications of finite state automata in natural language processing*

Finite state automata are computational devices that generate regular languages, but they can also be viewed as *recognizing* devices: given some automaton A and a word w, it is easy to determine whether $w \in L(A)$. Observe that such a task can be performed in time linear in the length of w, hence the efficiency of the representation is optimal. This reversed view of automata motivates their use for a simple yet necessary application of natural language processing: dictionary lookup.

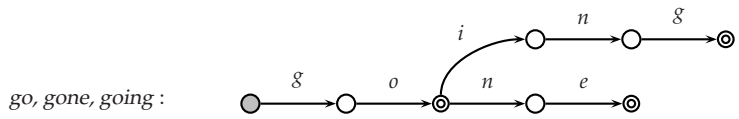
Example 20 (Dictionaries as finite state automata). Many NLP applications require the use of lexicons or dictionaries, sometimes storing hundreds of thousands of entries. Finite state automata provide an efficient means for storing dictionaries, accessing them, and modifying their contents. Assume that an alphabet is fixed (say, $\Sigma = \{a, b, \ldots, z\}$) and consider how a single word, say *go*, can be represented. As we have seen above, a naïve representation would be to construct an automaton with a single path whose arcs are labeled by the letters of the word *go*:



To represent more than one word, add paths to the FSA, one path for each additional word. For example, after adding the words *gone* and *going*, we obtain:



This automaton can then be determinized and minimized, yielding:



The organization of the lexicon as outlined above is extremely simplistic. A possible extension attaches to the final states of the FSA additional information pertaining to the words that decorate the paths to those states. Such information can include definitions, morphological information, translations, etc. FSA are thus suitable for representing various kinds of dictionaries, in addition to simple lexicons.

Regular languages are particularly appealing for natural language processing for two main reasons. First, it turns out that most phonological and morphological processes can be straightforwardly described using the operations that regular languages are closed under, in particular concatenation. With very few exceptions (such as the interdigitation word-formation processes of Semitic languages or the duplication phenomena of some Asian languages), the morphology of most natural languages is limited to simple concatenation of affixes, with some morphophonological alternations, usually on a morpheme boundary. Such phenomena are easy to model with regular languages, and hence are easy to implement with finite state automata. Second, many of the algorithms one would want to apply to finite state automata take time proportional to the length of the word being processed, independently of the size of the automaton. Finally, the various closure properties facilitate modular development of FSA for natural languages.

4.7 Regular relations

While finite state automata, which *define* (regular) languages, are sufficient for some natural language applications, it is often useful to have a mechanism for *relating* two (formal) languages. For example, a part-of-speech tagger can be viewed as an application that relates a set of natural language strings (the *source* language) to a set of part-of-speech tags (the *target* language). A morphological

analyzer can be viewed as a relation between natural language strings (the surface forms of words) and their internal structure (say, as sequences of morphemes). In this section we discuss a computational device, very similar to finite state automata, which defines a *relation* over two regular languages.

Example 21 (Relations over languages). Consider a simple part-of-speech tagger: an application which associates with every word in some natural language a tag, drawn from a finite set of tags. In terms of formal languages, such an application implements a relation over two languages. Assume that the natural language is defined over $\Sigma_1 = \{a, b, \ldots, z\}$ and that the set of tags is $\Sigma_2 = \{$*PRON, V, DET, ADJ, N, P*$\}$. Then the part-of-speech relation might contain the following pairs (here, a string over Σ_1 is mapped to a single element of Σ_2):

I	PRON	*the*	DET
know	V	*Cat*	N
some	DET	*in*	P
new	ADJ	*the*	DET
tricks	N	*Hat*	N
said	V		

As another example, assume that Σ_1 is as above, and Σ_2 is a set of part-of-speech and morphological tags, including {*-PRON, -V, -DET, -ADJ, -N, -P, -1, -2, -3, -sg, -pl, -pres, -past, -def, -indef*}. A morphological analyzer is a relation between a language over Σ_1 and a language over Σ_2. Some of the pairs in such a relation are:

I	I-PRON-1-sg	*the*	the-DET-def
know	know-V-pres	*Cat*	cat-N-sg
some	some-DET-indef	*in*	in-P
new	new-ADJ	*the*	the-DET-def
tricks	trick-N-pl	*Hat*	hat-N-sg
said	say-V-past		

Finally, consider the relation that maps every English noun in singular to its plural form. While the relation is highly regular (namely, adding "*s*" to the singular form), some nouns are irregular. Some instances of this relation are:

cat	*cats*	*hat*	*hats*
ox	*oxen*	*child*	*children*
mouse	*mice*	*sheep*	*sheep*
goose	*geese*		

Summing up, a regular relation is defined over *two* alphabets, Σ_1 and Σ_2. Of course, the two alphabets can be identical, but for many natural language applications they differ. If a relation in $\Sigma^* \times \Sigma^*$ is regular, its projections on both coordinates are regular languages (not all relations that satisfy this condition are regular; additional constraints must hold on the underlying mapping which we

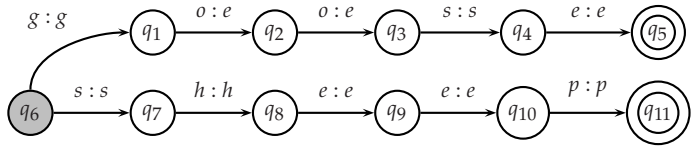
ignore here). Informally, a regular relation is a set of pairs, each of which consists of one string over Σ_1 and one string over Σ_2, such that both the set of strings over Σ_1 and that over Σ_2 constitute regular languages. We provide a precise characterization of regular relations via finite state transducers below.

4.8 Finite state transducers

Finite state automata are a computational device for defining regular languages; in a very similar way, *finite state transducers (FSTs)* are a computational device for defining regular relations. Transducers are similar to automata, the only difference being that the edges are not labeled by single letters, but rather by pairs of symbols: one symbol from Σ_1 and one symbol from Σ_2. The following is a preliminary definition that we will revise presently:

DEFINITION 8. *A **finite state transducer** is a six-tuple $\langle Q, q_0, \Sigma_1, \Sigma_2, \delta, F\rangle$, where Q is a finite set of states, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states, Σ_1 and Σ_2 are alphabets, and δ is a subset of $Q \times \Sigma_1 \times \Sigma_2 \times Q$.*

Example 22 (Finite state transducers). Following is a finite state transducer relating the singular forms of two English words with their plural form. In this case, both alphabets are identical: $\Sigma_1 = \Sigma_2 = \{a, b, \ldots, z\}$. The set of nodes is $Q = \{q_1, q_2, \ldots, q_{11}\}$, the initial state is q_6 and the set of final states is $F = \{q_5, q_{11}\}$. The transitions from one state to another are depicted as labeled edges; each edge bears two symbols, one from Σ_1 and one from Σ_2, separated by a colon (:). So, for example, $\langle q_1, o, e, q_2\rangle$ is an element of δ.



Observe that each path in this device defines *two* strings: a concatenation of the left-hand-side labels of the arcs, and a concatenation of the right-hand-side labels. The upper path of the above transducer thus defines the pair *goose:geese*, whereas the lower path defines the pair *sheep:sheep*.

What constitutes a *computation* with a transducer? Similarly to the case of automata, a computation amounts to "walking" a path of the transducer, starting from the initial state and ending in some final state. Along the path, edges bear bi-symbol labels: one can view the left-hand-side symbol as an "input" symbol and the right-hand-side symbol as an "output" symbol. Thus, each path of the transducer defines a pair of strings, an input string (over Σ_1) and an output string (over Σ_2). This pair of strings is a member of the relation defined by the transducer.

DEFINITION 9. *Let* $T = \langle Q, q_0, \Sigma_1, \Sigma_2, \delta, F\rangle$ *be a finite state transducer. Define* $\hat{\delta} \subseteq Q \times \Sigma_1^* \times \Sigma_2^* \times Q$ *as follows:*

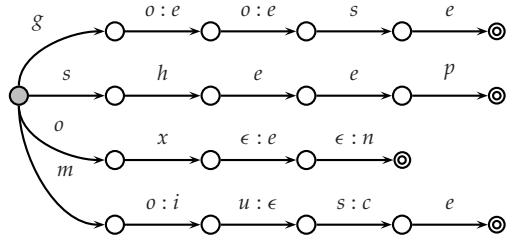
- *for each* $q \in Q$, $\hat{\delta}(q, \epsilon, \epsilon, q)$;
- *if* $\hat{\delta}(q_1, w_1, w_2, q_2)$ *and* $\delta(q_2, a, b, q_3)$, *then* $\hat{\delta}(q_1, w_1 \cdot a, w_2 \cdot b, q_3)$.

Then a pair $\langle w_1, w_2\rangle$ *is* ***accepted*** *(or* ***generated****) by T if and only if* $\hat{\delta}(q_0, w_1, w_2, w_f)$ *holds for some final state* $q_f \in F$. *The* ***relation defined by the transducer*** *is the set of all the pairs it accepts.*

As a shorthand notation, when an edge is labeled by two identical symbols, we depict only one of them and omit the colon.

The above definition of finite state transducers is not very useful: since each arc is labeled by exactly one symbol of Σ_1 and exactly one symbol of Σ_2, any relation that is implemented by such a transducer must relate only strings of exactly the same length. This should not be the case, and to overcome this limitation we extend the definition of δ to allow also ϵ-labels. In the extended definition, δ is a relation over Q, $\Sigma_1 \cup \{\epsilon\}$, $\Sigma_2 \cup \{\epsilon\}$ and Q. Thus a transition from one state to another can involve "reading" a symbol of Σ_1 without "writing" any symbol of Σ_2, or the other way round.

Example 23 (Finite state transducer with ϵ-labels). With the extended definition of transducers, we depict below an expanded transducer for singular–plural noun pairs in English.



Note that ϵ-labels can occur on the left or on the right of the ':' separator. The pairs accepted by this transducer are *goose:geese, sheep:sheep, ox:oxen*, and *mouse:mice*.

4.9 Properties of regular relations

The extension of automata to transducers carries with it some interesting results. First and foremost, finite state transducers define exactly the set of regular relations. Many of the closure properties of automata are valid for transducers, but some are not. As these properties bear not only theoretical but also practical significance, we discuss them in more detail in this section.

Given some transducer T, consider what happens when the labels on the arcs of T are modified such that only the left-hand symbol remains. In other words,

consider what is obtained when the transition relation δ is projected on three of its coordinates: Q, Σ_1, and Q only, ignoring the Σ_2 coordinate. It is easy to see that a finite state automaton is obtained. We call this automaton the *projection* of T to Σ_1. In the same way, we can define the projection of T to Σ_2 by ignoring Σ_1 in the transition relation. Since both projections yield finite state automata, they induce regular languages. Therefore the relation defined by T is a regular relation.

We can now consider certain operations on regular relations, inspired by similar operations on regular languages. For example, *union* is very easy to define. Recall that a regular relation is a subset of the Cartesian product of $\Sigma_1^* \times \Sigma_2^*$, that is, a set of pairs. If R_1 and R_2 are regular relations, then $R_1 \cup R_2$ is well defined, and it is straightforward to show that it is a regular relation. To define the union operation directly over transducers, extend the construction of FSA delineated in Section 4.5, namely add a new initial state with two edges labeled $\epsilon:\epsilon$ leading from it to the initial states of the given transducers. In a similar way, *concatenation* can be extended to regular relations: if R_1 and R_2 are regular relations then $R_1 \cdot R_2 = \{\langle w_1 \cdot w_2, w_3 \cdot w_4\rangle \mid \langle w_1, w_3\rangle \in R_1 \textit{ and } \langle w_2, w_4\rangle \in R_2\}$. Again, the construction for FSA can be straightforwardly extended to the case of transducers, and it is easy to show that $R_1 \cdot R_2$ is a regular relation.

Example 24 (Operations on finite state transducers). Let R_1 be the following relation, mapping some English words to their German counterparts: $R_1 = \{$*tomato:Tomate, cucumber:Gurke, grapefruit:Grapefruit, pineapple:Ananas, coconut:Koko*$\}$. Let R_2 be a similar relation: $R_2 = \{$*grapefruit:Pampelmuse, coconut:Kokusnuß*$\}$. Then: $R_1 \cup R_2 = \{$*tomato:Tomate, cucumber:Gurke, grapefruit:Grapefruit, grapefruit: Pampelmuse, pineapple:Ananas, coconut:Koko, coconut:Kokusnuß*$\}$.

A rather surprising fact is that regular relations are *not* closed under *intersection*. In other words, if R_1 and R_2 are two regular relations, then it very well might be the case that $R_1 \cap R_1$ is not a regular relation. It will take us beyond the scope of the material covered so far to explain this fact, but it is important to remember it when dealing with finite state transducers. For this reason exactly it follows that the class of regular relations is not closed under *complementation*: since intersection can be expressed in terms of union and complementation, if regular relations were closed under complementation they would have been closed also under intersection, which we know is not the case.

A very useful operation that is defined for transducers is *composition*. Intuitively, a transducer relates one word ("input") with another ("output"). When we have more than one transducer, we can view the output of the first transducer as the input to the second. The composition of T_1 and T_2 relates the input language of T_1 with the output language of T_2, bypassing the intermediate level (which is the output of T_1 and the input of T_2).

DEFINITION 10. *If R_1 is a relation from Σ_1^* to Σ_2^* and R_2 is a relation from Σ_2^* to Σ_3^* then the* ***composition*** *of R_1 and R_2, denoted $R_1 \circ R_2$, is a relation from Σ_1^* to Σ_3^* defined as $\{\langle w_1, w_3\rangle \mid$ there exists a string $w_2 \in \Sigma_2^*$ such that $w_1 R_1 w_2$ and $w_2 R_2 w_3\}$.*

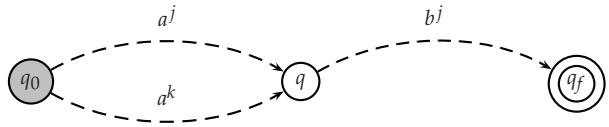
Example 25 (Composition of finite state transducers). Let R_1 be the following relation, mapping some English words to their German counterparts: $R_1 = \{$*tomato*: *Tomate, cucumber:Gurke, grapefruit:Grapefruit, grapefruit:Pampelmuse, pineapple:Ananas, coconut:Koko, coconut:Kokusnuß*$\}$. Let R_2 be a similar relation, mapping French words to their English translations: $R_2 = \{$*tomate:tomato, ananas*: *pineapple, pamplemousse:grapefruit, concombre:cucumber, cornichon: cucumber, noix-de-coco:coconut*$\}$. Then $R_2 \circ R_1$ is a relation mapping French words to their German translations (the English translations are used to compute the mapping, but are not part of the final relation): $R_2 \circ R_1 = \{$*tomate:Tomate, ananas:Ananas, pamplemousse:Grapefruit, pamplemousse: Pampelmuse, concombre:Gurke, cornichon:Gurke, noix-de-coco:Koko, noix-de-coco:Kokusnuße*$\}$.

5 Context-Free Languages

5.1 Where regular languages fail

Regular languages and relations are useful for various applications of natural language processing, but there is a limit to what can be achieved with such means. We mentioned in passing that not *all* languages over some alphabet Σ are regular; we now look at what kind of languages lie beyond the regular ones.

To exemplify a non-regular language, consider a simple language over the alphabet $\Sigma = \{a, b\}$ whose members are strings that consist of some number, n, of 'a's, followed by *the same* number of 'b's. Formally, this is the language $L = \{a^n \cdot b^n \mid n > 0\}$. Assume towards a contradiction that this language is regular, and therefore a deterministic finite state automaton A exists whose language is L. Consider the language $L_i = \{a^i \mid i > 0\}$. Since every string in this language is a prefix of some string $(a^i \cdot b^i)$ of L, there must be a path in A starting from the initial state for every string in L_i. Of course, there is an infinite number of strings in L_i, but by its very nature, A has a finite number of states. Therefore there must be two different strings in L_i that lead the automaton to a single state. In other words, there exist two strings, a^j and a^k, such that $j \neq k$ but $\hat{\delta}(q_0, a^j) = \hat{\delta}(q_0, a^k)$. Let us call this state q. There must be a path labeled b^j leading from q to some final state q_f, since the string $a^j b^j$ is in L. This situation is schematically depicted below (the dashed arrows represent paths):



Therefore, there is also an accepting path $a^k b^j$ in A, and hence also $a^k b^j$ is in L, in contradiction to our assumption. Hence no deterministic finite state automaton exists whose language is L.

We have seen one language, namely $L = \{a^n \cdot b^n \mid n > 0\}$, which cannot be defined by a finite state automaton and therefore is not regular. In fact, there are several other such languages, and there is a well-known technique, the so-called *pumping lemma*, for proving that certain languages are not regular. If a language is not regular, then it cannot be denoted by a regular expression. We must look for alternative means of specification for non-regular languages.

5.2 Grammars

In order to specify a class of more complex languages, we introduce the notion of a *grammar*. Intuitively, a grammar is a set of rules that manipulate symbols. We distinguish between two kinds of symbols: *terminal* ones, which should be thought of as elements of the target language, and *non-terminal* ones, which are auxiliary symbols that facilitate the specification. It might be instructive to think of the non-terminal symbols as *syntactic categories*, such as *Sentence, Noun Phrase*, or *Verb Phrase*. However, formally speaking, non-terminals have no "special," external interpretation where formal languages are concerned. Similarly, terminal symbols might correspond to letters of some natural language, or to words, or to something else: they are simply elements of some finite set.

Rules can express the internal structure of "phrases," which should not necessarily be viewed as natural language phrases. A rule is a non-empty sequence of symbols, a mixture of terminals and non-terminals, with the only requirement that the first element in the sequence be a non-terminal one (alternatively, one can define a rule as an ordered pair whose first element is a non-terminal symbol and whose second element is a sequence of symbols). We write such rules with a special symbol, '$\rightarrow$,' separating the distinguished leftmost non-terminal from the rest of the sequence. The leftmost non-terminal is sometimes referred to as the *head* of the rule, while the rest of the symbols are called the *body* of the rule.

Example 26 (Rules). Assume that the set of terminals is {*the, cat, in, hat*} and the set of non-terminals is {*D, N, P, NP, PP*}. Then possible rules over these two sets include:

$D \rightarrow the$	$NP \rightarrow D\,N$
$N \rightarrow cat$	$PP \rightarrow P\,NP$
$N \rightarrow hat$	$NP \rightarrow NP\,PP$
$P \rightarrow in$	

Note that the terminal symbols correspond to words of English, and not to letters as was the case above.

Consider the rule $NP \rightarrow D\,N$. If we interpret *NP* as the syntactic category *noun phrase*, *D* as *determiner*, and *N* as *noun*, then what the rule informally means is that one possible way to construct a noun phrase is by concatenating a determiner with a noun. More generally, a rule specifies one possible way to construct a "phrase" of

the category indicated by its head: this way is by concatenating phrases of the categories indicated by the elements in the body of the rule. Of course, there might be more than one way to construct a phrase of some category. For example, there are two rules which define the structure of the category *NP* in Example 26: either by concatenating a phrase of category *D* with one of category *N*, or by concatenating an *NP* with a *PP*.

In Example 26, rules are of two kinds: the ones on the left have a single terminal symbol in their body, while the ones on the right have one or more non-terminal symbols, but no rule mixes both terminal and non-terminal symbols in its body. While this is a common practice where grammars for natural languages are concerned, nothing in the formalism requires such a format for rules. Indeed, rules can mix any combination of terminal and non-terminal symbols in their bodies.

Formal language theory defines rules and grammars in a much broader way than that which was discussed above, and the definition below is actually only a special case of rules and grammars. For various reasons that have to do with the format of the rules, this special case is known as *context-free* rules. This has nothing to do with the ability of grammars to refer to context; the term should not be taken mnemonically. In the next section we discuss other rule-based systems. In this section, however, we use the terms *rule* and *context-free rule* interchangeably, as we do for grammars, derivations, etc.

DEFINITION 11. *A **context-free grammar** is a four-tuple $G = \langle V, \Sigma, P, S\rangle$, where V is a finite set of **non-terminal symbols**, Σ is an alphabet of **terminal symbols**, $P \subseteq V \times (V \cup \Sigma)^*$ is a set of **rules** and $S \in V$ is the **start symbol**.*

Note that this definition permits rules with empty bodies. Such rules, which consist of a left-hand-side only, are called ϵ-rules, and are useful both for formal and for natural languages. Example 33 below makes use of an ϵ-rule.

Example 27 (Grammar). The set of rules depicted in Example 26 can constitute the basis for a grammar $G = \langle V, \Sigma, P, S\rangle$, where $V = \{D, N, P, NP, PP\}$, $\Sigma = \{$*the, cat, in, hat*$\}$, P is the set of rules, and the start symbol S is *NP*.

In the sequel we depict grammars by listing their rules only, as we did in Example 26. We keep a convention of using uppercase letters for the non-terminals and lowercase letters for the terminals, and we assume that the set of terminals is the smallest that includes all the terminals mentioned in the rules, and the same for the non-terminals. Finally, we assume that the start symbol is the head of the first rule, unless stated otherwise.

5.3 Derivation

In order to define the language denoted by a grammar we need to define the concept of *derivation*. Derivation is a relation that holds between two *forms*, each a sequence of grammar symbols (terminal and/or non-terminal).

DEFINITION 12. *Let $G = \langle V, \Sigma, P, S\rangle$ be a grammar. The set of* ***forms*** *induced by G is $(V \cup \Sigma)^*$. A form α* ***immediately derives*** *a form β, denoted by $\alpha \Rightarrow \beta$, if and only if there exist $\gamma_l, \gamma_r \in (V \cup \Sigma)^*$ such that $\alpha = \gamma_l A \gamma_r$ and $\beta = \gamma_l \gamma_c \gamma_r$, and $A \to \gamma_c$ is a rule in P. A is called the* ***selected symbol****.*

A form α immediately derives β if a single non-terminal symbol, A, occurs in α, such that whatever is to its left in α, the (possibly empty) sequence of terminal and non-terminal symbols γ_l, occurs at the leftmost edge of β; and whatever is to the right of A in α, namely the (possibly empty) sequence of symbols γ_r, occurs at the rightmost edge of β; and the remainder of β, namely γ_c, constitutes the body of some grammar rule of which A is the head.

Example 28 (Immediate derivation). Let G be the grammar of Example 27. The set of forms induced by G contains all the (infinitely many) sequences of elements from V and Σ, such as $\langle\rangle$, $\langle NP\rangle$, $\langle D\ cat\ P\ D\ hat\rangle$, $\langle D\ N\rangle$, $\langle the\ cat\ in\ the\ hat\rangle$, etc.

Let us start with a simple form, $\langle NP\rangle$. Observe that it can be written as $\gamma_l NP\gamma_r$, where both γ_l and γ_r are empty. Observe also that NP is the head of some grammar rule: the rule $NP \to D\ N$. Therefore, the form is a good candidate for derivation: if we replace the selected symbol NP with the body of the rule, while preserving its environment, we obtain $\gamma_l D\ N\gamma_r = D\ N$. Therefore, $\langle N\rangle \Rightarrow \langle D\ N\rangle$.

We now apply the same process to $\langle D\ N\rangle$. This time the selected symbol is D (we could have selected N, of course). The left context is again empty, while the right context is $\gamma_r = N$. As there exists a grammar rule whose head is D, namely $D \to the$, we can replace the rule's head by its body, preserving the context, and obtain the form $\langle the\ N\rangle$. Hence $\langle D\ N\rangle \Rightarrow \langle the\ N\rangle$.

Given the form $\langle the\ N\rangle$, there is exactly one non-terminal that we can select, namely N. However, there are two rules that are headed by N: $N \to cat$ and $N \to hat$. We can select either of these rules to show that both $\langle the\ N\rangle \Rightarrow \langle the\ cat\rangle$ and $\langle the\ N\rangle \Rightarrow \langle the\ hat\rangle$.

Since the form $\langle the\ cat\rangle$ consists of terminal symbols only, no non-terminal can be selected and hence it derives no form.

We now extend the immediate derivation relation from a single step to an arbitrary number of steps by considering the reflexive transitive closure of the relation.

DEFINITION 13. *The* ***derivation*** *relation, denoted '$\overset{*}{\Rightarrow}$,' is defined recursively as follows: $\alpha \overset{*}{\Rightarrow} \beta$ if $\alpha = \beta$, or if $\alpha \Rightarrow \gamma$ and $\gamma \overset{*}{\Rightarrow} \beta$.*

Example 29 (Extended derivation). In Example 28 we showed that the following immediate derivations hold: $\langle NP\rangle \Rightarrow \langle D\ N\rangle$; $\langle D\ N\rangle \Rightarrow \langle the\ N\rangle$; $\langle the\ N\rangle \Rightarrow \langle the\ cat\rangle$. Therefore, $\langle NP\rangle \overset{*}{\Rightarrow} \langle the\ cat\rangle$.

The derivation relation is the basis for defining the language denoted by a grammar. Consider the form obtained by taking a single grammar symbol, say $\langle A\rangle$; if this form derives a sequence of terminals, this string is a member of the language denoted by A. The language of a grammar G, $L(G)$, is the language denoted by its start symbol.

DEFINITION 14. *Let $G = \langle V, \Sigma, P, S\rangle$ be a grammar. The* ***language of a non-terminal*** *$A \in V$ is*

$$L_G(A) = \{a_1 \cdots a_n \mid a_i \in \Sigma \text{ for } 1 \leq i \leq n \text{ and } \langle A\rangle \stackrel{*}{\Rightarrow} \langle a_1, \ldots, a_n\rangle\}$$

The ***language of the grammar*** *G is $L(G) = L_G(S)$.*

Example 30 (Language of a grammar). Consider again the grammar G of Example 27. It is fairly easy to see that the language denoted by the non-terminal symbol D, $L_G(D)$, is the singleton set $\{the\}$. Similarly, $L_G(P)$ is $\{in\}$ and $L_G(N) = \{cat, hat\}$. It is more difficult to define the languages denoted by the non-terminals *NP* and *PP*, although it should be straightforward that the latter is obtained by concatenating $\{in\}$ with the former. We claim without providing a proof that $L_G(NP)$ is the denotation of the regular expression $(the \cdot (cat + hat) \cdot (in \cdot the \cdot (cat + hat))^*)$.

5.4 Derivation trees

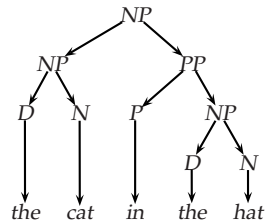
Sometimes two derivations of the same string differ only in the order in which they were applied. Consider again the grammar of Example 27. Starting with the form $\langle NP\rangle$ it is possible to derive the string *the cat* in two ways:

(1) $\langle NP\rangle \Rightarrow \langle D\, N\rangle \Rightarrow \langle D\, cat\rangle \Rightarrow \langle the\, cat\rangle$
(2) $\langle NP\rangle \Rightarrow \langle D\, N\rangle \Rightarrow \langle the\, N\rangle \Rightarrow \langle the\, cat\rangle$

Derivation (1) applies first the rule $N \to cat$ and then the rule $D \to the$ whereas derivation (2) applies the same rules in the reverse order. But since both use the same rules to derive the same string, it is sometimes useful to collapse such "equivalent" derivations into one. To this end the notion of *derivation trees* is introduced.

A derivation tree (sometimes called *parse* tree, or simply tree) is a visual aid in depicting derivations, and a means for imposing structure on a grammatical string. Trees consist of vertices and branches; a designated vertex, the *root* of the tree, is depicted on the top. Branches are connections between pairs of vertices. Intuitively, trees are depicted "upside down," since their root is at the top and their leaves are at the bottom. An example of a derivation tree for the string *the cat in the hat* with the grammar of Example 27 is given in Example 31.

Example 31 (Derivation tree).



Formally, a tree consists of a finite set of vertices and a finite set of branches (or arcs), each of which is an ordered pair of vertices. In addition, a tree has a designated vertex, the *root*, which has two properties: it is not the target of any arc, and every other vertex is accessible from it (by following one or more branches). When talking about trees we sometimes use family notation: if a vertex v has a branch leaving it which leads to some vertex u, then we say that v is the *mother* of u and u is the *daughter*, or *child*, of v. If u has two daughters, we refer to them as *sisters*. Derivation trees are defined with respect to some grammar G, and must obey the following conditions:

(1) every vertex has a *label*, which is either a terminal symbol, a non-terminal symbol, or ϵ;
(2) the label of the root is the start symbol;
(3) if a vertex v has an outgoing branch, its label must be a non-terminal symbol; furthermore, this symbol must be the head of some grammar rule; and the elements in the body of the same rule must be the labels of the children of v, in the same order;
(4) if a vertex is labeled ϵ, it is the only child of its mother.

A *leaf* is a vertex with no outgoing branches. A tree induces a natural "left-to-right" order on its leaves; when read from left to right, the sequence of leaves is called the *frontier*, or *yield*, of the tree.

Derivation trees correspond very closely to derivations. In fact, it is easy to show that a non-terminal symbol A derives a form α if and only if α is the yield of some parse tree whose root is A. In other words, whenever some string can be derived from a non-terminal, there exists a derivation tree for that string, with the same non-terminal as its root. However, sometimes there exist different derivations of the same string that correspond to a single tree. The tree representation collapses exactly those derivations that differ from each other only in the order in which rules are applied.

Sometimes, however, different derivations (of the same string!) correspond to different trees. This can happen only when the derivations differ in the rules which they apply. When more than one tree exists for some string, we say that the string is *ambiguous*. Ambiguity is a major problem when grammars are used for certain formal languages, in particular for programming languages. But for natural languages, ambiguity is unavoidable as it corresponds to properties of the natural language itself.

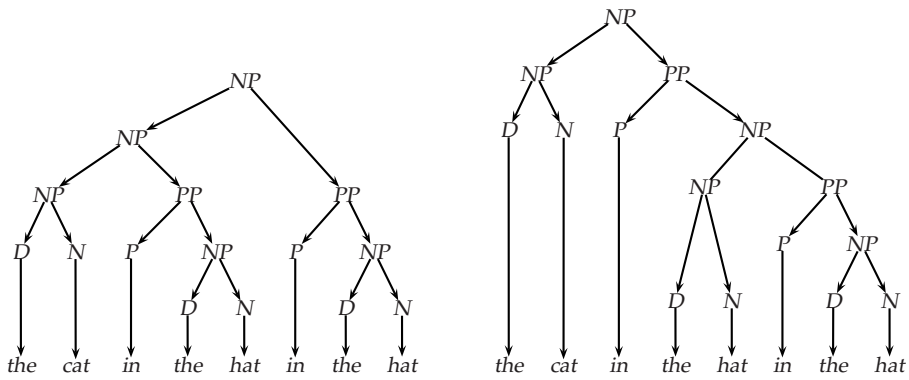
Example 32 (Ambiguity). Consider again the grammar of Example 27, and the string *the cat in the hat in the hat*. Intuitively, there can be (at least) two readings for this string: one in which a certain cat wears a hat-in-a-hat, and one in which a certain cat-in-a-hat is inside a hat. If we wanted to indicate the two readings with parentheses, we would distinguish between

((the cat in the hat) in the hat)

and

(the cat in (the hat in the hat))

This distinction in intuitive meaning is reflected in the grammar, and two different derivation trees, corresponding to the two readings, are available for this string:



Using linguistic terminology, in the left tree the second occurrence of the prepositional phrase *in the hat* modifies the noun phrase *the cat in the hat*, whereas in the right tree it only modifies the (first occurrence of) the noun phrase *the hat*. This situation is known as *syntactic* or *structural* ambiguity.

5.5 Expressiveness

Context-free grammars are more expressive than regular expressions. In Section 5.1 we claimed that the language $L = \{a^n b^n \mid n > 0\}$ is not regular; we now show a context-free grammar for this language. The grammar, $G = \langle V, \Sigma, P, S \rangle$, has two terminal symbols, $\Sigma = \{a, b\}$, and one non-terminal symbol, $V = \{S\}$. The idea is that whenever S is used recursively in a derivation (rule 1), the current form is extended by exactly one a on the left and one b on the right, hence the number of 'a's and 'b's must be equal.

Example 33 (A context-free grammar for $L = \{a^n b^n \mid n \geq 0\}$).

(1) $S \to a\, S\, b$
(2) $S \to \epsilon$

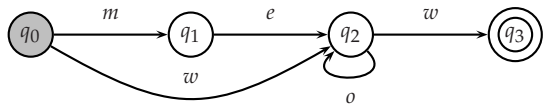
DEFINITION 15. *The class of languages that can be generated by context-free grammars is the class of* ***context-free languages****.*

The class of context-free languages properly contains the regular languages: given some finite state automaton which generates some language L, it is always possible to construct a context-free grammar whose language is L. We conclude this section with a discussion of converting automata to context-free grammars.

Let $A = \langle Q, q_0, \delta, F \rangle$ be a deterministic finite state automaton with no ϵ-moves over the alphabet Σ. The grammar we define to simulate A is $G = \langle V, \Sigma, P, S \rangle$,

where the alphabet Σ is that of the automaton, and where the set of non-terminals, V, is the set Q of the automaton states. The idea is that a single (immediate) derivation step with the grammar simulates a single arc traversal with the automaton. Since automata states are simulated by grammar non-terminals, it is reasonable to simulate the initial state by the start symbol, and hence the start symbol S is q_0. What is left, of course, are the grammar rules. These come in two varieties: first, for every automaton arc $\delta(q, a) = q'$ we stipulate a rule $q \to a\, q'$. Then, for every final state $q_f \in F$, we add the rule $q_f \to \epsilon$.

Example 34 (Simulating a finite state automaton by a grammar). Consider the automaton $\langle Q, q_0, \delta, F\rangle$ depicted below, where $Q = \{q_0, q_1, q_2, q_3\}$, $F = \{q_3\}$, and δ is $\{\langle q_0, m, q_1\rangle, \langle q_1, e, q_2\rangle, \langle q_2, o, q_2\rangle, \langle q_2, w, q_3\rangle, \langle q_0, w, q_2\rangle\}$:



The grammar $G = \langle V, \Sigma, P, S\rangle$ which simulates this automaton has $V = \{q_0, q_1, q_2, q_3\}$, $S = q_0$, and the set of rules:

(1) $q_0 \to m\, q_1$
(2) $q_1 \to e\, q_2$
(3) $q_2 \to o\, q_2$
(4) $q_2 \to w\, q_3$
(5) $q_0 \to w\, q_2$
(6) $q_3 \to \epsilon$

The string *meoow*, for example, is generated by the automaton by walking along the path $q_0 - q_1 - q_2 - q_2 - q_2 - q_3$. The same string is generated by the grammar with the derivation

$$\langle q_0\rangle \overset{1}{\Rightarrow} \langle mq_1\rangle \overset{2}{\Rightarrow} \langle meq_2\rangle \overset{3}{\Rightarrow} \langle meoq_2\rangle \overset{3}{\Rightarrow} \langle meooq_2\rangle \overset{4}{\Rightarrow} \langle meoowq_3\rangle \overset{6}{\Rightarrow} \langle meoow\rangle$$

Since every regular language is also a context-free language, and since we have shown a context-free language that is not regular, we conclude that the class of regular languages is properly contained within the class of context-free languages.

Observing the grammar of Example 34, a certain property of the rules stands out: the body of each of the rules either consists of a terminal followed by a non-terminal or is empty. This is a special case of what are known as *right-linear* grammars. In a right-linear grammar, the body of each rule consists of a (possibly empty) sequence of terminal symbols, optionally followed by a single non-terminal symbol. Most importantly, no rule exists whose body contains more than one non-terminal; and if a non-terminal occurs in the body, it is in the final position. Right-linear grammars are a restricted variant of context-free grammars, and it can be shown that they generate all and only the regular languages.

5.6 Formal properties of context-free languages

Context-free languages are more expressive than regular languages; this additional expressive power comes with a price: given an arbitrary context-free grammar G and some string w, determining whether $w \in L(G)$ takes time proportional to the cube of the length of w, $O(|w|^3)$ (in the worst case). In addition, context-free languages are not closed under some of the operations that the regular languages are closed under.

It should be fairly easy to see that context-free languages are closed under union. Given two context-free grammars $G_1 = \langle V_1, \Sigma_1, P_1, S_1 \rangle$ and $G_2 = \langle V_2, \Sigma_2, P_2, S_2 \rangle$, a grammar $G = \langle V, \Sigma, P, S \rangle$ whose language is $L(G_1) \cup L(G_2)$ can be constructed as follows: the alphabet Σ is the union of Σ_1 and Σ_2, the non-terminal set V is a union of V_1 and V_2, plus a new symbol S, which is the start symbol of G. Then, the rules of G are just the union of the rules of G_1 and G_2, with two additional rules: $S \to S_1$ and $S \to S_2$, where S_1 and S_2 are the start symbols of G_1 and G_2 respectively. Clearly, every derivation in G_1 can be simulated by a derivation in G using the same rules exactly, starting with the rule $S \to S_1$, and similarly for derivations in G_2. Also, since S is a new symbol, no other derivations in G are possible. Therefore $L(G) = L(G_1) \cup L(G_2)$.

A similar idea can be used to show that the context-free languages are closed under concatenation: here we only need one additional rule, namely $S \to S_1 S_2$, and the rest of the construction is identical. Any derivation in G will "first" derive a string of G_1 (through S_1) and then a string of G_2 (through S_2). To show closure under the Kleene-closure operation, use a similar construction with the added rules $S \to \epsilon$ and $S \to S S_1$.

However, it is possible to show that the class of context-free languages is not closed under intersection. That is, if L_1 and L_2 are context-free languages, then it is not guaranteed that $L_1 \cap L_2$ is context-free as well. From this fact it follows that context-free languages are not closed under complementation either. While context-free languages are not closed under intersection, they *are* closed under intersection with regular languages: if L is a context-free language and R is a regular language, then it is guaranteed that $L \cap R$ is context-free.

In the previous section we have shown a correspondence between two specification formalisms for regular languages: regular expressions and finite state automata. For context-free languages, we focused on a declarative formalism, namely context-free grammars, but they, too, can be specified using a computational model. This model is called *push-down automata*, and it consists of finite state automata augmented with unbounded memory in the form of a *stack*. Computations can use the stack to store and retrieve information: each transition can either push a symbol (taken from a special alphabet) onto the top of the stack, or pop one element off the top of the stack. A computation is successful if it ends in a final state with an empty stack. It can be shown that the class of languages defined by push-down automata is exactly the class of context-free languages.

5.7 Normal forms

The general definition of context-free grammars stipulates that the body of a rule may consist of any sequence of terminal and non-terminal symbols. However, it is possible to restrict the form of the rules without affecting the generative capacity of the formalism. Such restrictions are known as *normal forms* and are the topic of this section.

The best-known normal form is the Chomsky normal form (CNF): under this definition, rules are restricted to be of either of two forms. The body of any rule in a grammar may consist either of a single terminal symbol, or of exactly two non-terminal symbols (as a special case, empty bodies are also allowed). For example, the rules $D \rightarrow \textit{the}$ and $NP \rightarrow D\,N$ can be included in a CNF grammar, but the rule $S \rightarrow a\,S\,b$ cannot.

Unlike the right-linear grammars defined in Section 5.5, which can only generate regular languages, CNF grammars are equivalent in their weak generative capacity to general context-free grammars: it can be proven that for every context-free language L there exists a CNF grammar G such that $L = L(G)$. In other words, CNF grammars can generate all the context-free languages.

The utility of normal forms is in their simplicity. When some property of context-free languages has to be proven, it is sometimes much simpler to prove it for the restricted version of the formalism (e.g., for CNF grammars only), because the result can then extend to the entire class of languages. Similarly, processing normal-form grammars may be simpler than processing the general class of grammars. Thus, the first parsing algorithms for context-free grammars were limited to grammars in CNF. In natural language grammars, a normal form can embody the distinction between "real" grammar rules and the lexicon; a commonly used normal form defines grammar rules to have either a single terminal symbol or any sequence of zero or more non-terminal symbols in their body (notice that this is a relaxation of CNF).

6 The Chomsky Hierarchy

6.1 A hierarchy of language classes

We focus in this section on grammars as formalisms which denote languages. We have seen two types of grammars: context-free grammars, which generate the class of context-free languages; and right-linear grammars, which generate the class of regular languages. Right-linear grammars are a special case of context-free grammars, where additional constraints are imposed on the form of the rules. More generally, constraining the form of the rules can constrain the expressive power of the formalism. Similarly, more freedom in the form of the rules can extend the expressiveness of the formalism.

One way to achieve this is to allow more than a single non-terminal symbol in the *head* of the rules or, in other words, restrict the application of rules to a

specified *context*. In context-free grammars, a rule can be applied during a derivation whenever its head, A, is an element in a form. In the extended formalism such a derivation is allowed only if the context of A in the form, that is, A's neighbors to the right and left, are as specified in the rule. Due to this reference to context, this formalism is known as *context-sensitive* grammars. A rule in a context-sensitive grammar has the form $\alpha_1 A \alpha_2 \rightarrow \alpha_1 \beta \alpha_2$, where α_1, α_2, and β are all (possibly empty) sequences of terminal and non-terminal symbols. The other components of context-sensitive grammars are as in context-free grammars.

As usual, the class of languages that can be generated by context-sensitive grammars is called the *context-sensitive languages*. Considering that every context-free grammar is a special case of context-sensitive grammars (with an empty context), it should be clear that every context-free language is also context-sensitive or, in other words, that the context-free languages are contained in the set of the context-sensitive ones. As it turns out, this containment is proper, and there are context-sensitive languages that are not context-free.

This establishes a *hierarchy* of classes of languages: the regular languages are properly contained in the context-free languages, which are properly contained in the context-sensitive languages. These, in turn, are known to be properly contained in the set of languages generated by the so-called *unrestricted* or *general phrase-structure* grammars (this set is called the *recursively enumerable languages*). Each of the language classes in this hierarchy is associated with a computational model: FSA and push-down automata for the regular and context-free languages respectively; linear bounded Turing machines for the context-sensitive languages; and Turing machines for the recursively enumerable languages.

This hierarchy of language classes is called the *Chomsky hierarchy of languages*, and is schematically depicted in Figure 1.1.

6.2 The location of natural languages in the hierarchy

The Chomsky hierarchy of languages reflects a certain order of complexity: in some sense, the lower the language class is in the hierarchy, the simpler are its possible constructions. Furthermore, lower language classes allow for more efficient processing (in particular, the recognition problem is tractable for regular and context-free languages, but not for higher classes). If formal grammars are used to express the structure of *natural* languages, then we must know the location of these languages in the hierarchy.

Chomsky presents a theorem that says "English is not a regular language" (1957: 21); as for context-free languages, he says "I do not know whether or not English is itself literally outside the range of such analyses" (1957: 34). For many years, however, it was well accepted that natural languages were beyond the expressive power of context-free grammars. This was only proven in the 1980s, when two natural languages (Dutch and a dialect of Swiss German) were shown to be trans-context-free (that is, beyond the expressive power of context-free grammars). Still, the constructions in natural languages that necessitate more than

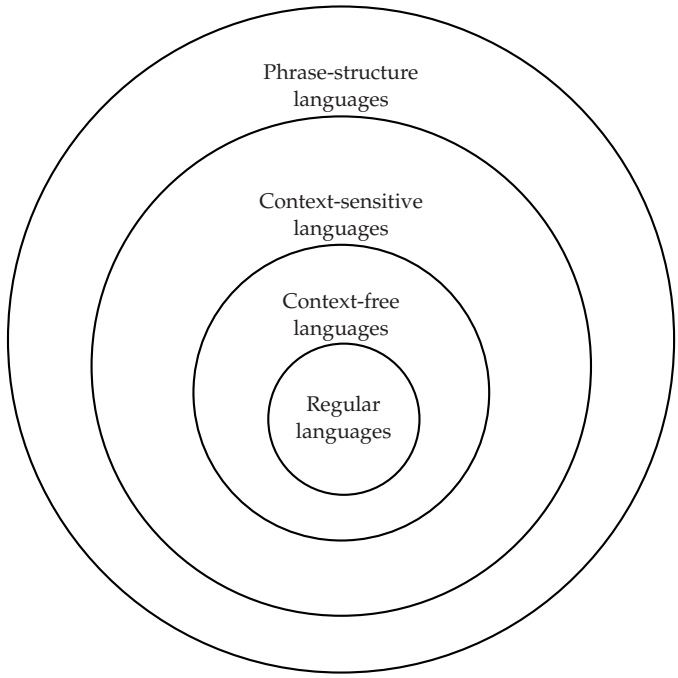


Figure 1.1 Chomsky's hierarchy of languages.

context-free power are few and very specific. (Most of these constructions boil down to patterns of the form $a^n b^m c^n d^m$, known as *cross-serial dependencies*; with some mathematical machinery, based mostly on closure properties of the context-free languages, it can be proven that languages that include such patterns cannot be context-free.) This motivated the definition of the class of *mildly context-sensitive languages*, which we discuss in Section 7.

6.3 Weak and strong generative capacity

So far we have only looked at grammars as generating sets of strings (i.e., languages), and ignored the structures that grammars impose on the strings in their languages. In other words, when we say that English is not a regular language we mean that no regular expression exists whose denotation is the set of all and only the sentences of English. Similarly, when a claim is made that some natural language, say Dutch, is not context-free, it should be read as saying that no context-free grammar exists whose language is Dutch. Such claims are propositions about the *weak generative capacity* of the formalisms involved: the weak generative capacity of regular expressions is insufficient for generating English; the weak generative capacity of context-free languages is insufficient for Dutch. Where natural languages are concerned, however, weak generative capacity might

not correctly characterize the relationship between a formalism (such as regular expressions or context-free grammars) and a language (such as English or Dutch). This is because one expects the formalism not only to be able to generate the strings in a language, but also to assign them "correct" structures.

In the case of context-free grammars, the structure assigned to strings is a derivation tree. Other linguistic formalisms may assign other kinds of objects to their sentences. We say that the *strong generative capacity* of some formalism is sufficient to generate some language if the formalism can (weakly) generate all the strings in the language, and also to assign them the "correct" structures. Unlike weak generative capacity, which is a properly defined mathematical notion, strong generative capacity is poorly defined, because no accepted definition of the "correct" structure for some string in some language exists.

7 Mildly Context-Sensitive Languages

When it was finally proven that context-free grammars are not even weakly adequate as models of natural languages, research focused on "mild" extensions of the class of context-free languages. In a seminal work, Joshi (1985) coined the term *mildly context-sensitive languages*, which is loosely defined as a class of languages that:

(1) properly contains all the context-free languages;
(2) can be parsed in polynomial time;
(3) can properly account for the constructions in natural languages that context-free languages fail to account for, such as cross-serial dependencies; and
(4) has the linear-growth property (this is a formal property that we ignore here).

One formalism that complies with these specifications (and which motivated their design) is *tree adjoining grammars* (TAGs). Motivated by linguistic considerations, TAGs extend the scope of locality in which linguistic constraints can be expressed. The elementary building blocks of the formalism are trees. Whereas context-free grammar rules enable one to express constraints among the mother in a local tree and its immediate daughters, the elementary trees of TAG facilitate the expression of constraints between arbitrarily distant nodes, as long as they are part of the same elementary tree. Two operations, *adjunction* and *substitution*, construct larger trees from smaller ones, so that the basic operations that take place during derivations are not limited to string concatenation. Crucially, these operations facilitate nesting of one tree within another, resulting in extended expressiveness.

The class of languages generated by tree adjoining grammars is naturally called the *tree adjoining languages*. It contains the context-free languages, and several trans-context-free ones, such as the language $\{a^n b^m c^n d^m \mid n, m \geq 0\}$. As usual, the added expressiveness comes with a price, and determining membership of a string w in a language generated by some TAG can only be done in time proportional to $|w|^6$.

Several linguistic formalisms were proposed as adequate for expressing the class of natural languages. Noteworthy among them are three formalisms: *head grammars*, *linear indexed grammars*, and *combinatory categorial grammars*. All three were developed independently with natural languages as their main motivation; and all three were proven to be (weakly) equivalent to TAG. The class of tree adjoining languages, therefore, may be just the correct formal class in which all natural languages reside.

8 Further Reading

Much of the material presented in this chapter can be found in introductory textbooks on formal language theory. Hopcroft and Ullman (1979, chapter 1) provide a formal presentation of formal language theory; just as rigorous, but with an eye to linguistic uses and applications, is the presentation of Partee et al. (1990, chapters 1–3). For the ultimate reference, consult the *Handbook of Formal Languages* (Rozenberg & Salomaa 1997).

A very good formal exposition of regular languages and the computing machinery associated with them is given by Hopcroft and Ullman (1979, chapters 2–3). Another useful source is Partee et al. (1990, chapter 17). Theorem 1 is due to Kleene (1956); Theorem 2 is due to Rabbin and Scott (1959); Theorem 3 is a corollary of the Myhil–Nerode theorem (Nerode 1958). The pumping lemma for regular languages is due to Bar-Hillel et al. (1961).

For natural language applications of finite state technology refer to Roche and Schabes (1997a), which is a collection of papers ranging from mathematical properties of finite state machinery to linguistic modeling using them. The introduction (Roche & Schabes 1997b) can be particularly useful, as will be Karttunen (1991). Kaplan and Kay (1994) is a classic work that sets the very basics of finite state phonology, referring to automata, transducers, and two-level rules. As an example of an extended regular expression language, with an abundance of applications to natural language processing, see Beesley and Karttunen (2003). Finally, Karttunen et al. (1996) is a fairly easy paper that relates regular expressions and relations to finite automata and transducers, and exemplifies their use in several language engineering applications.

Context-free grammars and languages are discussed by Hopcroft and Ullman (1979, chapters 4, 6) and Partee et al. (1990, chapter 18). The correspondence between regular languages and right-linear grammars is due to Chomsky and Miller (1958). A cubic-time parsing algorithm for context-free languages was first proposed by Kasami (1965); see also Younger (1967). Push-down automata were introduced by Oettinger (1961); see also Schützenberger (1963). Chomsky (1962) proved that they were equivalent to context-free grammars.

A linguistic formalism that is based on the ability of context-free grammars to provide adequate analyses for natural languages is generalized phrase-structure grammars, or GPSGs (Gazdar et al., 1985).

The Chomsky hierarchy of languages is due to Chomsky (1956, 1959). The location of the natural languages in this hierarchy is discussed in several papers, of which the most readable, enlightening, and amusing is Pullum and Gazdar (1982). Several other works discussing the non-context-freeness of natural languages are collected in Part III of Savitch et al. (1987). Rounds et al. (1987) inquire into the relations between formal language theory and linguistic theory, in particular referring to the distinction between weak and strong generative capacity. Works showing that natural languages cannot be described by context-free grammars include Bresnan et al. (1982) (Dutch), Shieber (1985) (Swiss German), and Manaster-Ramer (1987) (Dutch). Miller (1999) is dedicated to generative capacity of linguistic formalisms, where strong generative capacity is defined as the model theoretic semantics of a formalism.

Tree adjoining grammars were introduced by Joshi et al. (1975) and are discussed in several subsequent papers Joshi (1985; 1987; 2003). A polynomial-time parsing algorithm for TAG is given by Vijay-Shanker and Weir (1993) and Satta (1994). The three formalisms that are equivalent to TAG are head grammars (Pollard 1984), linear-indexed grammars (Gazdar 1988), and combinatory categorial grammars (Steedman 2000); they were proven equivalent by Vijay-Shanker and Weir (1994).

2 Computational Complexity in Natural Language

IAN PRATT-HARTMANN

We have become so used to viewing natural language in computational terms that we need occasionally to remind ourselves of the methodological commitment this view entails. That commitment is this: we assume that to understand linguistic tasks – tasks such as recognizing sentences, determining their structure, extracting their meaning, and manipulating the information they contain – is to discover the algorithms required to perform those tasks, and to investigate their computational properties. To be sure, the physical realization of the corresponding processes in humans is a legitimate study too, but one from which the computational investigation of language may be pursued in splendid isolation. Complexity theory is the mathematical study of the resources – both in time and space – required to perform computational tasks. What bounds can we place – from above or below – on the number of steps taken to compute such-and-such a function, or a function belonging to such-and-such a class? What bounds can we place on the amount of memory required? It is therefore not surprising that, in the study of natural language, complexity-theoretic issues abound.

Since *any* computational task can be the object of complexity-theoretic investigation, it would be hopeless even to attempt a complete survey of complexity theory in the study of natural language. We focus therefore on a selection of topics in natural language where there has been a particular accumulation of complexity-theoretic results. Section 2 discusses parsing and recognition; Section 3 discusses the computation of logical form; and Section 4 discusses the problem of determining logical relationships between sentences in natural language. But we begin with a brief review of complexity theory itself.

1 A Brief Review of Complexity Theory

Any account of complexity theory rests on some model of computation. The most widely used such model is the multi-tape Turing machine; and that is the model we use here. Throughout this chapter, we employ standard notation for strings: if

The Handbook of Computational Linguistics and Natural Language Processing, First Edition.
Edited by Alexander Clark, Chris Fox and Shalom Lappin.

Σ is an alphabet (a finite, non-empty set of symbols), Σ^* denotes the set of *strings* (finite sequences of elements) over Σ. The length of any string σ is denoted $|\sigma|$; the empty (zero-length) string is denoted ϵ; and the concatenation of strings σ and τ is denoted $\sigma\tau$. We follow standard practice in ignoring the difference between elements of Σ and the corresponding one-element strings.

1.1 Turing machines and models of computation

Informally, a *multi-tape Turing machine* comprises a finite number of *tapes*, a finite set of *states*, and an *instruction table*. The tapes may be thought of as the machine's memory, the states as the line numbers of its program, and the instruction table as the instructions of that program. The tapes are numbered consecutively from 1 to (say) $K \geq 2$; Tape 1 is referred to as the *input tape* and Tape K as the *output tape*; all other tapes are *work-tapes* (Figure 2.1). Each tape consists of a one-way infinite sequence of squares (i.e., there is a leftmost square, but no rightmost square), and is scanned by its own *tape-head*, which is always located over one of these squares. Every square contains a unique symbol, which is either a member of some non-empty, finite set Σ, called the *alphabet* of the Turing machine, or one of the special symbols $\sqcup$ (read: 'blank') or $\triangleright$ (read: 'start').

The set of states, Q, is assumed to contain a pair of distinguished states: the *initial state* q_0 and the *halting state* q_1; otherwise, states have no internal structure. The instruction table of the Turing machine is a finite set T of quintuples

(1) $\langle p, \bar{s}, q, \bar{t}, \bar{d}\rangle$,

where p and q are states (i.e., elements of Q), $\bar{s} = (s_1, \ldots, s_K)$ and $\bar{t} = (t_1, \ldots, t_K)$ are K-tuples of symbols (i.e., elements of $\Sigma \cup \{\sqcup, \triangleright\}$), and $\bar{d} = (d_1, \ldots, d_K)$ is a K-tuple whose elements are the special tags `left`, `right`, and `stay`. Informally, the Turing machine interprets the instruction (1) as follows:

(2) If the current state is p, and, for each k ($1 \leq k \leq K$), the square currently being scanned on Tape k contains the symbol s_k, then set the new state to be q, and, for each k ($1 \leq k \leq K$) do the following: write t_k on the square currently being scanned on Tape k, and place Tape k's head either one square left, or one square right, or in its current location, as directed by d_k.

We can make Tape 1 a read-only tape by insisting that it is never altered (i.e., that $t_1 = s_1$); likewise, we can make Tape K a write-only tape by insisting that its head never moves to the left. The symbol $\triangleright$ is used to indicate the extreme left of a tape: we insist that, if any tape-head is over this symbol, it never receives an instruction to move left; moreover, $\triangleright$ is never written or overwritten. The halting state q_1 indicates that the computation is over, and we insist that no instruction can be executed in this state. (It is easy to specify these conditions formally.) Technically speaking, a Turing machine is simply a tuple $M = \langle K, \Sigma, Q, q_0, q_1, T\rangle$ conforming to the above specifications.

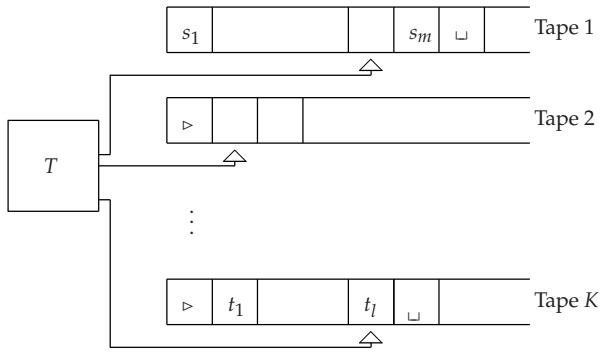


Figure 2.1 Architecture of a multi-tape Turing machine.

Turing machines perform *computations*, which proceed in discrete time-steps. At each time-step, the machine is in a specific *configuration*, consisting of its current state q, the position of the tape-head for each of the tapes, and the contents of each of the tapes. The initial configuration is as follows: the current state is q_0 (the initial state), with each tape-head positioned over the leftmost square of the tape; Tape 0 has the symbol ▷ in the leftmost square, followed by a string $\sigma \in \Sigma^*$, called the *input* of the computation, and is otherwise filled with ␣; all other tapes have the symbol ▷ in the leftmost square, and are otherwise filled with ␣. At each time-step, an instruction from T of the form (1) is executed as specified in (2), resulting in the next configuration. The computation halts when (and only when) no instruction in T can be executed. Note that, if the halting state q_1 is reached, the computation necessarily halts at that point. A *run* is a (finite or infinite) sequence of configurations obtained in this way; if the run is finite, so that the Turing machine halts, we call it a *terminating* run. Given a terminating run, the *output* of the computation is the string of Σ^* which, in the final configuration, is written on the output tape (strictly) between the ▷ and the first ␣. Notice that, in general, a Turing machine may be able to execute more than one instruction at any given time. In that case, we should think of the choice being made freely by the machine. We call a Turing machine *deterministic* just in case, for any state p and any K-tuple of symbols $\bar{s}$, T contains at most one instruction of the form (1) starting with the pair $\langle p, \bar{s} \rangle$ (i.e., the machine never has a choice as to which instruction to perform). A *non-deterministic Turing machine* is just another term for a Turing machine.

DEFINITION 1 (COMPUTABLE). *Let M be a deterministic Turing machine over alphabet Σ. For any string $\sigma \in \Sigma^*$, either M halts on input σ, or it does not. In the former case, M will output a definite string $\tau \in \Sigma^*$, and we can define the partial function $f_M : \Sigma^* \to \Sigma^*$ as follows.*

$$f_M(\sigma) = \begin{cases} \tau \text{ if } M \text{ halts on input } \sigma \\ \text{undefined otherwise} \end{cases}$$

We say that M ***computes*** *the function f_M. A partial function $f : \Sigma^* \to \Sigma^*$ is* ***Turing computable*** (*or just:* ***computable***) *if it is computed by some deterministic Turing machine.*

The instruction table of a Turing machine is fixed. Thus, a Turing machine is not a model of a computing machine in the sense we normally imagine, but rather of a computer program: there is only one thing it computes. On the other hand, since Turing machines are, formally, just tuples of finite objects, any Turing machine M can easily be coded as a string σ'_M over a suitable alphabet Σ', and that string can be input to another Turing machine, say M'. It can be shown that there exists a *universal Turing machine* U, which is able to simulate *any* Turing machine M over an alphabet Σ in the following sense: for any string, $\sigma \in \Sigma^*$, M has a non-terminating run on input σ if and only if U has a terminating run on input $\sigma'_M\sigma$; moreover, in case of termination, the output of M' is the same as the output of M. Any such Turing machine U *is* a model of a computing machine in the sense we normally imagine: it is able to execute an arbitrary 'program' σ'_M on arbitrary 'data' σ. Given such a coding scheme, consider the *halting function*, $H : (\Sigma')^* \to \{\top, \bot\}$ defined as

$$H(\sigma') = \begin{cases} \top \text{ if } \sigma' \text{ encodes a Turing machine } M \text{ that has a terminating run on input } \epsilon \\ \bot \text{ otherwise} \end{cases}$$

This function is clearly well defined, and indeed total. Perhaps the most fundamental fact in computability theory is due to Turing (1936–7):

THEOREM 1 (TURING). *The halting function is not computable.*

Definition 1 applies to functions $f : \Sigma^* \to \Sigma^*$ for any alphabet Σ. However, this definition can be extended to functions with other countable domains and ranges, relative to some coding of the relevant inputs and outputs as strings over an alphabet. Consider for instance the familiar coding of natural numbers as bit strings (elements of $\{0,1\}^*$). For $n \in \mathbb{N}$, denote by $\bar{n}$ the standard binary representation of n (without leading zeros); and for $s \in \{0,1\}^*$, denote by $\#s$ the natural number represented by s. If $f : \mathbb{N} \to \mathbb{N}$ is a function, we consider f computable if the function $g : \{0,1\}^* \to \{0,1\}^*$ defined by

$$g(s) = \overline{(f(\#s))}$$

is computable in the sense of Definition 1. Computability of functions with other domains and ranges – e.g., rational numbers, lists, graphs, etc. – is understood similarly. Technically, this extended notion of computability is relative to the coding scheme employed. In practice, however, all reasonable coding schemes usually yield the same computability (and complexity) results; if so, it is legitimate to speak of such functions as being computable or non-computable, leaving the operative coding scheme implicit.

The architecture of Turing machines given above is, in all essential details, that set out in Turing (1936–7). We have followed more recent practice in distinguishing input, output and work-tapes (Turing's machine had a single tape) to make it

a little easier to talk about space-bounded computations. But this makes no difference to any of the results reported here. The thesis that Turing computability captures our pre-theoretic notion of computability is generally referred to as the *Church–Turing thesis*. It is important to appreciate that this thesis does not rest on the existence of universal Turing machines, or indeed on any purely mathematical fact. Methodologically, the apparatus introduced above is an exercise in conceptual analysis: the proposed replacement of an informally understood notion with a rigorous definition. Historically, several competing analyses of computability were proposed at more or less the same time, most notably Gödel's notion of *recursive function* and Church's λ*-calculus*. All three notions in effect coincide, however; so there is general consensus about the formal model presented here. For an accessible modern treatment, see Papadimitriou (1994, Chapter 2).

The fundamental goal of complexity theory is to analyze the resources, in either time or space, required to perform computational tasks. The first step is to measure the computational resources required by particular algorithms.

DEFINITION 2. *Let M be a Turing machine with alphabet Σ, and let $g : \mathbb{N} \to \mathbb{N}$ be a function. We say M* ***runs in time*** *g if, for all but finitely many strings $\sigma \in \Sigma^*$, any run of M on input σ halts within at most $g(|\sigma|)$ steps. Similarly, M* ***runs in space*** *g if, for all but finitely many strings $\sigma \in \Sigma^*$, any run of M on input σ uses at most $g(|\sigma|)$ squares on any of its work-tapes.*

Allowing M to break the bound g in finitely many cases avoids problems caused by zero-length inputs and other trivial anomalies. Notice also the asymmetry in the definitions of time and space complexity: because measures of space complexity include only the *work-tapes* (and so exclude the input and output tapes), they can be *sublinear*. For time complexity, sublinear bounds make little sense, because they do not give the machine the opportunity to read its input.

Unfortunately, Definition 2 is too fragile to provide a meaningful measure of algorithmic complexity. Suppose M is a deterministic Turing machine computing some function in time g, and let c be a positive number. Provided g is moderately fast-growing (say, faster than linear growth), it is routine to construct another deterministic Turing machine M' – perhaps with more tapes or more states or a larger alphabet – that computes the same function in time $cg(n)$. That is: we can always speed up M by a linear factor! Since M and M' do not represent interestingly different algorithms, the statement that a Turing machine runs in time – say – $3n^2 + n + 4$ as opposed to $14n^2 + 87n + 11$ is, from an algorithmic point of view, not significant. Similar remarks also apply to space bounds.

DEFINITION 3. *Let M be a Turing machine, and G a set of functions from $\mathbb{N}$ to $\mathbb{N}$. We say that M* ***runs in time*** *G if, for some $g \in G$, M runs in time g. Similarly, we say that M* ***runs in space*** *G if, for some $g \in G$, M runs in space g.*

In particular, the following classes of functions suggest themselves.

DEFINITION 4 (*O*-NOTATION). *Let $g : \mathbb{N}^k \to \mathbb{N}$ be a function. Denote by $O(g)$ the set of functions*

$$O(g) = \{g' : \mathbb{N}^k \to \mathbb{N} \mid \textit{there exist } c \in \mathbb{N},\ n'_1, \ldots, n'_k \in \mathbb{N} \textit{ s.t.}$$
$$\textit{for all } n_1 > n'_1 \ldots \textit{for all } n_k > n'_k,\ g'(n_1, \ldots, n_k) \leq cg(n_1, \ldots, n_k)\}$$

Informally, $O(g)$ is the class of functions which are eventually dominated by some positive multiple of g. Combining Definitions 3 and 4, it makes sense to say, for example, that a given Turing machine runs in time (or space) $O(n^2)$, or $O(n^3)$, or $O(2^n)$. And this sort of complexity measure, it turns out, is *robust* under the expansions of computational resources considered above. For example, it can be shown that, for any $k > 0$, there is a function that can be computed by a deterministic Turing machine running in time $O(n^{k+1})$ which cannot be computed by any deterministic Turing machine running in time $O(n^k)$; and similarly for space bounds. (The precise statement of these theorems, known as *separation theorems*, is somewhat intricate; see Kozen, 2006, Lecture 3, or Papadimitriou, 1994: 143ff.) *O*-notation has the further advantage of permitting a useful degree of informality when analyzing the complexity of an algorithm, since a pseudo-code description of that algorithm, of the sort standardly found in computing texts, often suffices to show that it will run in time or space $O(g)$ (for some function g) without our having first to compile that description into a Turing machine. Finally, a word of caution. Knowing that a Turing machine (or algorithm) has time complexity $O(g)$ at best imposes a bound on how rapidly the cost of computation grows with the size of the input. That is, the complexity measures in question are *asymptotic*. In many cases, algorithms with suboptimal asymptotic complexity measures perform best in practice.

1.2 Decision problems

So far, we have discussed complexity measures for particular algorithms, understood as deterministic Turing machines. We now develop this idea in two crucial – though logically quite separate – ways.

The first development extends Definition 1 to *non-deterministic* computation. To do this, we first restrict attention to functions whose range contains just two elements – we conventionally employ $\top$ and $\bot$ – representing 'YES' and 'NO' respectively. A function $f : A \to \{\top, \bot\}$, where A is a countable set, is called a *decision problem*, or simply a *problem*. While decision problems may initially seem of limited practical interest, they play a central role in complexity theory. Moreover, the restriction to decision problems is less severe than might at first appear: the complexity of many functions can often be usefully characterized in terms of the complexity of closely related decision problems.

Now, any decision problem $f : A \to \{\top, \bot\}$ can alternatively be regarded as a *subset* of A – namely, the subset $\{a \in A \mid f(a) = \top\}$. In particular, if $A = \Sigma^*$ for some

alphabet Σ (or if the encoding of A in Σ^* is obvious), a decision problem defined on A is, in effect, a set of strings over Σ, or, in the parlance of formal language theory, a *language* over Σ. Conversely, of course, any language $L \subseteq \Sigma^*$ may be regarded as a decision problem $f : \Sigma^* \to \{\top, \bot\}$ given by:

$$f(\sigma) = \begin{cases} \top & \text{if } \sigma \in L \\ \bot & \text{otherwise} \end{cases}$$

The observation that decision problems and languages are essentially the same thing prompts the following definition.

DEFINITION 5. *Let M be a Turing machine over the alphabet Σ, and suppose without loss of generality that Σ contains the symbol $\top$. We say that M **accepts** a string $\sigma \in \Sigma^*$ if there exists a terminating run of M with input σ and output $\top$. The language $L \subseteq \Sigma^*$ **recognized by** M, denoted $L(M)$, is the set of strings accepted by M.*

It is important to bear in mind that, in Definition 5, M can be *non-deterministic*. That is: $L(M)$ is the set of inputs for which M *may* yield the output $\top$. (It is sometimes convenient to imagine a benign helper guiding M to make the 'right' choice of instructions required to accept a string $\sigma \in L$.) Equally important is that, if $\sigma \notin L$, there is no requirement for M to produce any particular output (as long as it is not $\top$, of course), or indeed to halt at all.

The case where M halts on every input is of particular interest, however:

DEFINITION 6 (DECIDABLE). *Let L be a language. We call L **decidable** if it is recognized by a Turing machine guaranteed to halt on every input.*

It is routine to show that any decidable language is in fact recognized by a *deterministic* Turing machine that halts on every input. Furthermore, that machine can easily be modified so as always to produce one of the two outputs $\top$, $\bot$. Thus, a decision problem $f : \Sigma^* \to \{\top, \bot\}$ is a computable function, in the sense of Definition 1, just in case the corresponding language $L = \{\sigma | f(\sigma) = \top\}$ is decidable, in the sense of Definition 6. Henceforth, then, we shall identify decision problems and languages, employing whichever term is most appropriate in context.

We may think of Definition 5 as a generalization of Definition 1 to the case of *non-deterministic* computation. The significance of this generalization is that, while deterministic and non-deterministic Turing machines recognize the same class of languages, they may not in general do so within the same computational bounds, a possibility which plays a central role in complexity theory.

We can generalize the above observations on linear speedup to the case of non-deterministic computation for decision problems. We give a reasonably precise version here:

THEOREM 2. *Let L be a language over some alphabet, let $g : \mathbb{N} \to \mathbb{N}$ and $h : \mathbb{N} \to \mathbb{N}$ be functions, let $c \geq 1$, and suppose $g(n) \geq n + 1$, and $h(n) \geq \log n$. If L is recognized by some Turing machine running in time $cg(n)$, then it is recognized by some Turing machine running in time $g(n)$. If L is recognized by some Turing machine running in*

space $ch(n)$, then it is recognized by some Turing machine running in space $h(n)$. The previous statements continue to hold when "Turing machine" is replaced throughout by "deterministic Turing machine."

Now for the second development in our analysis of complexity. So far, we have provided measures of the time and space requirements of particular *Turing machines* (or, by extension, and using O-notation, of particular *algorithms*). But what primarily interests us in complexity theory are the time and space requirements of a *maximally efficient* Turing machine for computing a particular *function* or, more specifically, solving a particular *decision problem*. Recalling the equivalence between decision problems and languages discussed above, we define:

DEFINITION 7. *Let L be a language over some alphabet, and let G be a set of functions from $\mathbb{N}$ to $\mathbb{N}$. We say that L is in TIME(G) (or SPACE(G)) if there exists a deterministic Turing machine M recognizing L, such that M runs in time (respectively, space) G.*

Classes of languages of the form TIME(G) or SPACE(G) are referred to as (deterministic) complexity classes. To avoid notational clutter, if g is a function from $\mathbb{N}$ to $\mathbb{N}$, we write TIME(g) instead of TIME($\{g\}$); and similarly for other complexity classes.

So far, we have encountered classes of functions of the form $O(g)$ for various g. When analyzing the complexity of *languages* (rather than of specific algorithms), however, larger classes of functions are typically more useful.

DEFINITION 8. *Let P, E, and E_k (for $k > 1$) be the sets of functions from $\mathbb{N}$ to $\mathbb{N}$ defined as follows:*

$$P = \left\{n^c \mid c > 0\right\}$$

$$E = \left\{2^{n^c} \mid c > 0\right\}$$

$$E_2 = \left\{2^{2^{n^c}} \mid c > 0\right\}$$

$$E_k = \left\{\underbrace{2^{2^{\cdots^{2}}}}_{k \text{ times}}{}^{n^c} \mid c > 0\right\}$$

*A function $g : \mathbb{N} \to \mathbb{N}$ which is in E_k for some k is said to be **elementary**.*

Non-elementary functions grow rapidly. However, it is easy to define a computable function which is non-elementary:

$$f(n) = \underbrace{2^{2^{\cdots^{2}}}}_{n \text{ times}}$$

Combining Definitions 7 and 8, we obtain complexity classes which are often known under the following, more pronounceable names:

$$\begin{array}{rr} & \mathrm{LOGSPACE} = \mathrm{SPACE}(\log n) \\ \mathrm{PTIME} = \mathrm{TIME}(P) & \mathrm{PSPACE} = \mathrm{SPACE}(P) \\ \mathrm{EXPTIME} = \mathrm{TIME}(E) & \mathrm{EXPSPACE} = \mathrm{SPACE}(E) \\ k\text{-EXPTIME} = \mathrm{TIME}(E_k) & k\text{-EXPSPACE} = \mathrm{SPACE}(E_k) \end{array}$$

Thus, PTIME is the class of languages recognizable by a deterministic Turing machine in polynomial time, EXPSPACE, the class of languages recognizable by a deterministic Turing machine in exponential space, and so on. In some texts, LOGSPACE is referred to as L, PTIME as P, and EXPTIME as EXP. Notice, incidentally, that there is no point in defining, say, $G = \{\log(n^c) \mid c > 0\}$ and then setting LOGSPACE = SPACE(G), since, by Theorem 2, linear factors may be ignored. Finally, if L is not recognizable by any Turing machine running in time bounded by an elementary function, then L is said to have *non-elementary complexity*. We shall encounter examples of decidable, but non-elementary, problems below.

Definition 7 may be adapted directly to deal with non-deterministic computation.

DEFINITION 9. *Let L be a language over some alphabet, and let G be a set of functions from $\mathbb{N}$ to $\mathbb{N}$. We say that L is in NTIME(G) (or NSPACE(G)) if there exists a Turing machine M recognizing L, such that M runs in time* (*respectively, space*) *G.*

Classes of languages of the form NTIME(G) or NSPACE(G) are referred to as (non-deterministic) complexity classes.

Combining Definitions 8 and 9, we obtain complexity classes which are often known under the following, more pronounceable names:

$$(3) \qquad \begin{array}{rr} & \mathrm{NLOGSPACE} = \mathrm{NSPACE}(\log n) \\ \mathrm{NPTIME} = \mathrm{NTIME}(P) & \mathrm{NPSPACE} = \mathrm{NSPACE}(P) \\ \mathrm{NEXPTIME} = \mathrm{NTIME}(E) & \mathrm{NEXPSPACE} = \mathrm{NSPACE}(E) \\ \mathrm{N}k\text{-EXPTIME} = \mathrm{NTIME}(E_k) & \mathrm{N}k\text{-EXPSPACE} = \mathrm{NSPACE}(E_k) \end{array}$$

In some texts, NLOGSPACE is referred to as NL, NPTIME as NP, and NEXPTIME as NEXP.

Notice the asymmetry involved in the notion of non-deterministic computation: M recognizes $L \subseteq \Sigma^*$ just in case, for each string $\sigma \in \Sigma^*$, $\sigma \in L$ if and only if *there exists* a successfully terminating run of M (i.e., a terminating run with output $\top$) on input σ – that is to say, $\sigma \in \Sigma^* \setminus L$ if and only if *all* runs of M on input σ fail to halt successfully. This asymmetry prompts us to define the *complement* classes as follows.

DEFINITION 10. *If $\mathcal{C}$ is a class of languages, then* Co-$\mathcal{C}$ *is the class of languages L such that $\Sigma^* \setminus L$ is in $\mathcal{C}$, where Σ is the alphabet of L.*

It is easy to see that, for any interesting class of functions G, TIME(G) = Co-TIME(G) and SPACE(G) = Co-SPACE(G). For this reason, we never speak of

Co-PTIME, Co-PSPACE, etc. The situation with non-deterministic complexity classes is different, however. It is not known whether NPTIME = Co-NPTIME; and similarly for many other classes of the form Co-NTIME(G). Indeed, such complexity classes are regularly encountered. In particular, putting together Definition 10, and the NTIME-classes listed in (3), we obtain the complexity classes Co-NPTIME, Co-NEXPTIME, and Co-Nk-EXPTIME. (And similarly for the corresponding space-complexity classes; but see Theorem 4.)

1.3 Relations between complexity classes

It is obvious from the above definitions that any language in TIME(G) (or SPACE(G)) is non-deterministically recognizable within the same bounds. Formally,

$$\text{TIME}(G) \subseteq \text{NTIME}(G) \qquad \text{SPACE}(G) \subseteq \text{NSPACE}(G)$$

A little less obviously, we see that:

$$\text{NPTIME} \subseteq \text{EXPTIME} \qquad \text{NEXPTIME} \subseteq \text{2-EXPTIME} \qquad \cdots$$

Consider the first of these inclusions. If M non-deterministically recognizes L, and p is a polynomial such that M is guaranteed to halt within time $p(n)$ on input of size n, the number of possible runs of M on inputs of this size is easily seen to be bounded by $2^{q(n)}$ for some polynomial q. But then a deterministic Turing machine M', simulating M, can check all of these runs in exponential time, outputting $\top$ if any one of them halts successfully. Hence, NPTIME $\subseteq$ EXPTIME. The inclusion NEXPTIME $\subseteq$ 2-EXPTIME follows analogously; and so on up the complexity hierarchy. In fact, similar arguments establish the following more elaborate system of inclusions.

$$\begin{aligned} &\text{PTIME} \subseteq \text{NPTIME} \subseteq \text{PSPACE} \subseteq \\ &\quad \text{EXPTIME} \subseteq \text{NEXPTIME} \subseteq \text{EXPSPACE} \subseteq \\ &\quad\quad \text{2-EXPTIME} \subseteq \text{2-NEXPTIME} \cdots \end{aligned} \tag{4}$$

The following result establishes that, for classes of sufficiently 'large' functions, non-determinism makes no difference to space complexity (Savitch 1970).

THEOREM 3 (SAVITCH). *If* $g(n) \geq \log n$, *then* $\text{NSPACE}(g(n)) \subseteq \text{SPACE}((g(n))^2)$

In some statements of this theorem, certain technical conditions are imposed on g; but see, e.g., Kozen (2006: 15–16). Since the classes of functions P, E, E_2, etc. are closed under squaring, we have NPSPACE = PSPACE, NEXPSPACE = EXPSPACE, and so on. As an instant corollary, since these deterministic classes are

equal to their complements, we have NPSPACE = Co-NPSPACE, NEXPSPACE = Co-NEXPSPACE, and so on.

Care is required when applying the reasoning of the previous paragraph. Setting $g(n) = \log n$, Theorem 3 tells us that NLOGSPACE $\subseteq$ SPACE($(\log n)^2$); however, this is not sufficient to imply that NLOGSPACE $\subseteq$ LOGPSPACE. Nevertheless, the following result establishes that equivalence under complementation continues to hold even in this case (Immerman 1988).

THEOREM 4 (IMMERMAN–SZELEPCSÉNYI). *If* $g(n) \geq \log n$, *then* NSPACE($g(n)$) = Co-NSPACE($g(n)$)

In some statements of this theorem, certain technical conditions are imposed on g; but again, see Kozen (2006: 22–4). As a special case, we have NSPACE(n) = Co-NSPACE(n), which settled a long-standing conjecture in formal language theory (see Section 2.3 below). As an instant corollary of Theorem 4, NLOGSPACE = Co-NLOGSPACE.

Adding these 'small' complexity classes to the inclusions (4), we obtain

$$\begin{aligned} &\text{LOGSPACE} \subseteq \text{NLOGSPACE} \subseteq \text{PTIME} \subseteq \text{NPTIME} \subseteq \\ &\quad \text{PSPACE} \subseteq \text{EXPTIME} \subseteq \text{NEXPTIME} \subseteq \\ &\qquad \text{EXPSPACE} \subseteq \text{2-EXPTIME} \subseteq \text{2-NEXPTIME} \cdots \end{aligned} \tag{5}$$

1.4 Lower bounds

Notwithstanding the above caveats on the interpretation of asymptotic complexity measures, saying that a language is in a complexity class $\mathcal{C}$ places some kind of *upper bound* on the resources required to recognize it. But what of *lower bounds*? What if we want to say that a language *cannot* be recognized within certain time or space bounds? For the complexity classes introduced above, useful lower-bound characterizations are indeed possible.

The basic idea is that of a *reduction* of one language (or decision problem) to another. Let L_1 and L_2 be languages, perhaps over different alphabets Σ_1 and Σ_2. Suppose that there exists a function $g : \Sigma_1^* \to \Sigma_2^*$ such that, for any string $\sigma \in \Sigma_1^*$, $\sigma \in L_1$ if and only if $g(\sigma) \in L_2$. We may think of g as a means of 'translating' L_1 into L_2: in particular, any Turing machine recognizing L_2 can be modified to recognize L_1 by simply prepending the translation g. If the cost of this translation is small, then we may regard L_2 as being 'at least as hard to recognize as' L_1.

DEFINITION 11 (REDUCTION). *Let* Σ_1 *and* Σ_2 *be alphabets, and let* L_i *be a language over* Σ_i ($i = 1, 2$). *A* ***reduction of*** L_1 ***to*** L_2 *is a function* $g : \Sigma_1^* \to \Sigma_2^*$, *such that* g *can be computed by a* (*deterministic*) *Turing machine in space* $O(\log n)$, *and for all* $\sigma \in \Sigma_1^*$, $\sigma \in L_1$ *if and only if* $g(\sigma) \in L_2$; *in that case, we say that* L_1 *is* ***reducible to*** L_2. *If, instead,* g *can merely be computed in time* $O(n^k)$ *for some* k, *we call it a* ***polynomial reduction***, *and we say that* L_1 *is* ***polynomially reducible to*** L_2.

Let $\mathcal{C}$ be any of the complexity classes mentioned in (5), or the complement of any of these classes. It can be shown that, if L_2 is in $\mathcal{C}$, and L_1 is reducible to L_2, then L_1 is in $\mathcal{C}$. We say that $\mathcal{C}$ is 'closed under reductions'. If $\mathcal{C}$ is any of the complexity classes mentioned in (4), then $\mathcal{C}$ is, similarly, 'closed under polynomial reductions.'

THEOREM 5. *The relation of reducibility is transitive: if L_1 is reducible to L_2, and L_2 to L_3, then L_1 is reducible to L_3.*

We remark that Theorem 5 is not obvious (though its analogue in the case of *polynomial* reducibility is) see, e.g., Papadimitriou (1994: 164).

Now we can give our characterization of lower complexity bounds.

DEFINITION 12 (HARDNESS AND COMPLETENESS). *Let $\mathcal{C}$ be a complexity class. A language L is said to be* ***hard for*** *$\mathcal{C}$, or* ***$\mathcal{C}$-hard****, if any language in $\mathcal{C}$ is reducible to L; L is said to be* ***complete for*** *$\mathcal{C}$, or* ***$\mathcal{C}$-complete****, if L is $\mathcal{C}$-hard and also in $\mathcal{C}$. Additionally, L is said to be* ***$\mathcal{C}$-hard under polynomial reduction*** *if any decision problem in $\mathcal{C}$ is polynomially reducible to L; similarly for* ***$\mathcal{C}$-completeness under polynomial reduction****.*

It follows from Theorem 5 that, if L_1 is $\mathcal{C}$-hard for some complexity class $\mathcal{C}$, and L_1 is reducible to L_2, then L_2 is $\mathcal{C}$-hard. Similarly, *mutatis mutandis*, for hardness under polynomial reductions. Notice that the notion of LOGSPACE-completeness is uninteresting: any problem in LOGSPACE is by definition LOGSPACE-complete. Under polynomial reductions, the notion of PTIME-completeness is similarly uninteresting. Definition 12 reflects the fact that reducibility in logarithmic space is taken to be the default in complexity theory. However, for most higher complexity classes, it is generally easier and just as informative to work with reducibility in polynomial time; and this is what is often done in practice. Hardness results, in the sense of Definition 12, are sometimes referred to, for obvious reasons, as 'lower complexity bounds.' However, it is important not to be misled by this terminology: for example, it is easy to show that there are PTIME-hard problems in TIME(n); but TIME(n) is properly contained in PTIME!

Many natural problems (it is easier here to speak of problems rather than languages) can be shown to be complete for the complexity classes introduced above. Here are three very well-known examples. In the context of propositional logic, a *literal* is a proposition letter or a negated proposition letter; proposition letters are said to be *positive* literals, their negations *negative* literals. A *clause* is a disjunction of literals; a clause is said to be *Horn* if it contains at most one positive literal. Theorems 6–9 are among the most fundamental in complexity theory. For an accessible treatment, see, e.g., Papadimitriou (1994: 171, 176, and 398 respectively). Theorem 6 is due to Cook (1971).

THEOREM 6 (COOK). *The problem of determining whether a given set of clauses is satisfiable is NPTIME-complete.*

THEOREM 7. *The problem of determining whether a given set of Horn clauses is satisfiable is PTIME-complete.*