Handbook of Natural Language Processing

Handbook of Natural Language Processing

edited by

Robert Dale
Macquarie University
Sydney, Australia

Hermann Moisl
University of Newcastle
Newcastle-upon-Tyne, England

Harold Somers
UMIST
Manchester, England

MARCEL DEKKER NEW YORK

Library of Congress Cataloging-in-Publication Data

Handbook of natural language processing/edited by Robert Dale, Hermann Moisl, Harold Somers.
p. cm.
ISBN: 0-8247-9000-6 (alk. paper)
1. Natural language processing (Computer science)—Handbooks, manuals, etc. I. Dale, Robert. Moisl, Hermann. III. Somers, H. L.
QA76.9.N38 H363 2000
006.3′5—dc21 00-031597

This book is printed on acid-free paper.

Headquarters
Marcel Dekker
270 Madison Avenue, New York, NY 10016
tel: 212-696-9000; fax: 212-685-4540

Eastern Hemisphere Distribution
Marcel Dekker AG
Hutgasse 4, Postfach 812, CH-4001 Basel, Switzerland
tel: 41-61-261-8482; fax: 41-61-261-8896

World Wide Web
http://www.dekker.com

The publisher offers discounts on this book when ordered in bulk quantities. For more information, write to Special Sales/Professional Marketing at the headquarters address above.

Current printing (last digit)
10 9 8 7 6 5 4

PRINTED IN THE UNITED STATES OF AMERICA

Preface

The discipline of Natural Language Processing (NLP) concerns itself with the design and implementation of computational machinery that communicates with humans using natural language. Why is it important to pursue such an endeavor? Given the self-evident observation that humans communicate most easily and effectively with one another using natural language, it follows that, in principle, it is the easiest and most effective way for humans and machines to interact; as technology proliferates around us, that interaction will be increasingly important. At its most ambitious, NLP research aims to design the language input–output components of artificially intelligent systems that are capable of using language as fluently and flexibly as humans do. The robots of science fiction are archetypical: stronger and more intelligent than their creators and having access to vastly greater knowledge, but human in their mastery of language. Even the most ardent exponent of artificial intelligence research would have to admit that the likes of HAL in Kubrick's *2001: A Space Odyssey* remain firmly in the realms of science fiction. Some success, however, has been achieved in less ambitious domains, where the research problems are more precisely definable and therefore more tractable. Machine translation is such a domain, and one that finds ready application in the internationalism of contemporary economic, political, and cultural life. Other successful areas of application are message-understanding systems, which extract useful elements of the propositional content of textual data sources such as newswire reports or banking telexes, and front ends for information systems such as databases, in which queries can be framed and replies given in natural language. This handbook is about the design of these and other sorts of NLP systems. Throughout, the emphasis is on practical tools and techniques for implementable systems; speculative research is minimized, and polemic excluded.

The rest of this preface is in three sections. In Section 1 we provide a sketch of the development of NLP, and of its relationship with allied disciplines such as linguistics and cognitive science. Then, in Section 2, we go on to delineate the scope of the handbook in terms of the range of topics covered. Finally, in Section 3, we provide an overview of how the handbook's content is structured.

1. THE DEVELOPMENT OF NATURAL LANGUAGE PROCESSING

A. History

In the 1930s and 1940s, mathematical logicians formalized the intuitive notion of an effective procedure as a way of determining the class of functions that can be computed algorithmically. A variety of formalisms were proposed—recursive functions, lambda calculus, rewrite systems, automata, artificial neural networks—all equivalent in terms of the functions they can compute. The development of these formalisms was to have profound effects far beyond mathematics. Most importantly, it led to modern computer science and to the computer technology that has transformed our world in so many ways. But it also generated a range of new research disciplines that applied computational ideas to the study and simulation of human cognitive functions: cognitive science, generative linguistics, computational linguistics, artificial intelligence, and natural language processing. Since the 1950s, three main approaches to natural language processing have emerged. We consider them briefly in turn.

- Much of the NLP work in the first 30 years or so of the field's development can be characterized as "NLP based on generative linguistics." Ideas from linguistic theory, particularly relative to syntactic description along the lines proposed by Chomsky and others, were absorbed by researchers in the field, and refashioned in various ways to play a role in working computational systems; in many ways, research in linguistics and the philosophy of language set the agenda for explorations in NLP. Work in syntactic description has always been the most thoroughly detailed and worked-out aspect of linguistic inquiry, so that at this level a great deal has been borrowed by NLP researchers. During the late 1980s and the early 1990s, this led to a convergence of interest in the two communities to the extent that at least some linguistic theorizing is now undertaken with explicit regard to computational issues. Head-driven Phase Structure Grammar and Tree Adjoining Grammar are probably the most visible results of this interaction. Linguistic theoreticians, in general, have paid less attention to the formal treatment of phenomena beyond syntax, and approaches to semantics and pragmatics within NLP have consequently tended to be somewhat more ad hoc, although ideas from formal semantics and speech act theory have found their way into NLP.
- The generative linguistics-based approach to NLP is sometimes constrasted with "empirical" approaches based on statistical and other data-driven analyses of raw data in the form of text corpora, that is, collections of machine-readable text. The empirical approach has been around since NLP began in the early 1950s, when the availability of computer technology made analysis of reasonably large text corpora increasingly viable, but it was soon pushed into the background by the well-known and highly influential opinions of Chomsky and his followers, who were strongly opposed to empirical methods in linguistics. A few researchers resisted the trend, however, and work on corpus collections, such as the Brown and LOB corpora, continued, but it is only in the last 10 years or so that empirical NLP has reemerged as a major alternative to "rationalist" lin-

guistics-based NLP. This resurgence is mainly attributable to the huge data storage capacity and extremely fast access and processing afforded by modern computer technology, together with the ease of generating large text corpora using word processors and optical character readers.

Corpora are primarily used as a source of information about language, and a number of techniques have emerged to enable the analysis of corpus data. Using these techniques, new approaches to traditional problems have also been developed. For example, syntactic analysis can be achieved on the basis of statistical probabilities estimated from a training corpus (as opposed to rules written by a linguist), lexical ambiguities can be resolved by considering the likelihood of one or another interpretation on the basis of context both near and distant, and measures of style can be computed in a more rigorous manner. "Parallel" corpora—equivalent texts in two or more languages—offer a source of contrastive linguistic information about different languages, and can be used, once they have been aligned, to extract bilingual knowledge useful for translation either in the form of traditional lexicons and transfer rules or in a wholly empirical way in example-based approaches to machine translation.

- Artificial neural network (ANN)–based NLP is the most recent of the three approaches. ANNs were proposed as a computational formalism in the early 1940s, and were developed alongside the equivalent automata theory and rewrite systems throughout the 1950s and 1960s. Because of their analogy with the physical structure of biological brains, much of this was strongly oriented toward cognitive modeling and simulation. Development slowed drastically when, in the late 1960s, it was shown that the ANN architectures known at the time were not universal computers, and that there were some very practical and cognitively relevant problems that these architectures could not solve. Throughout the 1970s relatively few researchers persevered with ANNs, but interest in them began to revive in the early 1980s, and for the first time some language-oriented ANN papers appeared. Then, in the mid-1980s, discovery of a way to overcome the previously demonstrated limitations of ANNs proved to be the catalyst for an explosion of interest in ANNs both as an object of study in their own right and as a technology for a range of application areas. One of these application areas has been NLP: since 1986 the volume of NL-oriented—and specifically NLP—research has grown very rapidly.

B. NLP and Allied Disciplines

The development of NLP is intertwined with that of several other language-related disciplines. This subsection specifies how, for the purposes of the handbook, these relate to NLP. This is not done merely as a matter of interest. Each of the disciplines has its own agenda and methodology, and if one fails to keep them distinct, confusion in one's own work readily ensues. Moreover, the literature associated with these various disciplines ranges from very large to huge, and much time can be wasted engaging with issues that are simply irrelevant to NLP. The handbook takes NLP to be exclusively concerned with the design and implementation of effective natural language input and output components for computational systems. On the basis

of this definition, we can compare NLP with each of the related disciplines in the following ways.

1. *Cognitive Science*

Cognitive science is concerned with the development of psychological theories; the human language faculty has always been central to cognitive theorizing and is, in fact, widely seen as paradigmatic for cognition generally. In other words, cognitive science aims in a scientific sense to explain human language. NLP as understood by this book does not attempt such an explanation. It is interested in designing devices that implement some linguistically relevant mapping. No claims about cognitive explanation or plausibility are made for these devices. What matters is whether they actually implement the desired mapping, and how efficiently they do so.

2. *Generative Linguistics*

Like cognitive science, generative linguistics aims to develop scientific theories, although about human language in particular. For Chomsky and his adherents, linguistics is, in fact, part of cognitive science; the implication of this for present purposes has already been stated. Other schools of generative linguistics make no cognitive claims, and regard their theories as formal systems, the adequacy of which is measured in terms of the completeness, internal consistency, and economy by which all scientific theories are judged. The rigor inherent in these formal systems means that, in many cases, they characterize facts and hypotheses about language in such a way that these characterizations can fairly straightforwardly be used in computational processing. Such transfers from the theoretical domain to the context of NLP applications are quite widespread. At the same time, however, there is always scope for tension between the theoretical linguist's desire for a maximally economical and expressive formal system and the NLP system designer's desire for broad coverage, robustness, and efficiency. It is this tension that has in part provoked interest in methods other than the strictly symbolic.

3. *Artificial Intelligence*

Research in artificial intelligence (AI) aims to design computational systems that simulate aspects of cognitive behavior. It differs from cognitive science in that no claim to cognitive explanation or plausibility is necessarily made with respect to the architectures and algorithms used. AI is thus close in orientation to NLP, but not identical: in the view we have chosen to adopt in this book, NLP aims not to simulate intelligent behavior per se, but simply to design NL input–output components for arbitrary computational applications; these might be components of AI systems, but need not be.

4. *Computational Linguistics*

Computational linguistics (CL) is a term that different researchers interpret in different ways; for many the term is synonymous with NLP. Historically founded as a discipline on the back of research into Machine Translation, and attracting researchers from a variety of neighboring fields, it can be seen as a branch of linguistics, computer science, or literary studies.

- As a branch of linguistics, CL is concerned with the computational implementation of linguistic theory: the computer is seen as a device for testing the completeness, internal consistency, and economy of generative linguistic theories.
- As a branch of computer science, CL is concerned with the relationship between natural and formal languages: interest here focuses on such issues as language recognition and parsing, data structures, and the relationship between facts about language and procedures that make use of those facts.
- As a branch of literary studies, CL involves the use of computers to process large corpora of literary data in, for example, author attribution of anonymous texts.

Within whatever interpretation of CL one adopts, however, there is a clear demarcation between domain-specific theory on the one hand and practical development of computational language processing systems on the other. It is this latter aspect—practical development of computational language processing systems—that, for the purposes of this book, we take to be central to NLP. In recent years, this approach has often been referred to as "language technology" or "language engineering."

Our view of NLP can, then, be seen as the least ambitious in a hierarchy of disciplines concerned with NL. It does not aim to explain or even to simulate the human language faculty. What it offers, however, is a variety of practical tools for the design of input–output modules in applications that can benefit from the use of natural language. And, because this handbook is aimed at language-engineering professionals, that is exactly what is required.

2. THE SCOPE OF THIS BOOK

Taken at face value, the expression "natural language processing" encompasses a great deal. Ultimately, applications and technologies such as word processing, desktop publishing, and hypertext authoring are all intimately involved with the handling of natural language, and so in a broad view could be seen as part of the remit of this book. In fact, it is our belief that, in the longer term, researchers in the field will increasingly look to integration of these technologies into mainstream NLP research. But there are limits to what can usefully be packed between the covers of a book, even of a large book such as this, and there is much else to be covered, so a degree of selection is inevitable. Our particular selection of topics was motivated by the following considerations.

Any computational system that uses natural language input and output can be seen in terms of a five-step processing sequence (Fig. 1):

1. The system receives a physical signal from the external world, where that signal encodes some linguistic behavior. Examples of such signals are speech waveforms, bitmaps produced by an optical character recognition or handwriting recognition system, and ASCII text streams received from an electronic source. This signal is converted by a transducer into a linguistic representation amenable to computational manipulation.
2. The natural language analysis component takes the linguistic representation from step 1 as input and transforms it into an internal representation appropriate to the application in question.

3. The application takes the output from step 2 as one of its inputs, carries out a computation, and outputs an internal representation of the result.
4. The natural language generation component takes part of the output from step 3 as input, transforms it into a representation of a linguistic expression, and outputs that representation.
5. A transducer takes the output representation from step 4 and transforms it into a physical signal that, in the external world, is interpretable by humans as language. This physical signal might be a text stream, the glyphs in a printed document, or synthesized speech.

Ultimately, the input to a language-processing system will be in graphemic or phonetic form, as will the outputs. Phonetic form corresponds to speech input. Traditionally, however, the only kind of graphemic input dealt with in language-processing systems is digitally encoded text. In the case of such graphemic input it is generally assumed either that this will be the native form of the input (an assumption that is indeed true for a great deal of the data we might wish to process) or that some independent process will map the native form into digitally encoded text. This assumption is also made in much—although certainly not all—work on speech: a great deal of this research proceeds on the basis that mapping speech signals into a textual representation consisting of sequences of words can be achieved independently of any process that subsequently derives some other representation from this text. It is for these reasons that we have chosen in the present work to focus on the processing techniques that can be used with textual representations, and thus exclude from direct consideration techniques that have been developed for speech processing.

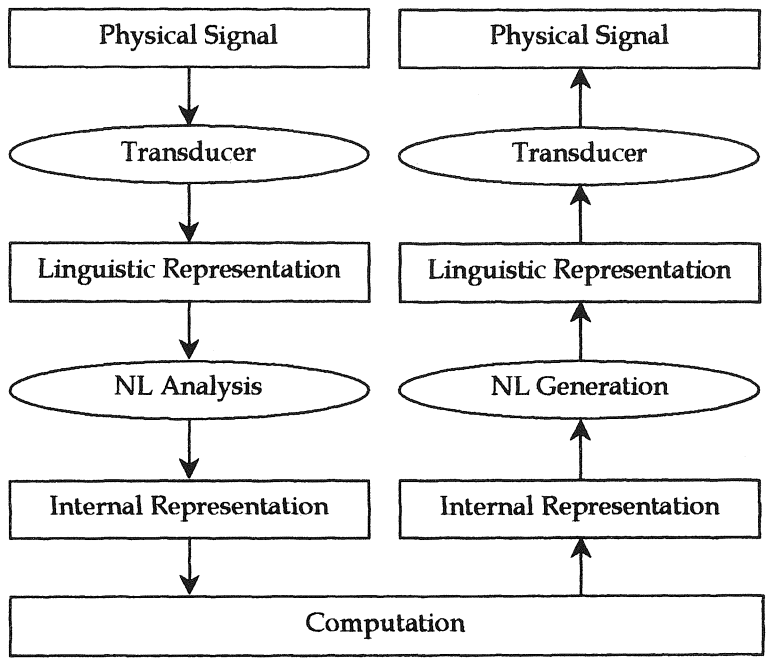

Fig. 1 The five steps in computational processing of natural language input and output.

Subject, then, to the foregoing restrictions of scope, coverage of NLP tools and applications in this handbook attempts to strike a balance between comprehensiveness and depth of coverage with the aim of providing a practical guide for practical applications.

3. THE STRUCTURE OF THIS BOOK

After some five decades of NLP research, and of NLP-related work in the aforementioned disciplines, a huge research literature has accumulated. Because the primary function of this handbook is to present that material as accessibly as possible, clarity of organization is paramount. After much discussion, we realized that no organization was going to be ideal, but two reasonable possibilities did emerge: a historically based organization and a topic-based one. The first of these would partition the material into three sections corresponding to the three main strands of development described in the foregoing historical sketch, and the second would attempt to identify a set of research topics that would comprehensively describe NLP and NLP-related research to date by presenting the relevant work in each topic area. There are drawbacks to both. The main problem with the historically based organization is that it might be seen to reinforce the distinctions between the three approaches to NLP: although these distinctions have historical validity, they are increasingly being broken down, and for the handbook to partition the material in this way would distort the current state of increasingly ecumenical NLP research. The main problem with topic-based organization was identification of a suitable set of research areas. Demarcations that seemed natural from one point of view seemed less so from another, and attempts to force all the material into canonical compromise categories simply distorted it, with consequent effects on clarity. A morphology–syntax–semantics categorization, for example, seems perfectly natural from the point of view of linguistics-based NLP, but far less so from an empirical or ANN-based one. There are, moreover, important topics unique to each of the main strands of historical development, such as representation in the ANN-based approach. In fact, the more closely one looks, the more problematical identification of an adequate set of topics becomes.

In the end, the historically based organization was adopted. The motivation for this was the need for clarity: each of the three approaches to NLP can be described in its own self-contained section without the distortion that a topic-based organization would often require. To mitigate the problem of insularity, each section indicates areas of overlap with the others. Inevitably, though, the problem has not thereby been eliminated.

The handbook is, then, organized as follows. There are three main parts corresponding to the three main strands of historical development:

1. Symbolic approaches to NLP, which focuses on NLP techniques and applications that have their origins in generative linguistics
2. NLP based on empirical corpus analysis
3. NLP based on artificial neural networks

Again for clarity, each section is subdivided into the following subsections:

1. An introduction that gives an overview of the approach in question
2. A set of chapters describing the fundamental concepts and tools appropriate to that approach
3. A set of chapters describing particular applications in which the approach has been successfully used

Beyond this basic format, no attempt has been made to impose uniformity on the individual parts of the handbook. Each is the responsibility of a different editor, and each editor has presented his material in a way that he considers most appropriate. In particular, given that work in ANN–based NLP will be relatively new to many readers of this book, the introduction to Part III includes a detailed overview of how these techniques have developed and found a place in NLP research.

Some acknowledgments are appropriate. First and foremost, we express our thanks to the many contributors for their excellent efforts in putting together what we hope will be a collection of long-standing usefulness. We would also like to thank the editorial staff of Marcel Dekker, Inc., for their considerable patience in response to the succession of delays that is inevitable in the marshaling of a book of this size and authorial complexity. Finally, Debbie Whittington of the Microsoft Research Institute at Macquarie University and Rowena Bryson of the Centre for Research in Linguistics at Newcastle University deserve special mention for their administrative help.

Robert Dale
Hermann Moisl
Harold Somers

Contents

Contributors

James Allen Department of Computer Science, University of Rochester, Rochester, New York

Elisabeth André German Research Center for Artificial Intelligence, Saarbrücken, Germany

Ion Androutsopoulos Institute of Informatics and Telecommunications, National Centre for Scientific Research "Demokritos," Athens, Greece

Eric Brill Microsoft Research, Redmond, Washington

John A. Carroll School of Cognitive and Computing Sciences, University of Sussex, Brighton, England

Chun-Hsien Chen Department of Information Management, Chang Gung University, Kwei-Shan, Tao-Yuan, Taiwan, Republic of China

Jim Cowie Computing Research Laboratory, New Mexico State University, Las Cruces, New Mexico

Ido Dagan Department of Mathematics and Computer Science, Bar-Ilan University, Ramat Gan, Israel

Robert Dale Department of Computing, Macquarie University, Sydney, Australia

Georg Dorffner Austrian Research Institute for Artificial Intelligence, and Department of Medical Cybernetics and Artificial Intelligence, University of Vienna, Vienna, Austria

George Ferguson Department of Computer Science, University of Rochester, Rochester, New York

Barbara J. Grosz Division of Engineering and Applied Sciences, Harvard University, Cambridge, Massachusetts

Simon Haykin Communications Research Laboratory, McMaster University, Hamilton, Ontario, Canada

George E. Heidorn Microsoft Research, Redmond, Washington

Stefan Heil Technical University of Munich, Munich, Germany

Vasant Honavar Department of Computer Science, Iowa State University, Ames, Iowa

Richard I. Kittredge* Department of Linguistics and Translation, University of Montreal, Montreal, Quebec, Canada

Ludovic Lebart Economics and Social Sciences Department, Ecole Nationale Supérieure des Télécommunications, Paris, France

Karen E. Lochbaum U S WEST Advanced Technologies, Boulder, Colorado

Simon Lucas Department of Electronic Systems Engineering, University of Essex, Colchester, England

Yuji Matsumoto Graduate School of Information Science, Nara Institute of Science and Technology, Ikoma, Nara, Japan

David D. McDonald Brandeis University, Arlington, Massachusetts

Tony McEnery Department of Linguistics, Bowland College, Lancaster University, Lancaster, England

Kathleen R. McKeown Department of Computer Science, Columbia University, New York, New York

Dieter Merkl Department of Software Technology, Vienna University of Technology, Vienna, Austria

Risto Miikkulainen Department of Computer Sciences, The University of Texas at Austin, Austin, Texas

Bradford W. Miller Cycorp, Austin, Texas

Hermann Moisl Centre for Research in Linguistics, University of Newcastle, Newcastle-upon-Tyne, England

**Also affiliated with*: CoGenTex, Inc., Ithaca, New York.

Michael Oakes Department of Linguistics, Lancaster University, Lancaster, England

David D. Palmer The MITRE Corporation, Bedford, Massachusetts

Rajesh G. Parekh Allstate Research and Planning Center, Menlo Park, California

Massimo Poesio Human Communication Research Centre and Division of Informatics, University of Edinburgh, Edinburgh, Scotland

Alain Polguère Department of Linguistics and Translation, University of Montreal, Montreal, Quebec, Canada

Dragomir R. Radev School of Information, University of Michigan, Ann Arbor, Michigan

Martin Rajman Artificial Intelligence Laboratory (LIA), EPFL, Swiss Federal Institute of Technology, Lausanne, Switzerland

Eric K. Ringger* Department of Computer Science, University of Rochester, Rochester, New York

Graeme Ritchie Division of Informatics, University of Edinburgh, Edinburgh, Scotland

Christer Samuelsson Xerox Research Centre Europe, Grenoble, France

Jürgen Schmidhuber IDSIA, Lugano, Switzerland

Candace L. Sidner Lotus Research, Lotus Development Corporation, Cambridge, Massachusetts

Teresa Sikorski Zollo Department of Computer Science, University of Rochester, Rochester, New York

Harold Somers Centre for Computational Linguistics, UMIST, Manchester, England

Richard Sproat Department of Human/Computer Interaction Research, AT&T Labs–Research, Florham Park, New Jersey

Henry S. Thompson Division of Informatics, University of Edinburgh, Edinburgh, Scotland

**Current affiliation*: Microsoft Research, Redmond, Washington

Takehito Utsuro Graduate School of Information Science, Nara Institute of Science and Technology, Ikoma, Nara, Japan

Stefan Wermter Centre for Informatics, SCET, University of Sunderland, Sunderland, England

Yorick Wilks Department of Computer Science, University of Sheffield, Sheffield, England

Mats Wirén Telia Research, Stockholm, Sweden

Michael Witbrock Lycos Inc., Waltham, Massachusetts

Dekai Wu Department of Computer Science, The Hong Kong University of Science and Technology, Kowloon, Hong Kong

David Yarowsky Department of Computer Science, Johns Hopkins University, Baltimore, Maryland

1

Symbolic Approaches to Natural Language Processing

ROBERT DALE

Macquarie University, Sydney, Australia

1. INTRODUCTION

Ever since the early days of machine translation in the 1950s, work in natural language processing (NLP) has attempted to use symbolic methods—where knowledge about language is explicitly encoded in rules or other forms of representation—as a means to solving the problem of how to automatically process human languages. In this first part of the handbook, we look at the body of techniques that have developed in this area over the past forty years, and we examine some categories of applications that make use of these techniques. In more recent years, the availability of large corpora and datasets has reminded us of the difficulty of scaling-up many of these symbolic techniques to deal with real-world problems; many of the techniques described in the second and third parts of this handbook are a reaction to this difficulty. Nonetheless, the symbolic techniques described here remain important when the processing of abstract representations is required, and it is now widely recognised that the key to automatically processing human languages lies in the appropriate combination of symbolic and nonsymbolic techniques. Indeed, some of the chapters in this part of the book already demonstrate, both in terms of techniques and the use of those techniques within specific applications, that the first steps toward hybrid solutions are being taken.

In this introduction, we provide an overview of the contents of this part of the handbook. The overview separates the chapters into two clusters, corresponding to the major subdivision we have adopted in all three parts of the book: the first six chapters contained here focus on specific techniques that can be used in building

natural language processing applications, and the remaining seven chapters describe applications that all use some combination of the techniques described earlier.

2. TECHNIQUES

Traditionally, work in natural language processing has tended to view the process of language analysis as being decomposable into a number of stages, mirroring the theoretical linguistic distinctions drawn between SYNTAX, SEMANTICS, and PRAGMATICS. The simple view is that the sentences of a text are first analysed in terms of their syntax; this provides an order and structure that is more amenable to an analysis in terms of semantics, or literal meaning; and this is followed by a stage of pragmatic analysis whereby the meaning of the utterance or text in context is determined. This last stage is often seen as being concerned with DISCOURSE, whereas the previous two are generally concerned with sentential matters. This attempt at a correlation between a stratificational distinction (syntax, semantics, and pragmatics) and a distinction in terms of granularity (sentence versus discourse) sometimes causes some confusion in thinking about the issues involved in natural language processing; and it is widely recognised that in real terms, it is not so easy to separate the processing of language neatly into boxes corresponding to each of the strata. However, such a separation serves as a useful pedagogical aid, and also constitutes the basis for architectural models that make the task of natural language analysis more manageable from a software-engineering point of view.

Nonetheless, the tripartite distinction into syntax, semantics, and pragmatics serves, at best, only as a starting point when we consider the processing of real natural language texts. A finer-grained decomposition of the process is useful when we take into account the current state of the art in combination with the need to deal with real language data; this is reflected in Fig. 1.

Unusually for books on natural language processing, we identify here the stage of tokenisation and sentence segmentation as a crucial first step. Natural language text is generally not made up of the short, neat, well-formed and well-delimited sentences we find in textbooks; and for languages such as Chinese, Japanese, or Thai, which do not share the apparently easy space-delimited tokenisation we might believe to be a property of languages such as English, the ability to address issues of tokenisation is essential to getting off the ground at all. We also treat lexical analysis as a separate step in the process. To some degree this finer-grained decomposition reflects our current state of knowledge about language processing: we know quite a lot about general techniques for tokenisation, lexical analysis, and syntactic analysis, but much less about semantics and discourse-level processing. But it also reflects the fact that the known is the surface text, and anything deeper is a representational abstraction that is harder to pin down; so it is not so surprising that we have better-developed techniques at the more concrete end of the processing spectrum.

Natural language analysis is only one half of the story. We also have to consider natural language generation, when we are concerned with mapping from some (typically nonlinguistic) internal representation to a surface text. In the history of the field so far, there has been much less work on natural language generation than there has been on natural language analysis. One sometimes hears the suggestion that this is because natural language generation is easier, so that there is less to be said. As

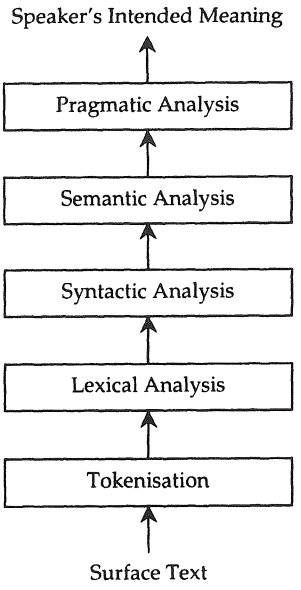

Fig. 1 The stages of analysis in processing natural language.

David McDonald demonstrates in Chap. 7, this is far from the truth: there are a great many complexities to be addressed in generating fluent and coherent multi-sentential texts from an underlying source of information. A more likely reason for the relative lack of work in generation is precisely the correlate of the observation made at the end of the previous paragraph: it is relatively straightforward to build theories around the processing of something known (such as a sequence of words), but much harder when the input to the process is more or less left to the imagination. This is the question that causes researchers in natural language generation to wake in the middle of the night in a cold sweat: what does generation start from? Without a clear idea of where to start, and without independently motivated input representations, it can be hard to say very much that is not idiosyncratic. McDonald offers some suggestions here, although many of the problems disappear when we face the need to construct real applications of natural language generation, two of which are described in the second half of this part of the handbook.

A. Chapter 2: Tokenisation and Sentence Segmentation

As we have already noted, not all languages deliver text in the form of words neatly delimited by spaces. Languages such as Chinese, Japanese, and Thai require first that a segmentation process be applied, analogous to the segmentation process that must first be applied to a continuous speech stream to identify the words that make up an utterance. As Palmer demonstrates in Chap. 2, there are also significant segmentation and tokenisation issues in apparently easier-to-segment languages—such as English. Fundamentally, the issue here is that of what constitutes a word; as Palmer shows, there is no easy answer here. This chapter also looks at the problem of sentence segmentation: because so much work in natural language processing

views the sentence as the unit of analysis, clearly it is of crucial importance to ensure that, given a text, we can break it into sentence-sized pieces. This turns out not to be so trivial either. Palmer offers a catalog of tips and techniques that will be useful to anyone faced with dealing with real raw text as the input to an analysis process, and provides a healthy reminder that these problems have tended to be idealised away in much earlier, laboratory-based work in natural language processing.

B. Chapter 3: Lexical Analysis

The previous chapter addressed the problem of breaking a stream of input text into the words and sentences that will be subject to subsequent processing. The words are not atomic and are themselves open to further analysis. Here we enter the realms of computational morphology, the focus of Richard Sproat's chapter. By taking words apart, we can uncover information that will be useful at later stages of processing. The combinatorics also mean that decomposing words into their parts, and maintaining rules for how combinations are formed, is much more efficient in terms of storage space than would be true if we simply listed every word as an atomic element in a huge inventory. And, once more returning to our concern with the handling of real texts, there will always be words missing from any such inventory; morphological processing can go some way toward handling such unrecognised words. Sproat provides a wide-ranging and detailed review of the techniques that can be used to carry out morphological processing, drawing on examples from languages other than English to demonstrate the need for sophisticated processing methods; along the way he provides some background in the relevant theoretical aspects of phonology and morphology, and draws connections with questions of tokenisation discussed in the previous chapter.

C. Chapter 4: Parsing Techniques

A presupposition in most work in natural language processing is that the basic unit of meaning analysis is the sentence: a sentence expresses a proposition, idea, or thought, and says something about some real or imaginary world. Extracting the meaning from a sentence is thus a key issue. Sentences are not, however, just linear sequences of words, and so it is widely recognised that to carry out this task requires an analysis of each sentence which determines its structure in one way or another. In NLP approaches based on generative linguistics, this is generally taken to involve determination of the syntactic or grammatical structure of each sentence. In their chapter, Samuelsson and Wirén present a range of techniques that can be used to achieve this end. This area is probably the most well-established in the field of NLP, enabling the authors here to provide an inventory of basic concepts in parsing, followed by a detailed catalog of parsing techniques that have been explored in the literature. Samuelsson and Wirén end by pointing to some of the issues that arise in applying these techniques to real data, with the concomitant problems of undergeneration (where one's grammar is unable to provide an analysis for a specific sentence) and ambiguity (where multiple analyses are provided).

D. Chapter 5: Semantic Analysis

Identifying the underlying syntactic structure of a sequence of words is only one step in determining the meaning of a sentence; it provides a structured object that is more amenable to further manipulation and subsequent interpretation. It is these subsequent steps that derive a meaning for the sentence in question. Massimo Poesio's chapter turns to these deeper issues. It is here that we begin to reach the bounds of what has managed to move from the research laboratory to practical application. As pointed out earlier in this introduction, the semantics of natural language have been less studied than syntactic issues, and so the techniques described here are not yet developed to the extent that they can easily be applied in a broad-coverage fashion.

After setting the scene by identifying some of the issues that arise in semantic interpretation, Poesio begins with a brief primer on the variety of knowledge representation formalisms that have been used in semantic interpretation. He then goes on to show how semantic interpretation can be integrated with the earlier stages of linguistic processing, relying on some of the machinery introduced in Samuelsson and Wirén's chapter. A substantial proportion of the chapter is given over to a discussion of how semantic interpretation plays a role in one particularly ubiquitous problem in language processing: that of the interpretation of anaphoric elements, such as pronouns and definite noun phrases, the meaning of which relies on an interpretation of prior elements of a text.

E. Chapter 6: Discourse Structure and Intention Recognition

The chapter by Lochbaum, Grosz, and Sidner picks up where Poesio's leaves off. Just as a sentence is more than a linear sequence of words, a discourse is more than a linear sequence of sentences. The relations between the meanings of individual sentences mean that discourses can have quite complex hierarchical structures; and these structures are closely tied to the intentions of the speaker or speakers in creating the resulting discourse.

Lochbaum, Grosz, and Sidner consider these two interrelated aspects of discourse processing, showing how the speaker's intentions play a role in determining the structure of the discourse, and how it is important to recognise the speaker's intentions when attempting to determine the role of an individual utterance in a larger discourse. The chapter first surveys a number of approaches to analysing the structure of discourse, and then goes on to look at what is involved in recognising the intention or intentions underlying a speaker's utterance. These two aspects are then drawn together using the notion of SharedPlans.

The techniques presented in this chapter are currently beyond what we can reasonably achieve in the broad-coverage analysis of unrestricted text; however, they can serve a useful purpose in restricted domains of discourse, and their discussion raises awareness of the issues that can have ramifications for shallower approaches to multisentential text processing.

F. Chapter 7: Natural Language Generation

At the end of the day, determining the speaker's intentions underlying an utterance is only really one-half of the story of natural language processing: in many situations, a response then needs to be generated, either in natural language alone or in combina-

tion with other modalities. For many of today's applications, what is required here is rather trivial and can be handled by means of canned responses; increasingly, however, we are seeing natural language generation techniques applied in the context of more sophisticated back-end systems, where the need to be able to custom-create fluent multisentential texts on demand becomes a priority. Chapters 11 and 12 bear testimony to the scope here.

In Chap. 7, David McDonald provides a far reaching survey of work in the field of natural language generation. McDonald begins by lucidly characterising the differences between natural language analysis and natural language generation. He goes on to show what can be achieved using natural language generation techniques, drawing examples from systems developed over the last 25 years. The bulk of the chapter is then concerned with laying out a picture of the component processes and representations required to generate fluent multisentential or multiparagraph texts, built around the now-standard distinction between text planning and linguistic realisation.

3. APPLICATIONS

In one of the first widely available published reports on the scope for practical applications of natural language processing, the Ovum report on *Natural Language Computing: The Commercial Applications* (Johnson, 1985) identified a number of specific application types:

- Text critiquing
- Natural language database query
- Message understanding
- Machine translation
- Speech recognition

At the time, enthusiasm for the technology was high. With the benefit of hindsight, it is clear that the predictions made at that time regarding the size of the market that would develop have been undermined by slower progress in developing the technology than was then thought likely. However, significant advances have been made in the nearly 15 years since that report appeared; and, perhaps surprisingly, the perceived application types have remained relatively constant in broad terms. An application area not predicted by the Ovum report is that of automatic text generation: this has arisen as an area of significant promise because of the scope for dynamic document tailoring on the World Wide Web and the scope for the use of large existing databases of information as an input to a text generation process.

In this second half of this part of the handbook, we present a number of reports on working NLP systems from these application areas.

A. Chapter 8: Intelligent Writing Assistance

In this chapter, George Heidorn describes the workings of the grammar checker first introduced in Microsoft Word 97, probably the first natural language processing application to reach desktops in the millions. Heidorn goes through the various steps of processing used in the grammar checker in considerable detail, demonstrating how issues discussed in Chaps. 2–4, in particular, have a bearing in constructing a real-

world application of broad-coverage NLP technology. The chapter also contains useful and interesting details on a variety of issues in the development and testing of the grammar-checking technology; the chapter is a valuable look behind the scenes at what it takes to make a piece of successful commercial-strength NLP technology.

B. Chapter 9: Database Interfaces

The idea of being able to replace the manual construction of complex database queries in SQL or some other query language has long held an appeal for researchers in natural language processing; this chapter provides an extensive survey of the issues involved in building such systems.

After sketching some history, Androutsopoulos and Ritchie present a detailed analysis of the architecture of natural language database interfaces (NLDBIs). The model developed here draws on ideas presented in Chaps. 2–5. The authors go on to address the important question of the portability of these kinds of systems—we want to be able to avoid having to rebuild from scratch for each new database—and then consider a number of more advanced issues that arise in trying to build truly flexible and robust systems. The chapter ends with a discussion of variants on true natural language interfaces, where controlled languages or menu-based interfaces are used.

C. Chapter 10: Information Extraction

An area of significant interest in the late 1980s and the 1990s has been that commonly referred to as INFORMATION EXTRACTION or MESSAGE UNDERSTANDING. In opposition to the more traditionally conceived task of natural language processing, where the aim is to understand a complete text to a sophisticated degree of analysis, work in this area adopts a much simpler notion of what it is to extract meaning from a text. Specifically, if we know we have a collection of texts that are of essentially the same kind—for example, they may all be reports about terrorist incidents in some part of the world—then in many contexts it can be useful to extract from each of these texts a predefined set of elements; in essence, who did what to whom, when, and where.

The chapter by Cowie and Wilks provides an overview of the techniques involved in building such systems. The authors also provide some views on the role of this kind of work in the field of NLP as a whole.

D. Chapter 11: The Generation of Reports from Databases

In the first of two chapters focussing on practical applications of natural language generation technology, Richard Kittredge and Alain Polguère look at how textual reports can be automatically generated from databases. They begin by providing a useful characterisation of the properties a dataset needs to have to be amenable to the fruitful use of this technology; they then go on to describe the kinds of knowledge that need to be built into such systems, before describing a large number of example systems that carry out report generation. The second half of their chapter focusses on the computational techniques required for report generation, providing a usefully detailed analysis of many of the core ideas introduced in McDonald's earlier chapter

(see Chap. 7). The chapter ends by contrasting report generation from databases with the generation of text from richer knowledge bases.

E. Chapter 12: The Generation of Multimedia Presentations

Text is often not the best way to convey information: as the oft-cited adage reminds us, 'a picture is worth a thousand words.' Generally, however, those pictures need to be supported by words, or words need to be supported by pictures; neither alone is sufficient.

Following on from the previous chapter, Elisabeth André describes work that aims to generate documents that contain both text and graphic elements. After presenting a general architecture for the development of such systems, André first shows how ideas from notions of discourse structure developed in linguistics can be applied to the generation of documents containing pictoral elements; she then discusses a variety of issues that arise in building real working systems that integrate multiple modalities.

F. Chapter 13: Machine Translation

Machine translation is the oldest application of natural language processing technology, having been around for 50 years. Appropriately, Harold Somers begins this chapter by presenting a historical overview of the field. He then shows how various problems in natural language processing more generally manifest themselves in machine translation; the discussion here echoes points made in Chaps. 3–5 of this book. Somers goes on to describe the two principal approaches taken to the construction of machine translation systems: interlingua-based approaches, in which an intermediate language-independent representational level is used, and transfer approaches, in which mappings are carried out more directly between language pairs. The chapter ends by discussing the user's perspective on the technology, highlighting how a consideration of how the technology can actually be used in practice has an influence on how we build systems.

G. Chapter 14: Dialogue Systems: From Theory to Practice in TRAINS-96

Full-blown natural language dialogue systems are probably the ultimate dream of many of those working in natural language processing. Allen and his colleagues describe here a system that takes advantage, in one way or another, of all the techniques described in the first half of this part of the handbook: the TRAINS system, an end-to-end spoken dialogue system, interacts with the user to solve route planning problems. As well as using the symbolic approaches to syntax, semantics, and pragmatics, outlined earlier, the TRAINS system also makes use of some of the statistical techniques outlined in the second part of this handbook; and contrary to our focus in this book on the processing of text, the TRAINS system takes spoken input and produces spoken output. It is thus an excellent example of what can be achieved by integrating the results of the last several decades of research into NLP; although there are many outstanding issues to be addressed in scaling-up a system of the sophistication of TRAINS to a real-world application, the system serves as a demonstration of where we can expect developments to take us in the next 5–10 years.

4. FURTHER READING

Each of the authors of the chapters presented here has provided an extensive bibliography for material relevant to their specific subject matter. We provide here some pointers to more general background literature in symbolic approaches to natural language processing.

At the time of writing, the best widely available introduction to techniques in natural language understanding is James Allen's book (Allen, 1995); another excellent starting point is Gazdar and Mellish's book (Gazdar and Mellish, 1989). Reiter and Dale (2000) provide an extensive introduction to the issues involved in building natural language generation systems.

For a broad overview of seminal research in the field, see Grosz et al. (1986); although now a little dated, this collects together a large number of papers that were influential in setting directions for the field. A more recent collection of papers can be found in Pereira and Grosz (1993). Work in speech recognition is not addressed in this handbook; but see Waibel and Lee (1990) for a useful collection of important papers. For a broad overview of natural language processing that covers the territory exhaustively without going into technical detail, see the Cambridge Human Language Technology Survey (Cole et al., 1998).

REFERENCES

Allen J. 1995. Natural Language Understanding. 2nd ed. Redwood City, CA: Benjamin Cummings.

Cole R, Mariani J, Uszkoreit H, Varile GB, Zaenen A, Zampolli A. 1998. Survey of the State of the Art in Human Language Technology. Cambridge, UK: Cambridge University Press.

Gazdar G, Mellish C. 1989. Natural Language Processing in Lisp: An Introduction to Computational Linguistics. Wokingham, UK: Addison–Wesley.

Grosz BJ, Sparck Jones K, Webber B, eds. 1986. Readings in Natural Language Processing. San Francisco, CA: Morgan-Kaufmann.

Johnson T. 1985. Natural Language Computing: The Commercial Applications. London: Ovum Ltd.

Pereira F, Grosz BJ, eds. 1993. Artif Intell 63(1–2). Special Issue on Natural Language Processing.

Reiter E, Dale R. 2000. Building Natural Language Generation Systems. Cambridge, UK: Cambridge University Press.

Waibel A, Lee K-F. 1990. Readings in Speech Recognition. San Francisco, CA: Morgan Kaufmann.

2

Tokenisation and Sentence Segmentation

DAVID D. PALMER

The MITRE Corporation, Bedford, Massachusetts

1. INTRODUCTION

In linguistic analysis of a natural language text, it is necessary to clearly define what constitutes a word and a sentence. Defining these units presents different challenges depending on the language being processed, but neither task is trivial, especially when considering the variety of human languages and writing systems. Natural languages contain inherent ambiguities, and much of the challenge of Natural Language Processing (NLP) involves resolving these ambiguities.

In this chapter we will discuss the challenges posed by **text segmentation**, the task of dividing a text into linguistically meaningful units—at the lowest level characters representing the individual graphemes in a language's written system, words consisting of one or more characters, and sentences consisting of one or more words. Text segmentation is a frequently overlooked yet essential part of any NLP system, because the words and sentences identified at this stage are the fundamental units passed to further processing stages, such as morphological analyzers, part-of-speech taggers, parsers, and information retrieval systems.

Tokenisation is the process of breaking up the sequence of characters in a text by locating the **word boundaries**, the points where one word ends and another begins. For computational linguistics purposes, the words thus identified are frequently referred to as **tokens**. In written languages where no word boundaries are explicitly marked in the writing system, tokenisation is also known as **word segmentation**, and this term is frequently used synonymously with tokenisation.

Sentence segmentation is the process of determining the longer processing units consisting of one or more words. This task involves identifying **sentence boundaries** between words in different sentences. Since most written languages have punctuation marks that occur at sentence boundaries, sentence segmentation is frequently referred to as **sentence boundary detection**, **sentence boundary disambiguation**, or **sentence boundary recognition**. All these terms refer to the same task: determining how a text should be divided into sentences for further processing.

In practice, sentence and word segmentation cannot be performed successfully independent from one another. For example, an essential subtask in both word and sentence segmentation for English is identifying abbreviations, because a period can be used in English to mark an abbreviation as well as to mark the end of a sentence. When a period marks an abbreviation, the period is usually considered a part of the abbreviation token, whereas a period at the end of a sentence is usually considered a token in and of itself. Tokenising abbreviations is complicated further when an abbreviation occurs at the end of a sentence, and the period marks *both* the abbreviation and the sentence boundary.

This chapter will provide an introduction to word and sentence segmentation in a variety of languages. We will begin in Sec. 2 with a discussion of the challenges posed by text segmentation, and emphasize the issues that must be considered before implementing a tokenisation or sentence segmentation algorithm. The section will describe the dependency of text segmentation algorithms on the language being processed and the character set in which the language is encoded. It will also discuss the dependency on the application that uses the output of the segmentation and the dependency on the characteristics of the specific corpus being processed.

In Sec. 3 we will introduce some common techniques currently used for tokenisation. The first part of the section will focus on issues that arise in tokenising languages in which words are separated by whitespace. The second part of the section will discuss tokenisation techniques in languages for which no such whitespace word boundaries exist. In Sec. 4 we will discuss the problem of sentence segmentation and introduce some common techniques currently used to identify sentences boundaries in texts.

2. WHY IS TEXT SEGMENTATION CHALLENGING?

There are many issues that arise in tokenisation and sentence segmentation that need to be addressed when designing NLP systems. In our discussion of tokenisation and sentence segmentation, we will emphasize the main types of dependencies that must be addressed in developing algorithms for text segmentation: **language dependence** (Sec. 2.A), **character-set dependence** (Sec. 2.B), **application dependence** (Sec. C), and **corpus dependence** (Sec. 2.D). In Sec. 2.E, we will briefly discuss the evaluation of the text segmentation algorithms.

A. Language Dependence

We focus here on written language, rather than transcriptions of spoken language. This is an important distinction, for it limits the discussion to those languages that have established writing systems. Because every human society uses language, there are thousands of distinct languages and dialects; yet only a small minority of the

languages and dialects currently have a system of visual symbols to represent the elements of the language.

Just as the spoken languages of the world contain a multitude of different features, the written adaptations of languages have a diverse set of features. Writing systems can be **logographic**, in which a large number (often thousands) of individual symbols represent words. In contrast, writing systems can be **syllabic**, in which individual symbols represent syllables, or **alphabetic**, in which individual symbols (more or less) represent sounds; unlike logographic systems, syllabic and alphabetic systems typically have fewer than 100 symbols. In practice, no modern writing system employs symbols of only one kind, so no natural language writing system can be classified as purely logographic, syllabic, or alphabetic. Even English, with its relatively simple writing system based on the Roman alphabet, utilizes logographic symbols including Arabic numerals (0–9), currency symbols ($, £), and other symbols (%, &, #). English is nevertheless predominately alphabetic, and most other writing systems comprise symbols that are mainly of one type.

In addition to the variety of symbol types used in writing systems, there is a range of orthographic conventions used in written languages to denote the boundaries between linguistic units such as syllables, words, or sentences. In many written Amharic texts, for example, both word and sentence boundaries are explicitly marked, whereas in written Thai texts neither is marked. In the latter example in which no boundaries are explicitly indicated in the written language, written Thai is similar to spoken language, for which there are no explicit boundaries and few cues to indicate segments at any level. Between the two extremes are languages that mark boundaries to different degrees. English employs whitespace between most words and punctuation marks at sentence boundaries, but neither feature is sufficient to segment the text completely and unambiguously. Tibetan and Korean both explicitly mark syllable boundaries, either through layout or by punctuation, but neither marks word boundaries. Written Chinese and Japanese have adopted punctuation marks for sentence boundaries, but neither denotes word boundaries. For a very thorough description of the various writing systems employed to represent natural languages, including detailed examples of all languages and features discussed in this chapter, we recommend Daniels and Bright (1996).

The wide range of writing systems used by the languages of the world result in language-specific as well as orthography-specific features that must be taken into account for successful text segmentation. The first essential step in text segmentation is thus to understand the writing system for the language being processed. In this chapter we will provide general techniques applicable to a variety of different writing systems. As many segmentation issues are language-specific, we will also highlight the challenges faced by robust, broad-coverage tokenisation efforts.

B. Character-Set Dependence

At its lowest level, a computer-based text or document is merely a sequence of digital bits stored in a computer's memory. The first essential task is to interpret these bits as characters of a writing system of a natural language. Historically, this was trivial, because nearly all texts were encoded in the 7-bit ASCII character set, which allowed only 128 characters and included only the Roman alphabet and essential characters for writing English. This limitation required the "asciification" or "romanization" of

many texts, in which ASCII equivalents were defined for characters not defined in the character set. An example of this asciifiation is the adaptation of many European languages containing umlauts and accents, in which the umlauts are replaced by a double quotation mark or the letter **e** and accents are denoted by a single quotation mark or even a number code. In this system, the German word *über* would be written as *u"ber* or *ueber*, and the French word *déjà* would be written *de'ja`* or *de1ja2*. Languages less similar to English, such as Russian and Hebrew, required much more elaborate romanization systems, to allow ASCII processing. In fact, such adaptations are still common because many current e-mail systems are limited to this 7-bit encoding.

Extended character sets have been defined to allow 8-bit encodings, but an 8-bit computer byte can still encode just 256 distinct characters, and there are tens of thousands of distinct characters in all the writing systems of the world. Consequently, characters in different languages are currently encoded in a large number of overlapping character sets. Most alphabetic and syllabic writing systems can be encoded in a single byte, but larger character sets, such as those of written Chinese and Japanese, which have several thousand distinct characters, require a two-byte system in which a single character is represented by a pair of 8-bit bytes. It is further complicated by the fact that multiple encodings currently exist for the same character set; for example, the Chinese character set is encoded in two widely used variants: GB and Big-5.[1] The fact that the same range of numeric values represents different characters in different encodings can be a problem for tokenisation, because tokenisers are usually targeted to a specific language in a specific encoding. For example, a tokeniser for a text in English or Spanish, which are normally stored in the common encoding Latin-1 (or ISO 8859-1), would need to be aware that bytes in the (decimal) range 161–191 in Latin-1 represent punctuation marks and other symbols (such as '¡', '¿', '£', and '©'); tokenisation rules would be required to handle each symbol (and thus its byte code) appropriately for that language. However, the same byte range 161–191 in the common Thai alphabet encoding TIS620 corresponds to a set of Thai consonants, which would naturally be treated differently from punctuation or other symbols. A completely different set of tokenisation rules would be necessary to successfully tokenise a Thai text in TIS620 owing to the use of overlapping character ranges.[2]

Determining characters in two-byte encodings involves locating pairs of bytes representing a single character. This process can be complicated by the tokenisation equivalent of code-switching, in which characters from many different writing systems occur within the same text. It is very common in texts to encounter multiple writing systems and thus multiple encodings. In Chinese and Japanese newswire texts, which are usually encoded in a two-byte system, one byte (usually Latin-1) letters, spaces, punctuation marks (e.g., periods, quotation marks, and parentheses), and Arabic numerals are commonly interspersed with the Chinese and Japanese characters. Such texts also frequently contain Latin-1 SGML headers.

---

[1] The Unicode standard (Consortium, 1996) specifies a single 2-byte encoding system that includes 38,885 distinct coded characters derived from 25 supported scripts. However, until Unicode becomes universally accepted and implemented, the many different character sets in use will continue to be a problem.

[2] Actually, due to the nature of written Thai, there are more serious issues than character set encoding which prevent the use of an English tokeniser on Thai texts, and we will discuss these issues in Sec. 3.B.

C. Application Dependence

Although word and sentence segmentation are necessary, in reality, there is no absolute definition for what constitutes a word or a sentence. Both are relatively arbitrary distinctions that vary greatly across written languages. However, for the purposes of computational linguistics we need to define exactly what we need for further processing; usually, the language and task at hand determine the necessary conventions. For example, the English words *I am* are frequently contracted to *I'm*, and a tokeniser frequently expands the contraction to recover the essential grammatical features of the pronoun and verb. A tokeniser that does not expand this contraction to the component words would pass the single token *I'm* to later processing stages. Unless these processors, which may include morphological analysers, part-of-speech taggers, lexical lookup routines, or parsers, were aware of both the contracted and uncontracted forms, the token may be treated as an unknown word.

Another example of the dependence of tokenisation output on later-processing stages is the treatment of the English possessive *'s* in various tagged corpora.[3] In the Brown corpus (Francis and Kucera, 1982), the word *governor's* is considered one token and is tagged as a possessive noun. In the Susanne corpus (Sampson, 1995), on the other hand, the same word is treated as two tokens, *governor* and *'s*, tagged singular noun and possessive, respectively.

Because of this essential dependence, the tasks of word and sentence segmentation overlap with the techniques discussed in other chapters in this handbook: *Lexical Analysis* in Chap. 3, *Parsing Techniques* in Chap. 4, and *Semantic Analysis* in Chap. 5, as well as the practical applications discussed in later chapters.

D. Corpus Dependence

Until recently, the problem of robustness was rarely addressed by NLP systems, which normally could process only well-formed input conforming to their hand-built grammars. The increasing availability of large corpora in multiple languages that encompass a wide range of data types (e.g., newswire texts, e-mail messages, closed captioning data, optical character recognition [OCR] data, multimedia documents) has required the development of robust NLP approaches, as these corpora frequently contain misspellings, erratic punctuation and spacing, and other irregular features. It has become increasingly clear that algorithms that rely on input texts to be well formed are much less successful on these different types of texts.

Similarly, algorithms that expect a corpus to follow a set of conventions for a written language are frequently not robust enough to handle a variety of corpora. It is notoriously difficult to prescribe rules governing the use of a written language; it is even more difficult to get people to "follow the rules." This is in large part due to the nature of written language, in which the conventions are determined by publishing houses and national academies and are arbitrarily subject to change.[4] So, although punctuation roughly corresponds to the use of suprasegmental features in spoken

[3] This example is taken from Grefenstette and Tapanainen (1994).

[4] This is evidenced by the numerous spelling "reforms" that have taken place in various languages, most recently in the German language, or the attempts by governments (such as the French) to "purify" the language or to outlaw foreign words.

language, reliance on well-formed sentences delimited by predictable punctuation can be very problematic. In many corpora, traditional prescriptive rules are commonly ignored. This fact is particularly important to our discussion of both word and sentence segmentation, which largely depend on the regularity of spacing and punctuation. Most existing segmentation algorithms for natural languages are both language-specific and corpus-dependent, developed to handle the predictable ambiguities in a well-formed text. Depending on the origin and purpose of a text, capitalization and punctuation rules may be followed very closely (as in most works of literature), erratically (as in various newspaper texts), or not at all (as in e-mail messages or closed captioning). For example, an area of current research is summarization, filtering, and sorting techniques for e-mail messages and usenet news articles. Corpora containing e-mail and usenet articles can be ill-formed, such as Example {1}, an actual posting to a usenet newsgroup, which shows the erratic use of capitalization and punctuation, "creative" spelling, and domain-specific terminology inherent in such texts.

ive just loaded pcl onto my akcl. when i do an 'in-package' to load pcl, ill get the prompt but im not able to use functions like defclass, etc... is there womething basic im missing or am i just left hanging, twisting in the breeze? {1}

Many online corpora, such as those from OCR or handwriting recognition, contain substitutions, insertions, and deletions of characters and words, which affect both tokenisation and sentence segmentation. Example {2} shows a line taken from a corpus of OCR data in which large portions of several sentences (including the punctuation) were clearly elided by the OCR algorithm. In this extreme case, even accurate sentence segmentation would not allow for full processing of the partial sentences that remain.

newsprint. Furthermore, shoe presses have Using rock for granite roll Two years ago we reported on advances in {2}

Robust text segmentation algorithms designed for use with such corpora must, therefore, have the capability to handle the range of irregularities that distinguish these texts from well-formed newswire documents frequently processed.

E. Evaluation of Text Segmentation Algorithms

Because of the dependencies discussed in the foregoing, evaluation and comparison of text segmentation algorithms is very difficult. Owing to the variety of corpora for which the algorithms are developed, an algorithm that performs very well on a specific corpus may not be successful on another corpus. Certainly, an algorithm fine-tuned for a particular language will most likely be completely inadequate for processing another language. Nevertheless, evaluating algorithms provides important information about their efficacy, and there are common measures of performance for both tokenisation and sentence segmentation.

The performance of word segmentation algorithms is usually measured using *recall* and *precision*, where recall is defined as the percent of words in the manually segmented text identified by the segmentation algorithm, and precision is defined as the percentage of words returned by the algorithm that also occurred in the hand-segmented text in the same position. For a language such as English, in which a great

deal of the initial tokenisation can be done by relying on whitespace, tokenisation is rarely scored. However, for unsegmented languages (see Sec. 3.B), evaluation of word segmentation is critical for improving all aspects of NLP systems.

A sentence segmentation algorithm's performance is usually reported in terms of a single score equal to the number of punctuation marks correctly classified divided by the total number of punctuation marks. It is also possible to evaluate sentence segmentation algorithms using recall and precision, given the numbers of **false positives**, punctuation marks erroneously labeled as a sentence boundary, and **false negatives**, actual sentence boundaries not labeled as such.

3. TOKENISATION

Section 2 discussed the many challenges inherent in segmenting freely occurring text. In this section we will focus on the specific technical issues that arise in tokenisation.

Tokenisation is well established and well understood for artificial languages such as programming languages.[5] However, such artificial languages can be strictly defined to eliminate lexical and structural ambiguities; we do not have this luxury with natural languages, in which the same character can serve many different purposes and in which the syntax is not strictly defined. Many factors can affect the difficulty of tokenising a particular natural language. One fundamental difference exists between tokenisation approaches for **space-delimited** languages and approaches for **unsegmented** languages. In space-delimited languages, such as most European languages, some word boundaries are indicated by the insertion of whitespace. The character sequences delimited are not necessarily the tokens required for further processing, however, so there are still many issues to resolve in tokenisation. In unsegmented languages, such as Chinese and Thai, words are written in succession with no indication of word boundaries. Tokenisation of unsegmented languages, therefore, requires additional lexical and morphological information.

In both unsegmented and space-delimited languages, the specific challenges posed by tokenisation are largely dependent on both the writing system (logographic, syllabic, or alphabetic, as discussed in Sec. 2.A) and the typographical structure of the words. There are three main categories into which word structures can be placed,[6] and each category exists in both unsegmented and space-delimited writing systems. The morphology of words in a language can be **isolating**, in which words do not divide into smaller units; **agglutinating** (or **agglutinative**), in which words divide into smaller units (morphemes) with clear boundaries between the morphemes; or **inflectional**, in which the boundaries between morphemes are not clear and the component morphemes can express more than one grammatical meaning. Although individual languages show tendencies toward one specific type (e.g., Mandarin Chinese is predominantly isolating, Japanese is strongly agglutinative, and Latin is largely inflectional), most languages exhibit traces of all three.[7]

---

[5] For a thorough introduction to the basic techniques of tokenisation in programming languages, see Aho et al. (1986).

[6] This classification comes from Comrie et al. (1996) and Crystal (1987).

[7] A fourth typological classification frequently studied by linguists, **polysynthetic**, can be considered an extreme case of agglutinative, where several morphemes are put together to form complex words that can function as a whole sentence. Chukchi and Inuit are examples of polysynthetic languages.

Because the techniques used in tokenising space-deliminated languages are very different from those used in tokenising unsegmented languages, we will discuss the techniques separately, in Sects. 3.A and 3.B, respectively.

A. Tokenisation in Space-Delimited Languages

In many alphabetic writing systems, including those that use the Latin alphabet, words are separated by whitespace. Yet even in a well-formed corpus of sentences, there are many issues to resolve in tokenisation. Most tokenisation ambiguity exists among uses of punctuation marks, such as periods, commas, quotation marks, apostrophes, and hyphens, since the same punctuation mark can serve many different functions in a single sentence, let alone a single text. Consider example sentence {3} from the *Wall Street Journal* (1988).

> *Clairson International Corp. said it expects to report a net loss for its second quarter ended March 26 and doesn't expect to meet analysts' profit estimates of $3.9 to $4 million, or 76 cents a share to 79 cents a share, for its year ending Sept. 24.* {3}

This sentence has several items of interest that are common for Latinate, alphabetic, space-delimited languages. First, it uses periods in three different ways—within numbers as a decimal point (*$3.9*), to mark abbreviations (*Corp.* and *Sept.*), and to mark the end of the sentence, in which the period following the number *24* is not a decimal point. The sentence uses apostrophes in two ways—to mark the genitive case (where the apostrophe denotes possession) in *analysts'* and to show contractions (places where letters have been left out of words) in *doesn't*. The tokeniser must thus be aware of the uses of punctuation marks and be able to determine when a punctuation mark is part of another token and when it is a separate token.

In addition to resolving these cases, we must make tokenisation decisions about a phrase such as *76 cents a share*, which on the surface consists of four tokens. However, when used adjectivally such as in the phrase *a 76-cents-a-share dividend*, it is normally hyphenated and appears as one. The semantic content is the same despite the orthographic differences, so it makes sense to treat the two identically, as the same number of tokens. Similarly, we must decide whether to treat the phrase *$3.9 to $4 million* differently than if it had been written *3.9 to 4 million dollars* or *$3,900,000 to $4,000,000*. We will discuss these ambiguities and other issues in the following sections.

A logical initial tokenisation of a space-delimited language would be to consider as a separate token any sequence of characters preceded and followed by space. This successfully tokenises words that are a sequence of alphabetic characters, but does not take into account punctuation characters. Frequently, characters such as commas, semicolons, and periods, should be treated as separate tokens, although they are not preceded by whitespace (such as with the comma after *$4 million* in Example {3}). Additionally, many texts contain certain classes of character sequences that should be filtered out before actual tokenisation; these include existing markup and headers (including SGML markup), extra whitespace, and extraneous control characters.

1. Tokenising Punctuation

Although punctuation characters are usually treated as separate tokens, there are many cases when they should be "attached" to another token. The specific cases vary from one language to the next, and the specific treatment of the punctuation characters need to be enumerated within the tokeniser for each language. In this section we will give examples of English tokenisation.

Abbreviations are used in written language to denote the shortened form of a word. In many cases abbreviations are written as a sequence of characters terminated with a period. For this reason, recognizing abbreviations is essential for both tokenisation and sentence segmentation, because abbreviations can occur at the end of a sentence, in which case the period serves two purposes. Compiling a list of abbreviations can help in recognizing them, but abbreviations are productive, and it is not possible to compile an exhaustive list of all abbreviations in any language. Additionally, many abbreviations can also occur as words elsewhere in a text (e.g., the word *Mass* is also the abbreviation for *Massachusetts*). An abbreviation can also represent several different words, as is the case for *St.* which can stand for *Saint*, *Street*, or *State*. However, as *Saint* it is less likely to occur at a sentence boundary than *Street* or *State*. Examples {4} and {5} from the *Wall Street Journal* (1991 and 1987, respectively) demonstrate the difficulties produced by such ambiguous cases, where the same abbreviation can represent different words and can occur both within and at the end of a sentence.

> *The contemporary viewer may simply ogle the vast wooded vistas rising up from the Saguenay River and Lac St. Jean, standing in for the St. Lawrence River.* {4}

> *The firm said it plans to sublease its current headquarters at 55 Water St. A spokesman declined to elaborate.* {5}

Recognizing an abbreviation is thus not sufficient for complete tokenisation, and we will discuss abbreviations at sentence boundaries fully in Sec. 4.B.

Quotation marks and apostrophes (" ' ") are a major source of tokenisation ambiguity. Usually, single and double quotes indicate a quoted passage, and the extent of the tokenisation decision is to determine whether they open or close the passage. In many character sets, single quote and apostrophe are the same character; therefore, it is not always possible to immediately determine if the single quotation mark closes a quoted passage, or serves another purpose as an apostrophe. In addition, as discussed in Sec. 2.B, quotation marks are also commonly used when "romanizing" writing systems, in which umlauts are replaced by a double quotation mark and accents are denoted by a single quotation mark or apostrophe.

The apostrophe is a very ambiguous character. In English, the main uses of apostrophes are to mark the genitive form of a noun, to mark contractions, and to mark certain plural forms. In the genitive case, some applications require a separate token while some require a single token, as discussed in Sec. 2.C. How to treat the genitive case is important, as in other languages, the possessive form of a word is not marked with an apostrophe and cannot be as readily recognized. In German, for example, the possessive form of a noun is usually formed by adding the letter *s* to the word, without an apostrophe, as in *Peters Kopf* (*Peter's head*). However, in modern

(informal) usage in German, *Peter's Kopf* would also be common; the apostrophe is also frequently omitted in modern (informal) English such that *Peters head* is a possible construction. Further, in English, *'s* also serves as a contraction for the verb *is*, as in *he's*, *it's*, and *she's*, as well as the plural form of some words, such as *I.D.'s* or *1980's* (although the apostrophe is also frequently omitted from such plurals). The tokenisation decision in these cases is context-dependent and is closely tied to syntactic analysis.

In the case of apostrophe as contraction, tokenisation may require the expansion of the word to eliminate the apostrophe, but the cases for which this is necessary are very language-dependent. The English contraction *I'm* could be tokenised as the two words *I am*, and *we've* could become *we have*. And depending on the application, *wouldn't* probably would expand to *would not*, although it has been argued by Zwicky and Pullum (1981) that the ending *n't* is an inflectional marker, rather than a clitic. Written French contains a completely different set of contractions, including contracted articles (*l'homme, c'etait*), as well as contracted pronouns (*j'ai, je l'ai*) and other forms such as *n'y, qu'ils, d'ailleurs*, and *aujourd'hui*. Clearly, recognizing the contractions to expand requires knowledge of the language, and the specific contractions to expand, as well as the expanded forms, must be enumerated. All other word-internal apostrophes are treated as a part of the token and not expanded, which allows the proper tokenisation of multiply contracted words such as *fo'c's'le* and *Pudd'n'head* as single words. In addition, because contractions are not always demarcated with apostrophes, as in the French *du*, which is a contraction of *de la*, or the Spanish del, contraction of *de el*, other words to expand must also be listed in the tokeniser.

2. Multipart Words

To different degrees, many written languages contain space-delimited words composed of multiple units, each expressing a particular grammatical meaning. For example, the single Turkish word *çöplüklerimizdekilerdenmiydi* means "was it from those that were in our garbate cans?"[8] This type of construction is particularly common in strongly agglutinative languages such as Swahili, Quechua, and most Altaic languages. It is also common in languages such as German, where noun–noun (*Lebensversicherung*, life insurance), adverb–noun (*Nichtraucher*, nonsmoker), and preposition–noun (*Nachkriegszeit*, postwar period) compounding are all possible. In fact, although it is not an agglutinative language, German compounding can be quite complex, as in *Feuerundlebensversicherung* (fire and life insurance) or *Kundenzufriedenheitsabfragen* (customer satisfaction survey).

To some extent, agglutinating constructions are present in nearly all languages, although this compounding can be marked by hyphenation, in which the use of hyphens can create a single word with multiple grammatical parts. In English it is commonly used to create single-token words such as *end-of-line*, as well as multi-token words, such as *Boston-based*. As with the apostrophe, the use of the hyphen is not uniform; for example, hyphen usage varies greatly between British and American English, as well as between different languages. However, as with apostrophes as

[8] This example is from Hankamer (1986).

contractions, many common language-specific uses of hyphens can be enumerated in the tokeniser.

Many languages use the hyphen to create essential grammatical structures. In French, for example, hyphenated compounds such as *va-t-il*, *c'est-à-dire*, and *celui-ci* need to be expanded during tokenisation, to recover necessary grammatical features of the sentence. In these cases, the tokeniser needs to contain an enumerated list of structures to be expanded, as with the contractions, discussed previously.

Another tokenisation difficulty involving hyphens stems from the practice, common in traditional typesetting, of using hyphens at the ends of lines to break a word too long to include on one line. Such end-of-line hyphens can thus occur within words that are not normally hyphenated. Removing these hyphens is necessary during tokenisation, yet it is difficult to distinguish between such incidental hyphenation and cases where naturally hyphenated words happen to occur at a line break. In an attempt to dehyphenate the artificial cases, it is possible to incorrectly remove necessary hyphens. Grefenstette and Tapanainen (1994) found that nearly 5% of the end-of-line hyphens in an English corpus were word-internal hyphens which happened to also occur as end-of-line hyphens.

In tokenising multipart words, such as hyphenated or agglutinative words, whitespace does not provide much useful information to further processing stages. In such cases, the problem of tokenisation is very closely related both to tokenisation in unsegmented languages, discussed in Sec. 3.B of this chapter, and to lexical analysis, discussed in Chap. 3.

3. *Multiword Expressions*

Spacing conventions in written languages do not always correspond to the desired tokenisation. For example, the three-word English expression *in spite of* is, for all intents and purposes, equivalent to the single word *despite*, and both could be treated as a single token. Similarly, many common English expressions, such as *au pair*, *de facto*, and *joie de vivre*, consist of foreign loan words that can be treated as a single token.

Multiword numerical expressions are also commonly identified in the tokenisation stage. Numbers are ubiquitous in all types of texts in every language. For most applications, sequences of digits and certain types of numerical expressions, such as dates and times, money expressions, and percents, can be treated as a single token. Several examples of such phrases can be seen in the foregoing Example {3}: *March 26*, *$3.9 to $4 million*, and *Sept. 24* could each be treated as a single token. Similarly, phrases such as *76 cents a share* and *$3-a-share* convey roughly the same meaning, despite the difference in hyphenation, and the tokeniser should normalize the two phrases to the same number of tokens (either one or four). Tokenising numeric expressions requires knowledge of the syntax of such expressions, as numerical expressions are written differently in different languages. Even within a language or in languages as similar as English and French, major differences exist in the syntax of numeric expressions, in addition to the obvious vocabulary differences. For example, the English date *November 18, 1989* could alternatively appear in English texts as any number of variations, such as *Nov. 18, 1989*, or *18 November 1989*, or *11/18/89*.

Closely related to hyphenation, treatment of multiword expressions is highly language-dependent and application-dependent, but can easily be handled in the

tokenisation stage if necessary. We need to be careful, however, when combining words into a single token. The phrase *no one*, along with *noone* and *no-one*, is a commonly encountered English equivalent for *nobody* and should normally be treated as a single token. However, in a context such as *No one man can do it alone*, it needs to be treated as two words. The same is true of the two-word phrase *can not*, which is not always equivalent to the single word *cannot* or the contraction *can't*.[9] In such cases, it is safer to allow a later process (such as a parser) to make the decision.

4. A Flex Tokeniser

Tokenisation in space-delimited languages can usually be accomplished with a standard lexical analyzer, such as lex (Lesk and Schmidt, 1975) or flex (Nicol, 1993), or the UNIX tools awk, sed, or perl. Figure 1 shows an example of a basic English tokeniser written in flex, a lexical analyzer with a simple syntax that allows the user to write rules in a regular grammar. This flex example contains implementations of several issues discussed in the foregoing.

The first part of the tokeniser in Fig. 1 contains a series of character category definitions. For example, NUMBER_CHARACTER contains characters that may occur within a number, NUMBER_PREFIX contains characters that may occur at the beginning of a numeric expression, DIACRITIC_LETTER contains letters with diacritics (such as ñ), which are common in many European languages and may occur in some English words. Note that some category definitions contain octal byte codes (such as in NUMBER_PREFIX, where \243 and \245 represent the currency symbols for English pounds and Japanese yen) as discussed in Sec. 2.B.

The character categories defined in the first part of the tokeniser are combined in the second part in a series of rules explicitly defining the tokens to be output. The first rule, for example, states that if the tokeniser encounters an apostrophe followed by the letters 'v' and 'e', it should expand the contraction and output the word "have." The next rule states that any other sequence of letters following an apostrophe should (with the apostrophe) be tokenised separately and not expanded. Other rules in Fig. 1 show examples of tokenising multiword expressions (*au pair*), dates (*Jan. 23rd*), numeric expressions (*$4,347.12*), punctuation characters, abbreviations, and hyphenated words. Note that this basic tokeniser is by no means a complete tokeniser for English, but rather intended to show some simple examples of flex implementations of the issues discussed in previous sections.

B. Tokenisation in Unsegmented Languages

The nature of the tokenisation task in unsegmented languages, such as Chinese, Japanese, and Thai, is essentially different from tokenisation in space-delimited languages, such as English. The lack of any spaces between words necessitates a more informed approach than simple lexical analysis, and tools such as flex are not as successful. However, the specific approach to word segmentation for a particular unsegmented language is further limited by the writing system and orthography of the language, and a single general approach has not been developed. In Sec. 3.B.1, we will describe some general algorithms that have been

[9] For example, consider the following sentence: "Why is my soda can not where I left it?"

```
DIGIT                          [0-9]
NUMBER_CHARACTER               [0-9\.\,]
NUMBER_PREFIX                  [\$\243\245]
NUMBER_SUFFIX                  (\%|\242|th|st|rd)
ROMAN_LETTER                   [a-zA-Z]
DIACRITIC_LETTER               [\300-\377]
INTERNAL_CHAR                  [\-\/]
WORD_CHAR                      ({ROMAN_LETTER}|{DIACRITIC_LETTER})
ABBREVIATION                   (mr|dr)
WHITESPACE                     [\040\t\n]
APOSTROPHE                     [\']
UNDEFINED                      [\200-\237]
PERIOD                         [\.]
SINGLE_CHARACTER               [\.\,\;\!\?\:\"\251]
MONTH                          (jan(uary)?|feb(ruary)?)

%%
{APOSTROPHE}(ve)               { printf("have\n", yytext); } ;
{APOSTROPHE}{ROMAN_LETTER}+    { printf("%s\n", yytext); } ;
(au){WHITESPACE}+(pair) { printf("au pair\n", yytext); } ;

{MONTH}{PERIOD}*{WHITESPACE}+{DIGIT}+{NUMBER_SUFFIX}*          |
{MONTH}{PERIOD}*{WHITESPACE}+{DIGIT}+[\,]{WHITESPACE}+{DIGIT}+ |
{NUMBER_PREFIX}*{NUMBER_CHARACTER}+{NUMBER_SUFFIX}*            |
{SINGLE_CHARACTER} | {ABBREVIATION}{PERIOD}                     |
{WORD_CHAR}+({INTERNAL_CHAR}{WORD_CHAR}+)*   { printf("%s\n",yytext); } ;

{WHITESPACE}+|{UNDEFINED}+                                    ;

%%
yywrap() {
  printf("\n");
  return(1);
```

Fig. 1 An (incomplete) basic flex tokeniser for English.

applied to the problem to obtain an initial approximation for a variety of languages. In Sec. 3.B.2, we will give details of some successful approaches to Chinese segmentation, and in Sec. 3.B.3, we will describe some approaches that have been applied to other languages.

1. Common Approaches

An extensive word list combined with an informed segmentation algorithm can help achieve a certain degree of accuracy in word segmentation, but the greatest barrier to accurate word segmentation is in recognizing unknown words, words not in the lexicon of the segmenter. This problem is dependent both on the source of the lexicon as well as the correspondence (in vocabulary) between the text in question and the lexicon. Wu and Fung (1994) reported that segmentation accuracy is significantly higher when the lexicon is constructed using the same type of corpus as the corpus on which it is tested.

Another obstacle to high-accuracy word segmentation is that there are no widely accepted guidelines for what constitutes a word; therefore, there is no agreement on how to "correctly" segment a text in an unsegmented language. Native speakers of a language do not always agree about the "correct" segmentation, and the same text could be segmented into several very different (and equally correct) sets of words by different native speakers. A simple example from English would be the hyphenated phrase *Boston-based.* If asked to "segment" this phrase into words, some native English speakers might say *Boston-based* is a single word and some might say *Boston* and *based* are two separate words; in this latter case there might also be disagreement about whether the hyphen "belongs" to one of the two words (and to which one) or whether it is a "word" by itself. Disagreement by native speakers of Chinese is much more prevalent; in fact, Sproat et al. (1996) give empirical results showing that native speakers of Chinese frequently agree on the correct segmentation less than 70% of the time. Such ambiguity in the definition of what constitutes a word makes it difficult to evaluate segmentation algorithms that follow different conventions, as it is nearly impossible to construct a "gold standard" against which to directly compare results.

A simple word segmentation algorithm is to consider each character a distinct word. This is practical for Chinese because the average word length is very short (usually between one and two characters, depending on the corpus[10]), and actual words can be recognized with this algorithm. Although it does not assist in tasks such as parsing, part-of-speech tagging, or text-to-speech systems (see Sproat et al., 1996), the character-as-word segmentation algorithm has been used to obtain good performance in Chinese information retrieval (Buckley et al., 1996), a task in which the words in a text play a major role in indexing.

A very common approach to word segmentation is to use a variation of the *maximum matching algorithm*, frequently referred to as the *greedy algorithm.* The greedy algorithm starts at the first character in a text and, using a word list for the language being segmented, attempts to find the longest word in the list starting with that character. If a word is found, the maximum-matching algorithm marks a boundary at the end of the longest word, then begins the same longest match search starting at the character following the match. If no match is found in the word list, the greedy algorithm simply segments that character as a word (as in the foregoing character-as-word algorithm) and begins the search starting at the next character. A variation of the greedy algorithm segments a sequence of unmatched characters as a single word; this variant is more likely to be successful in writing systems with longer average word lengths. In this manner, an initial segmentation can be obtained that is more informed than a simple character-as-word approach. The success of this algorithm is largely dependent on the word list.

As a demonstration of the application of the character-as-word and greedy algorithms, consider an example of "desegmented" English, in which all the white space has been removed: the desegmented version of the phrase, *the table down there*, would thus be *thetabledownthere.* Applying the character-as-word algorithm would result in the useless sequence of tokens *t h e t a b l e d o w n t h e r e*, which is why this algorithm only makes sense for languages such as Chinese. Applying the greedy

[10] As many as 95% of Chinese words consist of one or two characters, according to Fung and Wu (1994).

algorithm with a "perfect" word list containing all known English words would first identify the word *theta*, since that is the longest sequence of letters starting at the initial *t* which forms an actual word. Starting at the *b* following *theta*, the algorithm would then identify *bled* as the maximum match. Continuing in this manner, *thetabledownthere* would be segmented by the greedy algorithm as *theta bled own there*.

A variant of the maximum matching algorithm is the *reverse maximum matching* algorithm, in which the matching proceeds from the end of the string of characters, rather than the beginning. In the foregoing example, *thetabledownthere* would be correctly segmented as *the table down there* by the reverse maximum matching algorithm. Greedy matching from the beginning and the end of the string of characters enables an algorithm such as *forward–backward matching*, in which the results are compared and the segmentation optimized based on the two results. In addition to simple greedy matching, it is possible to encode language-specific heuristics to refine the matching as it progress.

2. Chinese Segmentation

The Chinese writing system is frequently described as logographic, although it is not entirely so because each character (known as *Hanzi*) does not always represent a single word. It has also been classified as morphosyllabic (DeFrancis, 1984), in that each Hanzi represents both a single lexical (and semantic) morpheme as well as a single phonological syllable. Regardless of its classification, the Chinese writing system has been the focus of a great deal of computational linguistics research in recent years.

Most previous work[11] in Chinese segmentation falls into one of three categories: statistical approaches, lexical rule-based approaches, and hybrid approaches that use both statistical and lexical information. Statistical approaches use data, such as the mutual information between characters, compiled from a training corpus, to determine which characters are most likely to form words. Lexical approaches use manually encoded features about the language, such as syntactic and semantic information, common phrasal structures, and morphological rules, to refine the segmentation. The hybrid approaches combine information from both statistical and lexical sources. Sproat et al. (1996) describe such an approach that uses a weighted finite-state transducer to identify both dictionary entries as well as unknown words derived by productive lexical processes. Palmer (1997) also describes a hybrid statistical–lexical approach in which the segmentation is incrementally improved by a trainable sequence of transformation rules.

3. Other Segmentation Algorithms

According to Comrie et al. (1996), the majority of all written languages use an alphabetic or syllabic system. Common unsegmented alphabetic and syllabic languages are Thai, Balinese, Javenese, and Khmer. Although such writing systems have fewer characters, they also have longer words. Localized optimization is thus not as practical as in Chinese segmentation. The richer morphology of such lan-

[11] A detailed treatment of Chinese word segmentation is beyond the scope of this chapter. Much of our summary is taken from Sproat et al. (1996). For a comprehensive recent summary of work in Chinese segmentation, we also recommend Wu and Tseng (1993).

guages often allows initial segmentations based on lists of words, names, and affixes, usually using some variation of the maximum-matching algorithm. Successful high-accuracy segmentation requires a thorough knowledge of the lexical and morphological features of the language. Very little research has been published on this type of word segmentation, but a recent discussion can be found in Kawtrakul et al. (1996), which describes a robust Thai segmenter and morphological analyzer.

Languages such as Japanese and Korean have writing systems that incorporate alphabetic, syllabic, and logographic symbols. Modern Japanese texts, for example, frequently consist of many different writing systems: Kanji (Chinese symbols), hiragana (a syllabary for grammatical markers and for words of Japanese origin), katakana (a syllabary for words of foreign origin), romanji (words written in the Roman alphabet), Arabic numerals, and various punctuation symbols. In some ways, the multiple character sets make tokenisation easier, as transitions between character sets give valuable information about word boundaries. However, character set transitions are not enough, for a single word may contain characters from multiple character sets, such as inflected verbs, which can contain a Kanji base and katakana inflectional ending. Company names also frequently contain a mix of Kanji and romanji. To some extent, Japanese can be segmented using the same techniques developed for Chinese. For example, Nagata (1994) describes an algorithm for Japanese segmentation similar to that used for Chinese segmentation by Sproat et al. (1996).

4. SENTENCE SEGMENTATION

Sentences in most written languages are delimited by punctuation marks, yet the specific usage rules for punctuation are not always coherently defined. Even when a strict set of rules exists, the adherence to the rules can vary dramatically, based on the origin of the text source and the type of text. Additionally, in different languages, sentences and subsentences are frequently delimited by different punctuation marks. Successful sentence segmentation for a given language thus requires an understanding of the various uses of punctuation characters in that language. In most languages, the problem of sentence segmentation reduces to disambiguating all instances of punctuation characters that may delimit sentences. The scope of this problem varies greatly by language, as does the number of different punctuation marks that need to be considered.

Written languages that do not use many punctuation marks present a very difficult challenge in recognizing sentence boundaries. Thai, for one, does not use a period (or any other punctuation mark) to mark sentence boundaries. A space is sometimes used at sentence breaks, but very often the space is indistinguishable from the carriage return, or there is no separation between sentences. Spaces are sometimes also used to separate phrases or clauses, where commas would be used in English, but this is also unreliable. In cases such as written Thai for which punctuation gives no reliable information about sentence boundaries, locating sentence boundaries is best treated as a special class of locating word boundaries.

Even languages with relatively rich punctuation systems, such as English, present surprising problems. Recognizing boundaries in such a written language involves determining the roles of all punctuation marks that can denote sentence boundaries: periods, question marks, exclamation points, and sometimes semicolons,

colons, dashes, and commas. In large document collections, each of these punctuation marks can serve several different purposes, in addition to marking sentence boundaries. A period, for example, can denote a decimal point or a thousands marker, an abbreviation, the end of a sentence, or even an abbreviation at the end of a sentence. Ellipsis (a series of periods [...]), can occur both within sentences and at sentence boundaries. Exclamation points and question marks can occur at the end of a sentence, but also within quotation marks or parentheses (really!) or even (albeit infrequently) within a word, such as in the band name *Therapy?* and the language name *!Xũ*. However, conventions for the use of these two punctuation marks also vary by language; in Spanish, both can be unambiguously recognized as sentence delimiters by the presence of '¡' or '¿' at the start of the sentence. In this section we will introduce the challenges posed by the range of corpora available and the variety of techniques that have been successfully applied to this problem and discuss their advantages and disadvantages.

A. Sentence Boundary Punctuation

Just as the definition of what constitutes a sentence is rather arbitrary, the use of certain punctuation marks to separate sentences depends largely on an author's adherence to changeable, and frequently ignored, conventions. In most NLP applications, the only sentence boundary punctuation marks considered are the period, question mark, and exclamation point, and the definition of what constitutes a sentence is limited to the *text-sentence*, as defined by Nunberg (1990). However, grammatical sentences can be delimited by many other punctuation marks, and restricting sentence boundary punctuation to these three can cause an application to overlook many meaningful sentences or unnecessarily complicate processing by allowing only longer, complex sentences. Consider Examples {6} and {7}, two English sentences that convey exactly the same meaning; yet, by the traditional definitions, the first would be classified as two sentences, the second as just one. The semicolon in Example {7} could likewise be replaced by a comma or a dash, retain the same meaning, but still be considered a single sentence. Replacing the semicolon with a colon is also possible, although the resulting meaning would be slightly different.

Here is a sentence. Here is another. {6}

Here is a sentence; here is another. {7}

The distinction is particularly important for an application such as part-of-speech tagging. Many taggers, such as the one described in Cutting et al. (1991), seek to optimize a tag sequence for a sentence, with the locations of sentence boundaries being provided to the tagger at the outset. The optimal sequence will usually be different depending on the definition of sentence boundary and how the tagger treats "sentence-internal" punctuation.

For an even more striking example of the problem of restricting sentence boundary punctuation, consider Example {8}, from Lewis Carroll's *Alice in Wonderland*, in which *.!?* are completely inadequate for segmenting the meaningful units of the passage:

There was nothing so VERY remarkable in that; nor did Alice think it so VERY much out of the way to hear the Rabbit say to itself, 'Oh dear! Oh dear! I shall be late!' (when she thought it over afterwards, it occurred to her that she ought to have wondered at this, but at the time it all seemed quite natural); but when the Rabbit actually TOOK A WATCH OUT OF ITS WAISTCOAT-POCKET, and looked at it, and then hurried on, Alice started to her feet, for it flashed across her mind that she had never before seen a rabbit with either a waistcoat-pocket, or a watch to take out of it, and burning with curiosity, she ran across the field after it, and fortunately was just in time to see it pop down a large rabbit-hole under the hedge. {8}

This example contains a single period at the end and three exclamation points within a quoted passage. However, if the semicolon and comma were allowed to end sentences, the example could be decomposed into as many as ten grammatical sentences. This decomposition could greatly assist in nearly all NLP tasks, because long sentences are more likely to produce (and compound) errors of analysis. For example, parsers consistently have difficulty with sentences longer than 15–25 words, and it is highly unlikely that any parser could ever successfully analyze this example in its entirety.

In addition to determining which punctuation marks delimit sentences, the sentence in parentheses as well as the quoted sentences *'Oh dear! Oh dear! I shall be late!'* suggest the possibility of a further decomposition of the sentence boundary problem into types of sentence boundaries, one of which would be "embedded sentence boundary." Treating embedded sentences and their punctuation differently could assist in the processing of the entire text-sentence. Of course, multiple levels of embedding would be possible, as in Example {9}, taken from Adams (1972). In this example, the main sentence contains an embedded sentence (delimited by dashes), and this embedded sentence also contains an embedded quoted sentence.

The holes certainly were rough—"Just right for a lot of vagabonds like us," said Bigwig—but the exhausted and those who wander in strange country are not particular about their quarters. {9}

It should be clear from these examples that true sentence segmentation, including treatment of embedded sentences, can be achieved only through an approach which integrates segmentation with parsing. Unfortunately there has been little research in integrating the two; in fact, little research in computational linguistics has focused on the role of punctuation in written language.[12] With the availability of a wide range of corpora and the resulting need for robust approaches to natural language processing, the problem of sentence segmentation has recently received a lot of attention. Unfortunately, nearly all published research in this area has focused on the problem of sentence boundary detection in English, and all this work has focused exclusively on disambiguating the occurrences of period, exclamation point, and question mark. A promising development in this area is the recent focus on trainable approaches to sentence segmentation, which we will discuss in Sec. 4.D.

[12] A notable exception is Nunberg (1990).

These new methods, which can be adapted to different languages and different text genres, should make a tighter coupling of sentence segmentation and parsing possible. While the remainder of this chapter will focus on published work that deals with the segmentation of a text into text-sentences, the foregoing discussion of sentence punctuation indicates the application of trainable techniques to broader problems may be possible.

B. The Importance of Context

In any attempt to disambiguate the various uses of punctuation marks, whether in text-sentences or embedded sentences, some amount of the context in which the punctuation occurs is essential. In many cases, the essential context can be limited to the character immediately following the punctuation mark. When analyzing well-formed English documents, for example, it is tempting to believe that sentence boundary detection is simply a matter of finding a period followed by one or more spaces followed by a word beginning with a capital letter, perhaps also with quotation marks before or after the space. A single rule[13] to represent this pattern would be:

```
IF (right context = period + space + capital letter
            OR period + quote + space + capital letter
            OR period + space + quote + capital letter)
THEN sentence boundary
```

Indeed, in some corpora (e.g., literary texts) this single pattern accounts for almost all sentence boundaries. In *The Call of the Wild* by Jack London, for example, which has 1640 periods as sentence boundaries, this single rule will correctly identify 1608 boundaries (98%). However, the results are different in journalistic texts, such as the *Wall Street Journal (WSJ)*. In a small corpus of the *WSJ* from 1989 that has 16,466 periods as sentence boundaries, this simple rule would detect only 14,562 (88.4%), while producing 2,900 **false positives**, placing a boundary where one does not exist.

Most of the errors resulting from this simple rule are when the period occurs immediately after an abbreviation. Expanding the context to consider the word preceding the period is thus a logical step. An improved rule would be:

```
IF ((right context = period + space + capital letter
            OR period + quote + space + capital letter
            OR period + space + quote + capital letter)
  AND (left context != abbreviation))
THEN sentence boundary
```

This can produce mixed results, for the use of abbreviations in a text depends on the particular text and text genre. The new rule improves performance on *The Call of the Wild* to 98.4% by eliminating 5 false positives (previously introduced by the phrase "St. Bernard" within a sentence). On the *WSJ* corpus, this new rule also eliminates all but 283 of the false positives introduced by the first rule. However, this rule also introduces 713 **false negatives**, erasing boundaries where they were pre-

[13] Parts of this discussion originally appeared in Bayer et al. (1998).

viously correctly placed, yet still improving the overall score. Recognizing an abbreviation, therefore, is not sufficient to disambiguate the period, because we also must determine if the abbreviation occurs at the end of a sentence.

The difficulty of disambiguating abbreviation-periods can vary depending on the corpus. Liberman and Church (1992) report that 47% of the periods in a *Wall Street Journal* corpus denote abbreviations, compared with only 10% in the Brown corpus (Francis and Kucera, 1982) as reported by Riley (1989). In contrast, Müller (1980) reports abbreviation-period statistics ranging from 54.7 to 92.8% within a corpus of English scientific abstracts. Such a range of figures suggests the need for a more informed treatment of the context that considers more than just the word preceding or following the punctuation mark. In difficult cases, such as an abbreviation that can occur at the end of a sentence, three or more words preceding and following must be considered. This is true in the following examples of "garden path sentence boundaries," the first consisting of a single sentence, the other of two sentences.

Two high-ranking positions were filled Friday by Penn St. University President Graham Spanier. {10}

Two high-ranking positions were filled Friday at Penn St. University President Graham Spanier announced the appointments. {11}

Many contextual factors assist sentence segmentation in difficult cases. These contextual factors include:

- **Case distinctions**—In languages and corpora where both upper-case and lower-case letters are consistently used, whether a word is capitalized provides information about sentence boundaries.
- **Part of speech**—Palmer and Hearst (1997) showed that the parts of speech of the words within three tokens of the punctuation mark can assist in sentence segmentation. Their results indicate that even an estimate of the *possible* parts of speech can produce good results.
- **Word length**—Riley (1989) used the length of the words before and after a period as one contextual feature.
- **Lexical endings**—Müller et al. (1980) used morphological analysis to recognize suffixes and thereby filter out words that were not likely to be abbreviations. The analysis made it possible to identify words that were not otherwise present in the extensive word lists used to identify abbreviations.
- **Prefixes and suffixes**—Reynar and Ratnaparkhi (1997) used both prefixes and suffixes of the words surrounding the punctuation mark as one contextual feature.
- **Abbreviation classes**—Riley (1989) and Reynar and Ratnaparkhi (1997) further divided abbreviations into categories, such as titles (which are not likely to occur at a sentence boundary) and corporate designators (which are more likely to occur at a boundary).

C. Traditional Rule-Based Approaches

The success of the few simple rules described in the previous section is a major reason sentence segmentation has been frequently overlooked or idealized away. In well-behaved corpora, simple rules relying on regular punctuation, spacing, and capita-

lization can be quickly written, and are usually quite successful. Traditionally, the method widely used for determining sentence boundaries is a regular grammar, usually with limited lookahead. More elaborate implementations include extensive word lists and exception lists to attempt to recognize abbreviations and proper nouns. Such systems are usually developed specifically for a text corpus in a single language and rely on special language-specific word lists; as a result they are not portable to other natural languages without repeating the effort of compiling extensive lists and rewriting rules. Although the regular grammar approach can be successful, it requires a large manual effort to compile the individual rules used to recognize the sentence boundaries. Nevertheless, since rule-based sentence segmentation algorithms can be very successful when an application does deal with well-behaved corpora, we provide a description of these techniques.

An example of a very successful regular–expression-based sentence segmentation algorithm is the text segmentation stage of the Alembic information extraction system (Aberdeen et al., 1995), which was created using the lexical scanner generator flex (Nicol, 1993). The Alembic system uses flex in a preprocess pipeline to perform tokenisation and sentence segmentation at the same time. Various modules in the pipeline attempt to classify all instances of punctuation marks by identifying periods in numbers, date, and time expressions, and abbreviations. The preprocess utilizes a list of 75 abbreviations and a series of over 100 hand-crafted rules and was developed over the course of more than 6 staff months. The Alembic system alone achieved a very high accuracy rate (99.1%) on a large *Wall Street Journal* corpus. However, the performance was improved when integrated with the trainable system Satz, described in Palmer and Hearst (1997) and summarized later in this chapter. In this hybrid system, the rule-based Alembic system was used to disambiguate the relatively unambiguous cases, whereas Satz was used to disambiguate difficult cases such as the five abbreviations *Co.*, *Corp.*, *Ltd.*, *Inc.*, and *U.S.*, which frequently occur in English texts both within sentences and at sentence boundaries. The hybrid system achieved an accuracy of 99.5%, higher than either of the two component systems alone.

D. Robustness and Trainability

Throughout this chapter we have emphasized the need for robustness in NLP systems, and sentence segmentation is no exception. The traditional rule-based systems, which rely on features such as spacing and capitalization, will not be as successful when processing texts where these features are not present, such as in Example {1}. Similarly, some important kinds of text consist solely of upper-case letters; closed captioning (CC) data is an example of such a corpus. In addition to being upper-case-only, CC data also has erratic spelling and punctuation, as can be seen from the following example of CC data from CNN:

THIS IS A DESPERATE ATTEMPT BY THE REPUBLICANS TO SPIN THEIR STORY THAT NOTHING SEAR WHYOUS— SERIOUS HAS BEEN DONE AND TRY TO SAVE THE SPEAKER'S SPEAKERSHIP AND THIS HAS BEEN A SERIOUS PROBLEM FOR THE SPEAKER, HE DID NOT TELL THE TRUTH TO THE COMMITTEE, NUMBER ONE. {12}

The limitations of manually crafted rule-based approaches suggest the need for trainable approaches to sentence segmentation, to allow for variations between languages, applications, and genres. Trainable methods provide a means for addressing the problem of embedded sentence boundaries discussed earlier, as well as the capability of processing a range of corpora and the problems they present, such as erratic spacing, spelling errors, single-case, and OCR errors.

For each punctuation mark to be disambiguated, a typical trainable sentence segmentation algorithm will automatically encode the context using some or all of the features just described. A set of training data, in which the sentence boundaries have been manually labeled, is then used to train a machine-learning algorithm to recognize the salient features in the context. As we describe in the following, machine-learning algorithms that have been used in trainable sentence segmentation systems have included neural networks, decision trees, and maximum entropy calculation.

E. Trainable Algorithms

One of the first published works describing a trainable sentence segmentation algorithm was by Riley (1989). The method described used regression trees (Breiman et al., 1984) to classify periods according to contextual features describing the single word preceding and following the period. These contextual features included word length, punctuation after the period, abbreviation class, case of the word, and the probability of the word occurring at beginning or end of a sentence. Riley's method was trained using 25 million words from the Associated Press (AP) newswire, and he reported an accuracy of 99.8% when tested on the Brown corpus.

Reynar and Ratnaparkhi (1997) described a trainable approach to identifying English sentence boundaries (.!?) that used a statistical maximum entropy model. The system used a system of contextual templates that encoded one word of context preceding and following the punctuation mark, using such features as prefixes, suffixes, and abbreviation class. They also reported success in including an abbreviation list from the training data for use in the disambiguation. The algorithm, trained in less than 30 min on 40,000 manually annotated sentences, achieved a high accuracy rate (98%+) on the same test corpus used by Palmer and Hearst (1997), without requiring specific lexical information, word lists, or any domain-specific information. Although they reported results only for English, they indicated the ease of trainability should allow the algorithm to be used with other Roman alphabet languages, given adequate training data.

Palmer and Hearst (1997) developed a sentence segmentation system called Satz, which used a machine-learning algorithm to disambiguate all occurrences of periods, exclamation points, and question marks. The system defined a contextual feature array for three words preceding and three words following the punctuation mark; the feature array encoded the context as the parts of speech that can be attributed to each word in the context. Using the lexical feature arrays, both a neural network and a decision tree were trained to disambiguate the punctuation marks, and both achieved a high accuracy rate (98–99%) on a large corpus from the *Wall Street Journal*. They also demonstrated the algorithm, which was trainable in as little as one minute and required fewer than one thousand sentences of training data, to be rapidly portable to new languages. They adapted the system to French and German, in each

case achieving a very high accuracy. Additionally, they demonstrated the trainable method to be extremely robust, as it was able to successfully disambiguate single-case texts and OCR data.

Trainable sentence segmentation algorithms, such as these, are clearly a step in the right direction toward enabling robust processing of a variety of texts and languages. Algorithms that offer rapid training while requiring small amounts of training data will permit systems to be retargeted in hours or minutes to new text genres and languages.

5. CONCLUSION

Until recently, the problem of text segmentation was overlooked or idealized away in most NLP systems; tokenisation and sentence segmentation were frequently dismissed as uninteresting "preprocessing" steps. This was possible because most systems were designed to process small texts in a single language. When processing texts in a single language with predictable orthographic conventions, it was possible to create and maintain hand-built algorithms to perform tokenisation and sentence segmentation. However, the recent explosion in availability of large unrestricted corpora in many different languages, and the resultant demand for tools to process such corpora, has forced researchers to examine the many challenges posed by processing unrestricted texts. The result has been a move toward developing robust algorithms that do not depend on the well-formedness of the texts being processed. Many of the hand-built techniques have been replaced by trainable corpus-based approaches that use machine learning to improve their performance.

Although the move toward trainable robust segmentation systems has been very promising, there is still room for improvement of tokenisation and sentence segmentation algorithms. Because errors at the text segmentation stage directly affect all later processing stages, it is essential to completely understand and address the issues involved in tokenisation and sentence segmentation and how they influence further processing. Many of these issues are language-dependent: the complexity of tokenisation and sentence segmentation and the specific implementation decisions depend largely on the language being processed and the characteristics of its writing system. For a corpus in a particular language, the corpus characteristics and the application requirements also affect the design and implementation of tokenisation and sentence segmentation algorithms. Frequently, because text segmentation is not the primary objective of NLP systems, it cannot be thought of as simply an independent "preprocessing" step, but rather, must be tightly integrated with the design and implementation of all other stages of the system.

REFERENCES

Aberdeen J, Burger J, Day D, Hirschman L, Robinson P, Vilain, M. (1995) MITRE: Description of the Alembic system used for MUC-6. Proceedings of the Sixth Message Understanding Conference (MUC-6), Columbia, MD, November 1995.

Adams R. (1972). *Watership Down*. New York: Macmillan Publishing.

Aho AV, Sethi R, Ullman JD. (1986). Compilers, Principles, Techniques, and Tools. Reading, MA: Addison–Wesley.

Bayer S, Aberdeen J, Burger J, Hirschman L, Palmer D, Vilain M. (1998). Theoretical and computational linguistics: toward a mutual understanding. In: Lawler J, Dry, HA, eds. Using Computers in Linguistics. London: Routledge.

Breiman L, Friedman J.H, Olshen R, Stone C.J (1984). Classification and Regression Trees. Belmont, CA: Wadsworth International Group.

Buckley C, Singhal A, Mitra M. (1996). Using query zoning and correlation within SMART: TREC 5. In: Harman DK, ed. Proceedings of the Fifth Text Retrieval Conference (TREC-5), Gaithersburg, MD, November 20–22.

Comrie B, Matthews S, Polinsky M. (1996). The Atlas of Languages. London: Quarto.

Consortium, Unicode. (1996). The Unicode Standard, Version 2.0. Reading, MA: Addison–Wesley.

Crystal D. (1987). The Cambridge Encyclopedia of Language. Cambridge, UK: Cambridge University Press.

Cutting D, Kupiec J, Pedersen J, Sibun P. (1991). A practical part-of-speech tagger. 3rd Conference on Applied Natural Processing, Trento, Italy.

Daniels PT, Bright W. (1996). The World's Writing Systems. New York: Oxford University Press.

DeFrancis J. (1984). The Chinese Language. Honolulu: The University of Hawaii Press.

Nelson FW, Kucera H. (1982). Frequency Analysis of English Usage. New York: Houghton Mifflin.

Fung, P, Wu D. (1994). Statistical augmentation of a Chinese machine-readable dictionary. Proceedings of Second Workshop on Very Large Corpora (WVLC-94).

Grefenstette G, Tapanainen P. (1994). What is a word, What is a sentence? Problems of tokenization. 3rd International Conference on Computational Lexicography (COMPLEX 1994).

Hankamer J. (1986) Finite state morphology and left to right phonology. Proceedings of the Fifth West Coast Conference on Formal Linguistics.

Harman DK. (1996). Proceedings of the Fifth Text Retrieval Conference (TREC-5). Gaithersburg, MD, November 20–22.

Kawtrakul A, Thumkanon C, Jamjanya T, Muangyunnan P, Poolwan K, Inagaki Y. (1996). A gradual refinement model for a robust Thai morphological analyzer. Proceedings of COLING96, Copenhagen, Denmark.

Lawler J, Dry HA. (1998). Using Computers in Linguistics. London: Routledge.

Lesk ME, Schmidt E. (1975). Lex—a lexical analyzer generator. Computing Science Technical Report 39, AT&T Bell Laboratories, Murray Hill, NJ.

Liberman MY, Church KW. (1992). Text analysis and word pronunciation in text-to-speech synthesis. In Furui S, Sondhi MM, eds, Advances in Speech Signal Processing. New York: Marcel Dekker, pp. 791–831.

Müller H, Amerl V, Natalis G. (1980). Worterkennungsverfahren als Grundlage einer Universalmethode zur automatischen Segmentierung von Texten in Sätze. Ein Verfahren zur maschinellen Satzgrenzenbestimmung im Englischen. Sprache und Datenverarbeitung, 1.

Nagata M. (1994). A stochastic Japanese morphological analyzer using a forward-dp backward A* n-best search algorithm. Proceedings of COLING94.

Nicol GT. (1993). Flex—The Lexical Scanner Generator. Cambridge, MA: Free Software Foundation.

Nunberg G. (1990). The Linguistics of Punctuation. CSLI Lecture Notes, Number 18. Center for the Study of Language and Information, Stanford, CA.

Palmer DD. (1997). A trainable rule-based algorithm for word segmentation. Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics (ACL97), Madrid.

Palmer DD, Hearst MA. (1997). Adaptive multilingual sentence boundary disambiguation. Comput Linguist 23:241–267.

Reynar JC, Ratnaparkhi A. (1997). A maximum entropy approach to identifying sentence boundaries. Proceedings of the Fifth ACL Conference on Applied Natural Language Processing, Washington, DC.

Riley MD. (1989). Some applications of tree-based modelling to speech and language indexing. Proceedings of the DARPA Speech and Natural Language Workshop, Morgan Kaufmann, pp 339–352.

Sampson GR. (1995). English for the Computer, New York: Oxford University Press.

Sproat RW, Shih C, Gale W, Chang N. (1996). A stochastic finite-state word-segmentation algorithm for Chinese. Comput Linguist 22:377–404.

Wu D, Fung P. (1994). Improving Chinese tokenization with linguistic filters on statistical lexical acquisition. Proceedings of the Fourth ACL Conference on Applied Natural Language Processing, Stuttgart, Germany.

Wu Z, Tseng G. (1993). Chinese text segmentation for text retrieval: Achievements and problems. J Am Soc Inform Sci, 44:532–542.

Zwicky AM, Pullum GK. (1981). Cliticization vs. inflection: English n't. 1981 Annual Meeting of the Linguistics Society of America.

3

Lexical Analysis

RICHARD SPROAT

AT&T Labs—Research, Florham Park, New Jersey

1. INTRODUCTION

An important component of any natural language system is *lexical analysis*, which can be broadly defined as the determination of lexical features for each of the individual words of a text. Which precise set of features are identified largely depends on the broader application within which the lexical analysis component is functioning. For example, if the lexical analysis component is part of a 'preprocessor' for a part-of-speech tagger or a syntactic parser, then it is likely that the lexical features of most interest will be purely grammatical features, such as part of speech, or perhaps verbal subcategorization features. If the lexical analysis is being performed as part of a machine-translation system, then it is likely that at least some 'semantic' information (such as the gloss in the other language) would be desired. And if the purpose is text-to-speech synthesis (TTS), then the pronunciation of the word will be among the set of features to be extracted.

The simplest model is one in which all words are simply listed along with their lexical features: lexical analysis thus becomes a simple matter of table lookup. It is safe to say that there is probably no natural language for which such a strategy is practical for anything other than a limited task. The reason is that all natural languages exhibit at least some productive lexical processes by which a great many words can be derived. To give a concrete if somewhat standard example, regular verbs in Spanish can typically occur in any of approximately thirty five forms.[1] The

[1] This is quite small by some standards: in Finnish it is possible to derive thousands of forms from a lexical item.

forms are derivable by regular rules, and their grammatical features are perfectly predictable: it is predictable, for example, that the first-person plural subjunctive present of the first conjugation verb *hablar* 'speak' is *hablemos*, composed of the stem *habl-* the present subjunctive theme vowel *-e-* and the first-person plural ending *-mos*; it is similarly predictable that the masculine form of the past participle is *hablado*, formed from the stem *habl-* the theme vowel *-a-*, the participial affix *-d-* and the masculine ending *-o*. Although it would certainly be possible in Spanish to simply list all possible forms of all verbs (say all the verbs that one might find in a large Spanish desk dictionary), this would be both a time-consuming and unnecessary exercise. Much more desirable would be to list a canonical form (or small set of forms) for each verb in the dictionary, and then provide a 'recipe' for producing the full set of extant forms, and their associated features. *Computational morphology* provides mechanisms for implementing such recipes.

The topic of this chapter is morphological analysis as it applies, primarily, to written language. That is, when we speak of analyzing *words* we are speaking of analyzing words as they are written *in the standard orthography for the language in question*. The reason for this restriction is mainly historical, in that nearly all extant morphological analyzers work off written text, rather than, say, the phonetic transcriptions output by a speech recognizer. There is every reason to believe that the techniques that will be described here would apply equally well in the speech recognition domain, but as yet there have been no serious applications of this kind.

Section 2 gives some linguistic background on morphology. Secs. 3, 4, and 5 will be devoted to introducing basic techniques that have been proposed for analyzing morphology. Although many approaches to computational morphology have been proposed, there are only really two that have been at all widespread, and these can be broadly characterized as *finite-state* techniques, and *unification-based* techniques. In computational models of phonology or (orthographic) spelling changes, there has really been only one dominant type of model; namely, the finite-state models. In this chapter I shall focus on these widely used techniques. A broader overview that includes discussion of other techniques in computational morphology can be found Ref. 1.

Section 6 reviews probabilistic approaches to morphology, in particular methods that combine probabilistic information with categorical models. Although there has been relatively little work in this area, it is an area that shows a lot of potential.

The analysis of written words does not provide as tight a delimitation of the topic of this chapter as one might imagine. What is a *written word*? The standard definition that it is any sequence of characters surround by white space or punctuation will not always do. One reason it that there are many multiword entities—*in spite of, by and large, shoot the bull*, and other such—which from at least some points of view should be considered single lexical items. But an even more compelling reason is that there are languages—Chinese, Japanese, and Thai, among others—in which white space is never used to delimit words. As we shall see in Sec. 7, lexical analysis for these languages is inextricably linked with tokenization, the topic of Chap. 2.

Section 8 concludes the chapter with a summary, and a mention of some issues that relate to lexical analysis, but that do not strictly come under the rubric of computational morphology.

2. SOME LINGUISTIC BACKGROUND

Several different theoretical models of morphology have been developed over the years, each with rather specific sets of claims about the nature of morphology, and each with rather specific focuses in terms of the data that are covered by the theory. There is insufficient space here to review the various aspects of work on theoretical morphology that are relevant to computational models. (A fairly thorough review of some aspects of the linguistic underpinnings of computational morphology are given in Chap. 2 of Ref. 1.) However, it is worth presenting a high-level classification of morphological theories, to clarify the intellectual background tacitly assumed by most computational models of word-formation.

Linguistic models of morphology can be broadly classified into two main 'schools.' Following the terminology of Hockett [2], the first could be termed *item-and-arrangement* (IA), the second *item-and-process* (IP). Broadly speaking, IA models treat morphologically complex words as being composed of two or more atomic *morphemes*, each of which contributes some semantic, grammatical, and phonological information to the composite whole. Consider the German noun *Fähigkeit* 'ability,' which is derived from the adjective *fähig* 'able.' An IA analysis of *Fähigkeit* would countenance two morphemes, the base *fähig* and the affix *-keit*, which can be glossed roughly as *-ity*. Crucially, each of these elements—*Fähig* and *-keit*—would be entries in the lexicon, with the first being listed as belonging to the category A(djective) and the second belonging to a category that takes As as a base and forms N(oun)s.

IP models view things differently. While items belonging to major grammatical categories [Ns, As, V(erb)s] are listed in the lexicon, elements such as *-keit* have no existence separate from the morphological rules that introduce them. For example, in the IP-style analysis of Beard [3], there would be a morphological rule that forms abstract nouns out of adjectives. The morphosyntactic effect of the rule would be to change the category of the word from adjective to noun; the semantic effect would be to assign a meaning to the new word along the lines of 'state of being X,' where X is the meaning of the base; and the phonological effect (for certain classes of base) would be to add the affix *-keit*.

Even though IP models are arguably more in line with traditional models of grammar, IA models, on the whole, have been more popular in Generative Linguistics, largely because of the influence of the preceding American Structuralists, such as Bloomfield. IA models include Refs. 4–8. Nonetheless there have also been IP theories developed including Refs. 3, 9–12. However, although it is generally possible to classify a given morphological theory as falling essentially into one of the two camps, it should be borne in mind that there are probably no morphological theories that are 'pure' IA or IP.

Computational models have, on the whole, subscribed more to the IA view than the IP. As we shall see, it is fairly typical to view the problem of morphologically decomposing a word as one of parsing the word into its component morphemes, each of which has its own representation in the system's lexicon. This is not to say that there have not been computational models for which affixes are modeled as rules applying to bases [13], but such systems have definitely been in the minority.

3. COMPUTATIONAL PHONOLOGY

Any discussion of computational morphology must start with a discussion of computational phonology. The reason for this is simple. When morphemes are combined to form a word, often phonological processes apply that change the form of the morphemes involved. There are really only two ways to handle this kind of alternation. One, exemplified by Ref. 14 is to simply 'precompile' out the alternations and use *morphotactic* (see Sec. 4) constraints to ensure that correct forms of morphemes are selected in a given instance. The other, and more widely adopted approach, is to handle at least some of this alternation through the use of phonological models. Such models are the topic of this section.

Consider, for example, the formation of partitive nouns in Finnish. The partitive affix in Finnish has two forms, *-ta*, and *-tä*; the form chosen depends on the final harmony-inducing vowel of the base. Bases whose final harmony-inducing vowel is back take *-ta*; those whose final harmony-inducing vowel is front take *-tä*. The vowels *i* and *e* are not harmony inducing; they are transparent to harmony. Thus in a stem, such as *puhelin* 'telephone,' the last harmony-inducing vowel is *u*, so the form of the partitive affix is *-ta*.

Nominative	**Partitive**	**Gloss**
taivas	taivas + ta	'sky'
puhelin	puhelin + ta	'telephone'
lakeus	lakeut + ta	'plain'
syy	syy + tä	'reason'
lyhyt	lyhyt + tä	'short'
ystävällinen	ystävällinen + tä	'friendly'

This alternation is part of a much more general *vowel harmony* process in Finnish, one which affects a large number of affixes. Within the linguistics literature there have been two basic approaches to dealing with phonological processes such as harmony. One approach, which can be broadly characterized as rule-based ('procedural') posits that such processes are handled by rewrite rules. An example of a traditional string-rewriting rule to handle the Finnish case just described follows:[2]

$$a \rightarrow ä / [ä, ö, y]C^* ([i,e]C^*)^*\_\_\_$$

This rule simply states that an *a* is changed into *ä* when preceded a vowel from the set *ä*, *ö*, *y*, with possible intervening *i*, *e*, and consonants. (This rule is a gross oversimplification of the true Finnish harmony alternation in several respects, but it will do for the present discussion.) A description of the form just given is quite outdated: more up-to-date rule-based analyses would posit a multitiered *autosegmental* representation where harmony features (in this case the feature [—back]) are spread from trigger vowels to the vowels in the affixes, either changing or filling in the features for those affixes found in the lexicon.[3] But the basic principle remains the same, namely that an input form is somehow modified by a rule to produce the correct output form.

---

[2] We assume here a basic familiarity with regular expression syntax.

[3] For a general introduction to such theories, see Ref. 15.

An alternative approach is declarative: one can think of the lexicon as generating all conceivable forms for an input item. So one would generate both, say, *lakeus* + *ta* and *lakeus* + *tä*. The function of the phonology is to select an appropriate form based on constraints. These constraints may be absolute ('hard') in the case of most work in the tradition of so-called *declarative phonology* [e.g., 16, 17]; or they may be ranked 'soft' constraints, as in Optimality Theory [e.g., 18], where the optimal analysis is chosen according to some ranking scheme.

Most work on computational phonology takes crucial advantage of a property of systems of phonological description as those systems are typically used: namely, that they are *regular* in the formal sense that they can be modeled as *regular relations* (if one is talking about rules) or *regular languages* (if one is talking about constraints).

The property of regularity is by no means one that is imposed by the linguistic formalisms themselves, nor is it necessarily imposed by most linguistic theories. That is, in principle, it is possible to write a rule of the form $\epsilon \rightarrow ab / a\_\_b$ which, if allowed to apply an unbounded number of times to its own output, will produce the set $a^n b^n$, $n > 0$, a context-free but nonregular language.[4] But unbounded applications of rules of this kind appear to be unnecessary in phonology. Similarly, one could write a rewrite rule that produces an exact copy of an arbitrarily long sequence. Such a rule would produce sets of strings of the form *ww*, and such a language is not regular. Rules of this second type may be appropriate for describing *reduplication* phenomena, though reduplication is typically bounded; thus, strictly speaking, can be handled by regular operations. But Culy reports on a case of seemingly unbounded reduplication in Bambara [19] (although this is presumably a morphological, rather than phonological rule, though, in turn, this is beside the point). So putting to one side the case of reduplication, and given that we never need to apply a rule an unbounded number of times to its own output, nearly all phonological operations can be modeled using regular languages or relations [20,21].

A. Rule-Based Approaches

Let us start with the case of phonological rewrite rules, and their implementation using finite-state transducers, as this is by far the most common approach used for phonological modeling in working computational morphology systems. Consider again the Finnish vowel harmony example described earlier. A simple finite-state transducer that implements the *a*/*ä* alternation is shown in Fig. 1.[5] In this transducer, state 0 is the initial state, and both states 0 and 1 are final states. The machine stays in state 0, transducing *a* to itself, until it hits one of the front vowels in the input *ä*, *ö*, *y*, in which case it goes to state 1, where it will transduce input *a* to *ä*.

Arguably the most significant work in computational phonology is the work of Kaplan and Kay at Xerox PARC [21], which presents a concrete method for the

[4] Here ϵ (conventionally) denotes the empty string.

[5] Readers unfamiliar with finite-state automata can consult standard texts such as Ref. 22. There is unfortunately much less accessible material on finite-state *transducers*: much of the material that has been written presumes a fair amount of mathematical background on the part of the reader. There is some background on transducers and their application to linguistic problems in Refs. 1, 21. One might also consult the slides from Ref. 23.

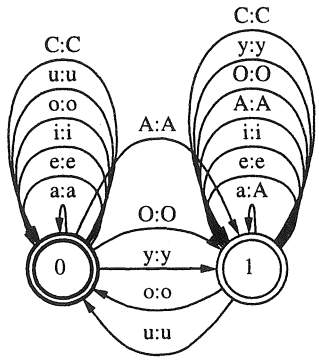

Fig. 1 A partial FST implementation of the Finnish vowel harmony rule. Here, *A* represents *ä*, *O* represents *ö* and *C* represents any consonant. The labels on the arcs represent the symbol-to-symbol transductions, with the input on the left and the output on the right of the colon.

compilation of rewrite rules into FSTs. Although only published in 1994, the work was started over a decade previously; in its early form, it was the inspiration for work on *two-level morphology* by Koskenniemi [24], as well as later work such as Refs. 25–27. The compilation algorithm presented by Kaplan and Kay is too complex to present in all its detail here; the reader is urged to consult their paper.[6] However the gist of the idea can be presented as follows. Imagine a phonologist applying the Finnish vowel harmony rule to a concrete example, say *lyhyt* + *tä*, and imagine that the phonologist was being more than usually careful in bookkeeping. The application of the rule can be notionally broken down into two phases: the identification of those parts of the input string that are going to be changed, and the actual changing of the parts. One can accomplish the first part by marking with brackets occurrences of potential contexts for the rule application.[7] The marked version of input *lyhyt* + *ta* would be *lyhyt* + *t* < *a* > . Note that the brackets have to be correctly restricted in that < should occur after a potential left context, and > before a potential right context—in this case null. The second part involves replacing a correctly bracketed input string with the appropriate output. Thus *lyhyt* + *t* < *a* > becomes *lyhyt* + *t* < *ä* > . The brackets may now be removed. The crucial insight of the Kaplan and Kay, and related algorithms is that all of these operations—bracket insertion, bracket restrictiotn, replacement, and bracket deletion—can be handled by finite-state transducers. Furthermore, FSTs can be composed together to produce a single FST that has the same input–output behavior as the sequential application of the individual machines. That is for any two transducers T_1 and T_2, one can produce the composition, $T_3 = T_1 \circ T_2$, which has the property that for any input I, $(I \circ T_1) \circ T_2$ is the same

[6] The algorithm in Ref. 27 is simpler, but is still relatively complex in all its details.

[7] This is not as simple as it sounds, as one application of the rule could *feed* a second application. This in fact happens in Finnish: in the example *pöydä* + *llä* + *kö* (table + ADESSIVE + QUESTION) 'on the table?', the first (boldface) *ä* generated by vowel harmony feeds the application of the rule to the second *ö*: compare the form of the *-ko* suffix in the word *kahvia* + *ko* (coffee + QUESTION). Thus the environments for the application will not in general be present in the initial string. This is handled in the rule compilation algorithms by a more involved placement of brackets than what is described here.

as $I \circ (T_1 \circ T_2) = I \circ T_3$. So, one can produce a single FST corresponding to the rule by composing together the FSTs that perform each of the individual operations.[8]

The main differences among rule-based computational models of phonology relate to the manner in which rules interact. Kaplan and Kay's model takes advantage of the observation that one can construct by composition a single transducer that has the same input–output behavior as a set of transducers connected 'in series.' Thus, one can model systems of ordered rewrite rules either as a series of transducers, or as a single transducer. Work in the tradition of Two-Level Morphology [e.g., 24, 26, 28, 29] applies sets of rules not in series but 'in parallel,' by intersection of the transducers modeling the rules.[9] One of the underlying assumptions of strict interpretations of two-level morphology is that all phonological rules can be stated as relations on lexical and surface representations, crucially obviating the need for intermediate representations. This may seem like a weakening of the power of rewrite rules, but two-level rules are more powerful than standard rewrite rules in one respect: they can refer to both surface and lexical representations, whereas standard rules can refer to only the input (lexical-side) level.[10]

From the point of view of the rule developer, the main difference is that in a two-level system one has to be slightly more careful about interrule interactions. Suppose that one has a pair of rules which in standard rewrite formalism would be written as follows:

$$w \rightarrow \epsilon / V\_\_V$$
$$a \rightarrow i / \_\_i$$

That is, an intervocalic *w* deletes, and an *a* becomes *i* before a following *i*. For an input *lawin* this would predict the output *liin*. In a two-level description the statement of the deletion rule would be formally equivalent, but the vowel coalescence rule must be stated such that it explicitly allows the possibility of an intervening deleted element (lexical *w*, surface ϵ). Despite these drawbacks two-level approaches have on the whole been more popular than ones based on ordered rules, and many systems and variations on the original model of Koskenniemi have been proposed; these include 'mixed' models combining both two-level morphology and serial composition [31], and multitape (as opposed to two-tape) models such as Ref. 32. Part of the reason for this popularity is undoubtedly due to an accident of history: when Kaplan and Kay first proposed modeling phonological rule systems in terms of serial composition of transducers, physical machine limitations made the construction of large rule sets by this approach slow and inefficient. It was partly to circumvent this ephemeral impracticality that Koskenniemi originally proposed the two-level

[8] This description does not accurately characterize all compilation methods. The two-level rule compilation method of Refs. 26 and 28, for example, depends on *partitions* of the input string into context and target regions. This partitioning serves the same function as brackets, although there are algorithmic differences.

[9] As Kaplan and Kay note, regular relations are closed under composition, they are not in general closed under intersection. Thus one cannot necessarily intersect two transducers. However, same-length relations and relations with only a bounded number of deletions or insertions are closed under intersection. As long as two-level rules obey these constraints, then it is possible to model systems of such rules using intersection.

[10] The formal relation between systems of two-level rules and systems of standard rewrite rules is not trivial. For some discussion see Ref. 1, pages 146–147, and Ref. 30.

approach which, because it was workable and because it was the first general phonological and morphological processing model, gained in popularity.

B. Declarative Approaches

The alternative to rule-based approaches is to state phonological regularities in terms of constraints. For the toy Finnish vowel harmony example such a model could be developed along the following lines. Imagine that the lexicon simply generates all variants for all elements as exemplified in Fig. 2. Then we could simply state a constraint disallowing *a* after front harmony-inducing vowels (FH), and *ä* after back harmony-inducing vowels (BH), with possibly intervening consonants and neutral vowels (NV):

$$\neg(FH(C \vee NV)^*\text{a}) \wedge \neg(BH(C \vee NV)^*\text{ä})$$

This constraint, implemented as finite-state acceptor and intersected with the acceptor in Fig. 2 will yield the single path *syytä.*

A computational model of phonology along these lines is Bird and Ellison's *one-level* phonology [33], in which phonological constraints are modeled computationally as finite-state acceptors, although of a state-labeled, rather than arc-labeled, variety of the kind we have been tacitly assuming here. Declarative approaches to phonology have been gaining ground in part because the more traditional rule-based approaches are (to some extent correctly) perceived as being overly powerful, and less explanatory. Still, although the linguistic reasons for choosing one approach over the other may be genuine, it is important to bear in mind that computational implementations of the two approaches—finite-state transducers versus finite-state acceptors—are only minimally different.

C. Summary

This rather lengthy digression on the topic of computational phonology was necessitated by the fact that word formation typically interacts in the phonology in rather complex ways. As a result, computational morphology systems must handle phonological alternations in some fashion; finite-state models are the most popular.

Although the topic of this section has been phonology, because the largest application of computational morphology is to written language, one is usually interested not so much in phonology as in orthographic spelling changes, such as the *y/i* alternation in English *try/tries.*[11] In some languages, such as Finnish, the orthography is sufficiently close to a phonemic transcription that this distinction is relatively unimportant. In other languages, English being a prime example, there is a

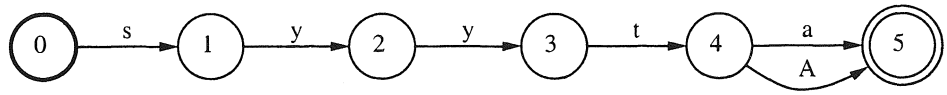

Fig. 2 An unpruned lattice for the Finnish partitive form *syy* + *tä* 'reason.' Again, *A* represents *ä.*

[11] There are applications, such as TTS, for which one needs to deal with both phonology and orthographic spelling changes.

large disparity between phonological and orthographic representations and alternations. But from the point of view of the techniques discussed here the distinction is not very material, as finite-state methods can be applied equally well to both.

4. FINITE-STATE TECHNIQUES IN MORPHOLOGY

Suppose one is building a syntactic parser for written Spanish. The lexical analysis component of such a system should be able to accept as input a written word form, and produce as output an annotation of that form in terms of grammatical features that are likely to be relevant for subsequent syntactic analysis. Such features would include information on the grammatical category of the word, but also other features, such as verbal person, number, tense, and mood features. For the purposes of this discussion, let us take a simple subset consisting of three verbs, *hablar* 'speak,' *cerrar* 'close,' and *cocer* 'cook,' conjugated in the present and preterite indicative forms:

Features	*hablar*	*cerrar*	*cocer*
ind pres 1 sg	hablo	**cie**rro	**cue**zo
ind pres 2 sg	hablas	**cie**rras	**cue**ces
ind pres 3 sg	habla	**cie**rra	**cue**ce
ind pres 1 pl	hablamos	cerramos	cocemos
ind pres 2 pl	habláis	cerráis	cocéis
ind pres 3 pl	hablan	**cie**rran	**cue**cen
ind pret 1 sg	hablé	cerré	cocí
ind pret 2 sg	hablaste	cerraste	cociste
ind pret 3 sg	habló	cerró	coció
ind pret 1 pl	hablamos	cerramos	cocimos
ind pret 2 pl	hablasteis	cerrasteis	cocisteis
ind pret 3 pl	hablaron	cerraron	cocieron

Represented here are two different conjugations, the *-ar* conjugation (*hablar* and *cerrar*) and the *-er* conjugation (*cocer*). Also represented are some vowel and consonant changes—the *c*/*z* alternation in *cocer*, and the diphthongization of the stem vowel in certain positions in *cerrar* and *cocer*. These stem changes are common, although by no means entirely predictable. In particular, whereas many verbs undergo the vowel alternations *o*/*ue*, *e*/*ie* there are also many verbs that do not (*comer* 'eat' does not for example). The correct generalization, for verbs that undergo the rule, is that the diphthong is found when the vowel is stressed. For the sake of simplicity in the present discussion we will assume that the alternation refers to a lexical feature **diph**, and is triggered by the presence of a single following unstressed syllable, a mostly reliable indicator that the current syllable is stressed.

Perhaps the simplest finite-state model that would generate all and only the correct forms for these verbs is an *arclist* model, borrowing a term from Ref. 14. Consider the arclist represented in the following table. On any line, the first column

represents the start state of the transition, the second column the end state, the third the input, and the fourth the output; a line with a single column represents the final state of the arclist. Thus one can go from the START state to the state labeled *ar* (representing *-ar* verbs) by reading the string *habl* and outputting the string *hablar* **vb**. The final state of the whole arclist is labeled WORD. More generally, verb stems are represented as beginning in the initial (START) state, and going to a state that records their paradigm affiliation, *-ar* or *-er*. From these paradigm states, one can attain the final WORD state by means of appropriate endings for that paradigm. The verb stems for *cerrar* and *cocer* have lexical features **diph** and **c/z**, which trigger the application of spelling changes.

START	ar	habl	hablar **vb**
START	ar	cerr **diph**	cerrar **vb**
START	er	coc **diph c/z**	cocer **vb**
ar	WORD	+o#	+ **ind pres 1 sg**
ar	WORD	+as#	+ **ind pres 2 sg**
ar	WORD	+a#	+ **ind pres 3 sg**
ar	WORD	+amos#	+ **ind pres 1 pl**
ar	WORD	+'ais#	+ **ind pres 2 pl**
ar	WORD	+an#	+ **ind pres 3 pl**
ar	WORD	+'e#	+ **ind pret 1 sg**
ar	WORD	+aste#	+ **ind pret 2 sg**
ar	WORD	+'o#	+ **ind pret 3 sg**
ar	WORD	+amos#	+ **ind pret 1 pl**
ar	WORD	+asteis#	+ **ind pret 2 pl**
ar	WORD	+aron#	+ **ind pret 3 pl**
er	WORD	+o#	+ **ind pres 1 sg**
er	WORD	+es#	+ **ind pres 2 sg**
er	WORD	+e#	+ **ind pres 3 sg**
er	WORD	+emos#	+ **ind pres 1 pl**
er	WORD	+'eis#	+ **ind pres 2 pl**
er	WORD	+en#	+ **ind pres 3 pl**
er	WORD	+'i#	+ **ind pret 1 sg**
er	WORD	+iste#	+ **ind pret 2 sg**
er	WORD	+i'o#	+ **ind pret 3 sg**
er	WORD	+imos#	+ **ind pret 1 pl**
er	WORD	+isteis#	+ **ind pret 2 pl**
er	WORD	+ieron#	+ **ind pret 3 pl**
WORD			

This arclist can easily be represented as a finite-state transducer, which we will call D (Figures 3 and 4.)[12] The spelling changes necessary for this fragment involve rules to diphthongize vowels, change c into z, and delete grammatical features and the morpheme boundary symbol:

e → ie / ___ C* **feat*** **diph feat*** + V* C* #
o → ue / ___ C* **feat*** **diph feat*** + V* C* #
c → z / __ **feat*** **c/z** + [+back]
(**feature** ∨ **boundary**) → ϵ

Call the transducer representing these rules R. Then a transducer mapping between the surface forms of the verbs and their decomposition into morphs will be given by $(D \circ R)^{-1}$ (i.e., the *inverse* of $D \circ R$). An analysis of an input verb, represented by a finite-state acceptor S, is simply $I \circ (D \circ R)^{-1}$. Thus *cuezo* $\circ(D \circ R)^{-1}$ is *cocer* **vb** + **ind pres 1 sg**

Strictly finite-state models of morphology are in general only minor variants of the model just presented. For example, the original system of Koskenniemi [24] modeled the lexicon as a set of *tries*, which can be thought of as a special kind of finite automaton. Morphological complexity was handled by *continuation patterns*, which were annotations at the end of morpheme entries indicating which set of tries to continue the search in. But, again, this mechanism merely simulates an ϵ-labeled arc in a finite-state machine. Spelling changes were handled by two-level rules implemented as parallel (virtually, if not actually, intersected) finite-state transducers, as we have seen. Search on the input string involved matching the input string with the input side of the two-level transducers, while simultaneously matching the lexical tries with the lexical side of the transducers, a process formally equivalent to the composition $I \circ (D \circ R)^{-1}$. Variants of two-level morphology, such as the system presented in Ref. 31, are even closer to the model presented here. Tzoukermann and Liberman's work [14] takes a somewhat different approach. Rather than separately constructing D and R and then composing them together, the transducer $(D \circ R)^{-1}$ is constructed directly from a 'precompiled' arclist. For each dictionary entry, morphophonological stem alternants are produced beforehand, along with special tags that limit those alternants to appropriate sets of endings.

Other methods of constructing finite-state lexica present themselves beside the arclist construction just described. For example, one could represent the lexicon of stems (e.g., *habl-*) as a transducer from lexical classes (e.g., *ar-verb*) to the particular stems that belong to those classes. Call this transducer S. The inflectional paradigms can be represented as an automaton that simply lists the endings that are legal for each class: *ar-verb o* **ind pres 1 sg** would be one path of such an automaton, which we may call M. A lexicon automaton can then be constructed by taking the righthand projection of M composed with S concatenated with the universal machine $\Sigma^* : \pi_2[M \circ (S \cdot \Sigma^*)]$.[13] Yet another approach would be to represent the word gram-

[12] Actually, the transducer in Fig. 3 is the inverse of the transducer represented by the arclist.

[13] Following standard notational conventions, Σ denotes the alphabet of a machine and Σ^* the set of all strings (including the null string) over that alphabet. Projection is notated by π.

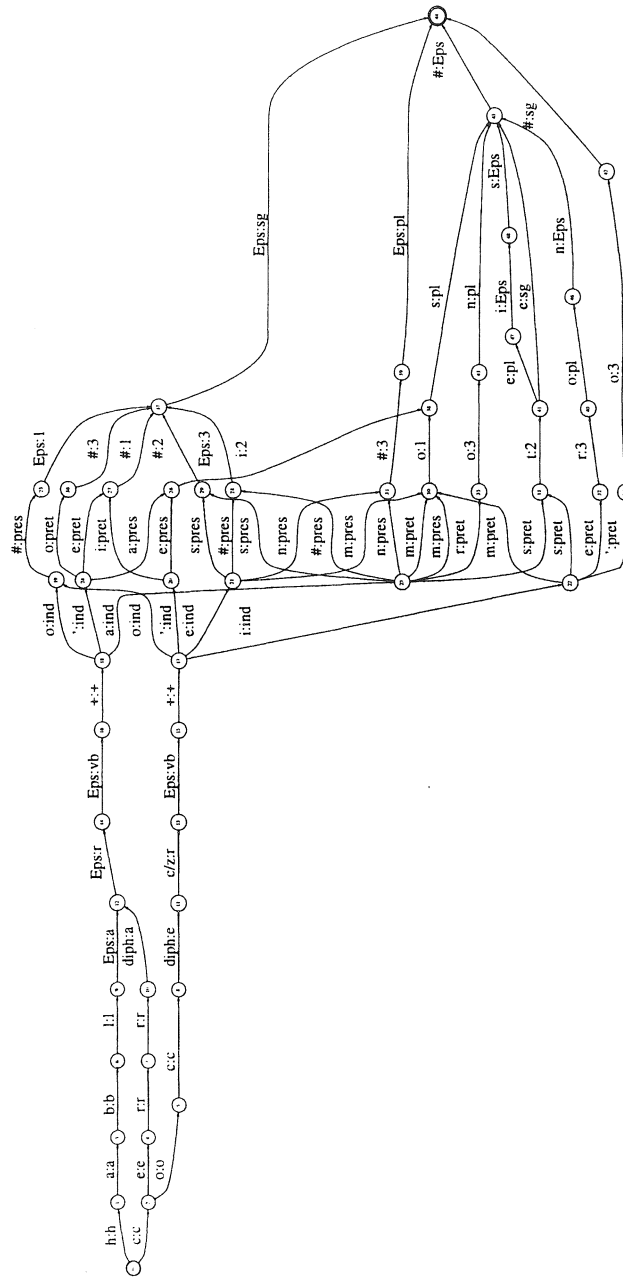

Fig. 3 A transducer for a small fragment of Spanish verbal morphology.

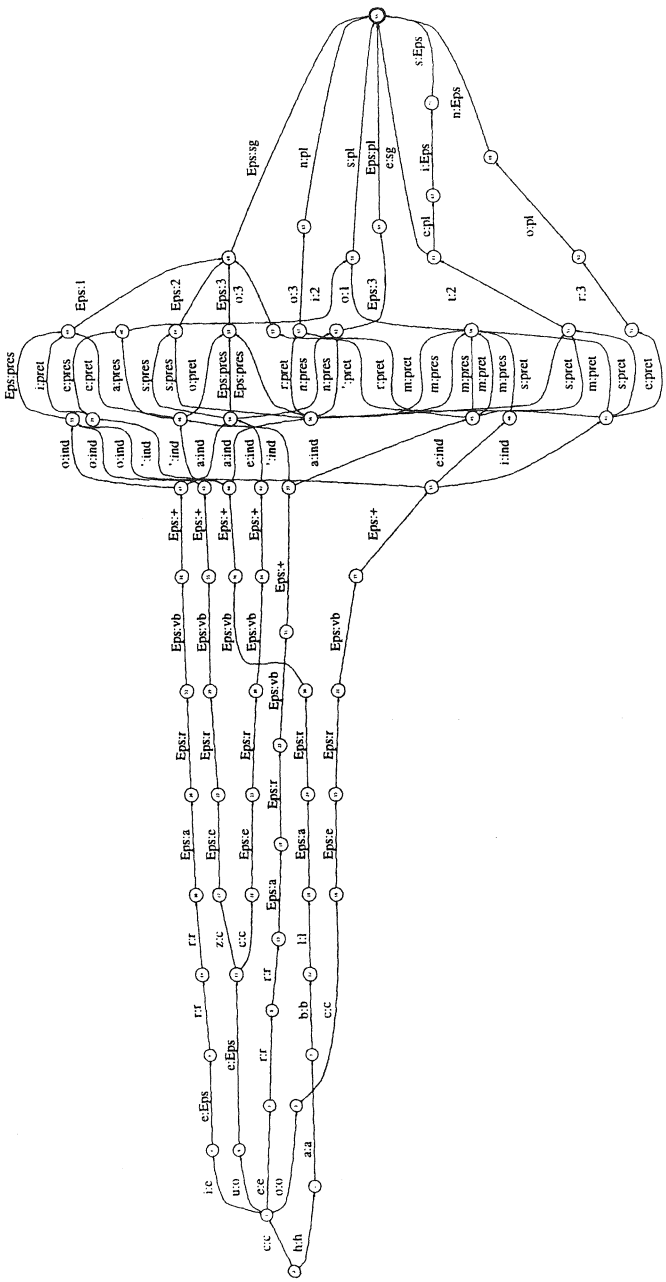

Fig. 4 A transducer that maps between surface and lexical forms for a small fragment of Spanish verbal morphology.

mar as a context-free grammar, and use a finite-state approximation algorithm [34] to produce a finite-state representation of the grammar.

The approach just sketched is sufficient for handling many morphological systems, but there are quite a few phenomena that are not so elegantly handled. One example from English involves cases for which the ability to attach a particular morpheme depends on the prior attachment of a morpheme that is not adjacent to it. The suffix *-able* productively attaches to verbs in English, not to other categories; the prefix *en-/em-* (unproductively) attaches to nouns or adjectives and makes verbs. As one might expect, *-able* can attach to verbs formed with *en-/em-*: *enrichable, empowerable, endangerable.* Crucially, *-able* either does not attach to the base nouns and adjectives, or else it has a very different meaning: **richable, powerable* (different meaning), **dangerable.* So it is only because of the prefix that *-able* is able to attach in these cases. The only way for a finite-state machine to model this is to duplicate a portion of the lexicon: there must be (at least) two entries for *rich, power,* and *danger,* one of which 'remembers' that *en-/em-* has been attached, the other of which remembers that it has not been attached. There are two possible solutions to this problem. One is simply to implement the morphology in a purely finite-state way anyway, but to hide the duplication of structure from the user by a clever set of grammar tools that allow the grammar developer to state the linguistic generalizations without worrying about how they are actually implemented as finite automata. The second solution is to construct higher-order computational models of morphotactics (the 'putting together' of morphemes to form a word), such as the unification-based methods described in Sec. 5.

The general class of models just described is particularly well suited to *concatenative* morphology, involving the stringing together of distinct affixes and roots. It is less well suited to handle so-called nonconcatenative morphology—phenomena such as reduplication, infixation, and the templatic morphology found in Semitic languages [see also Ref. 1, pp 159–170]. Some of these phenomena, such as infixation and reduplication, have never received a serious computational treatment, although it is certainly possible to handle them within strictly finite-state frameworks.

Others, such as Semitic template morphology, have proved more tractable. Various approaches to Semitic have been proposed [35–37]. An interesting recent approach is that of Kiraz [32], based in part on earlier work of Kay. Kiraz presents a multitape (rather than two-tape) transducer model, implementing regular n-relations, for $n > 2$. In this model one tape (typically) represents the input and the others represent different lexical tiers. To take a concrete example, consider the Arabic stem form *kattab*, the class II perfect active stem from the root *ktb*, which can be roughly translated as 'write.' According to standard autosegmental analyses [4], there are three clearly identifiable morphemes here, each on its own tier and each with their own predictable phonological, morphosyntactic, and (more or less) predictable semantic contributions to the form. These are the root itself; the vocalism, here *a*; and the template, which gives the general phonological shape to the word, and that for present purposes we will notate as *CVCCVC*. In Kiraz's model these three tiers are represented by three lexical tapes, which form part of a regular 4-relation with the surface tape. A typical representation would be as follows:

	a					*vocalism tape*
k		t			b	*root tape*
C	V	C	C	V	C	*pattern tape*
k	a	t	t	a	b	*surface tape*

The tapes are related by a set of *two-level* rules over *four* tapes, the particular formalism being an extension of that of Ref. 26 and previous work.

Finally, one issue that frequently comes up in discussions of finite-state morphology is efficiency. In principle, finite-state morphological transducers can become very large, and in the absence of efficient methods for representing them, and efficient algorithms for operations, such as composition, they can easily become unwieldy. Fortunately, efficient algorithms for these operations do exist [see e.g., 23, 38, 39].

5. UNIFICATION-BASED APPROACHES TO WORD STRUCTURE

Misgivings with the admittedly limited mechanisms for handling morpheme combinations within strictly finite-state morphology have led various researchers to augment the set of mechanisms available for expressing morphotactic restrictions. The most popular of these approaches involves *unification* [40]: the typical strategy is to use two-level morphology to model phonological alternations, with unification being used solely to handle morpheme combination possibilities.

An early example of this [41] uses a PATR-based unification scheme on acyclic finite automata, that effectively implements a chart parser. Consider the following example [41, p. 276], which models the affixation of *-ing* in English:

verb → *verb* + *ing*
1 2 3

1. < 2 *cat* >= *verb*
2. < 3 *lex* >=*-ing*
3. < 2 *form* >= *inf*
4. < 1 *cat* >= *verb*
5. < 1 *word* >=< 2*word* >
6. < 1 *form* >= [*tense* : *pres—part*]

Say that the analyzer has already built an arc for an infinitive verb form, and is now analyzing a sequence *-ing*. The foregoing rule will license the construction of a new arc spanning the whole sequence, with the category feature *verb*, the tense features *pres—part*, and the lexical features inherited from the base verb. The equations license this construction because the *cat* and *form* of item 2 match the lexical features of the arc over the verb, and the *lex* of item 3 matches *-ing*. Rule 5 states that the derived word 1 is a form of the base word 2.

Restrictions necessary to handle long-distance dependencies of the kind involving *en-* and *-able* can easily be stated in such a framework. All one needs is a rule that states that *en-* attaches to nouns or adjectives and that the resulting word is a verb; another rule would allow *-able* to attach to verbs, forming adjectives. An example of this kind of discontinuous morphology in Arabic has been discussed

[37], which presents a two-level type model for Arabic morphology. Consider the paradigm for the active imperfect of the class I verbal form of *ktb* (stem *ktub*):

	Singular	Dual	Plural
1	**?a** + *ktub* + **u**	**na** + *ktub* + **u**	**na** + *ktub* + **u**
2/M	**ta** + *ktub* + **u**	**ta** + *ktub* + **aani**	**ta** + *ktub* + **uuna**
2/F	**ta** + *ktub* + **iina**	**ta** + *ktub* + **aani**	**ta** + *ktub* + **na**
3/M	**ya** + *ktub* + **u**	**ya** + *ktub* + **aani**	**ya** + *ktub* + **uuna**
3/F	**6a** + *ktub* + **u**	**ta** + *ktub* + **aani**	**ya** + *ktub* + **na**

Consider the prefix *ya-*, and the suffix *-u*, both of which occur in several positions in the paradigm. An examination of the slots in which they occur (and simplifying somewhat from Beesley's account), suggests that *ya-* and *-u* encode the following sets of features, respectively:

$$
\begin{aligned}
\textit{ya-} &= (\textit{PERSON} = 3 \wedge \\
&\quad ((\textit{GENDER} = M \wedge (\textit{NUMBER} = SG \vee \textit{NUMBER} = DU)) \vee \\
&\quad (\textit{GENDER} = \textit{unspecified} \wedge \textit{NUMBER} = PL)) \\
\textit{-u} &= ((\textit{PERSON} = 1 \wedge \textit{GENDER} = \textit{unspecified} \wedge \textit{NUMBER} = \textit{unspecified}) \\
&\quad \vee (\textit{PERSON} = 2 \wedge \textit{GENDER} = M \wedge \textit{NUMBER} = SG) \\
&\quad \vee (\textit{PERSON} = 3 \wedge \textit{GENDER} = \textit{unspecified} \wedge \textit{NUMBER} = SG))
\end{aligned}
$$

Even if the affixes are ambiguous by themselves, in a particular combination, such as *ya* + *ktub* + *u*, they are unambiguous. That is, the features for the combination *ya—u* are unambiguously $(\textit{PERSON} = 3 \wedge \textit{GENDER} = M \wedge \textit{NUMBER} = SG)$. This set of features can be arrived at by unifying the feature structures for the two affixes. Other work [32,42] includes both finite-state techniques for phonology and unification-based approaches for word structure.

A richer unification-based model [see, 43–45] that, in addition to unification, makes use of general *feature passing conventions* in the style of GPSG [46]. For example, they make use of a 'word-head convention' stated as follows [44, p. 296]:

> The WHead feature-values in the mother should be the same as the WHead feature-values of the right daughter.

In other words, for a predetermined set of WHead (word head) features, a complex word inherits the values for those features from the righthand daughter. For example it is assumed that the set of WHead features includes a plurality feature PLU. So, a word such as *dog* + *s* would inherit the feature specification (*PLU* +) from the affix *-s*, which is lexically specified with that feature specification.[14]

6. PROBABILISTIC METHODS

The methods that we have discussed so far have been categorical in that, although they may assign a set of analyses to a word, rather than just a single analysis, they

[14] One point that I have said nothing about is how to model interactions between morphosyntactic features and phonological rules. Various models for doing this have been proposed [32, 47].

provide no way of ranking those analyses. Now, the use of ad hoc weights to different types of analysis has a long tradition. For example in the DECOMP module of the MITalk TTS system [48], greater cost was assigned to compounds than to words derived by affixation, so that for *scarcity* one prefers an affixational analysis (*scare* + *ity*) over a compounding analysis *scar* + *city*. The use of real corpus-derived weights in models of computational morphology is rarer, however.

One exception is work of Heemskerk [49], who develops a probabilistic context-free model of morphology, where the probabilities of a particular morphological construction are derived from a corpus (in this case the CELEX database for Dutch). For a Dutch word, such as *beneveling* 'intoxication' (the literal gloss would be 'clouding up'), there are a large number of potential analyses that conform to regular Dutch spelling changes: *be* + *nevel* + *ing* (correct), *be* + *neef* + *eling*, *been* + *e* + *veel* + *ing*. . . . Some of these can certainly be ruled out on purely morphotactic grounds, but not all. Of those that remain, Heemskerk's system ranks them according to the estimated probabilities of the context-free productions that would be necessary for their formation. Another probabilistic system is the finite-state Chinese word-segmentation system [50] and (see Sec. 7).

7. MORPHOLOGY AND TOKENISATION

The discussion in this chapter has been based on the premise that the input to lexical analysis has already been tokenised into words. But as we have seen in Chap. 2, there are languages such as Chinese, Japanese, and Thai, in which word boundaries are customarily not written and, therefore, tokenisation cannot be so cleanly separated from lexical analysis. Consider the case of Chinese, and as a concrete example, the approach based on weighted finite-state transducers [50]. Note that there have been many other publications on Chinese segmentation: a partial bibliography on this topic can be found in the paper cited here.

For the Mandarin sentence 日文章魚怎麼說 (*rì wén zhāng yú zěn mo shuō*) "How do you say octopus in Japanese?", we probably would want to say that it consists of four words, namely 日文 *ri-wén* 'Japanese', 章魚 *zhāng-yú* 'octopus', 怎麼 *zěn mo* 'how', and 說 *shuō* 'say'. The problem with this sentence is that 日 *rì* is also a word (e.g., a common abbreviation for Japan) as are 文章 *wén zhāng* 'essay,' and 魚 *yú* 'fish,' so there is not a unique segmentation. The task of segmenting Chinese text can be handled by the same kind of finite-state transduction techniques as have been used for morphology, with the addition of weights to model relative likelihoods of different analyses. So, one can represent a Chinese dictionary as a WFST D, with the input being strings of written characters and the output being lexical analyses. For instance, the word 章魚 'octopus' would be represented as the sequence of transductions 章:章/0.0 魚:魚/0.0 ϵ:noun/13.18, where 13.18 is the estimated cost (negative log-probability) of the word. Segmentation is then accomplished by finding the lowest weight string in $S \circ D^*$.[15]

[15] See Refs. 38, 50, and 51 for further information. For the purposes of this discussion we are assuming an interpretation of WFSTs where weights along a path are *summed* and for cases where there is more than one cost assigned to an analysis, the *minimal* cost is selected: this is the (*min*, +) interpretation. Other interpretations (formally: *semirings*) are possible, such as (+, ·).

As with English, no Chinese dictionary covers all of the words that one will encounter in Chinese text. For example, many words that are derived by productive morphological processes are not generally found in the dictionary. One such case in Chinese involves words derived via the nominal plural affix 們-*men*. While some words in 們 will be found in the dictionary (e.g., 他們 *tā-men* 'they'; 人們 *rén-men* 'people'), many attested instances will not: for example, 小將們 *xiǎo-jiāng-men* 'little' (military) generals', 青蛙們 *qīng-wā-men* 'frogs.' To handle such cases, the dictionary is extended using standard finite state morphology techniques. For instance, we can represent the fact that 們 attaches to nouns by allowing ϵ-transitions from the final states of noun entries, to the initial state of a subtransducer containing 們. However, for the present purposes it is not sufficient merely to represent the morphological decomposition of (say) plural nouns, for one also wants to estimate the cost of the resulting words. For derived words that occur in the training corpus we can estimate these costs as we would the costs for an underived dictionary entry. For nonoccurring, but possible plural forms, we use the Good–Turing estimate [e.g., 52], whereby the aggregate probability of previously unseen members of a construction is estimated as N_1/N, where N is the total number of observed tokens and N_1 is the number of types observed only once; again, we arrange the automaton so that noun entries may transition to 們, and the cost of the whole (previously unseen) construction sums to the value derived from the Good–Turing esimate. Similar techniques are used to compute analyses and cost estimates for other classes of lexical construction, including personal names, and foreign names in transliteration.

Even in languages such as English or French, for which white space is used to delimit words, tokenisation cannot be done entirely without recourse to lexical information, because one find many multiword phrases—*in spite of, over and above, give up*—which semantically at least are clearly a single lexical entry. See Ref. 53 for relevant discussion.

8. FINAL THOUGHTS

Even with tokenization, we do not exhaust the set of lexical analysis problems presented by unrestricted natural language text. For applications such as TTS one also needs to find canonical forms for such things as abbreviations and numerals. Techniques from computational morphology have been applied in the case of numeral expansion. So, van Leeuwen [54] and Traber [55] model the expansion of expressions such as *123* as *one hundred twenty three* using rewrite rules, represented computationally as FSTs. In the Bell Laboratories multilingual TTS system [56], the problem is factored into two stages: the expansion of digit strings into sums of products of powers of ten ($1{\cdot}10^2 + 2{\cdot}10^1 + 3{\cdot}10^0$); and the mapping from that representation into number names (e.g., $2{\cdot}10^1 \rightarrow$ *twenty*). Both of these processes can be modeled using FSTs.

To summarize: the basic challenge for a lexical analysis system is to take (usually written) words as input and find lexical properties that are relevant for the particular application. The discussion in this chapter has been neutral on what these lexical properties are, and has focused rather on methods for obtaining lexical analyses from written representations of words. This is by design: much of the discussion in the literature on 'the lexicon' tacitly assumes that a particular kind of lexical information (usually semantic information) should be the output of a

lexical analysis system. There is no basis for this bias. As outlined in the introduction, depending on the application one might be more interested in the semantic or pragmatic properties, syntactic properties, phonological properties or any combination of these.

ACKNOWLEDGMENTS

I thank George Kiraz and Evelyne Tzoukermann for comments on this chapter.

REFERENCES

1. R Sproat. Morphology and Computation. Cambridge, MA: MIT Press, 1992.
2. C Hockett. Two models of grammatical description. Word 10:210–231, 1954.
3. R Beard. Lexeme–Morpheme Base Morphology. Albany: SUNY, 1995.
4. J McCarthy. Formal problems in semitic morphology and phonology. PhD dissertation, The Massachusetts Institute of Technology, Cambridge, MA, 1979. Distributed by Indiana University Linguistics Club (1982).
5. A Marantz. On the Nature of Grammatical Relations. Cambridge, MA: MIT Press, 1984.
6. R Sproat. On deriving the lexicon. PhD dissertation, The Massachusetts Institute of Technology, Cambridge, MA, 1985. Distributed by MIT Working Papers in Linguistics.
7. M Baker. Incorporation: a Theory of Grammatical Function Changing. Chicago: University of Chicago Press, 1988.
8. R Lieber. Deconstructing Morphology: Word Formation in a Government-Binding Syntax. Chicago: University of Chicago Press, 1991.
9. P Matthews. Morphology. Cambridge, UK: Cambridge University Press, 1974.
10. M Aronoff. Word Formation in Generative Grammar. Cambridge, MA: MIT Press, 1976.
11. S Anderson. A-Morphous Morphology. Cambridge: Cambridge University Press, 1992.
12 WU Wurzel. Inflectional Morphology and Naturalness. Studies in Natural Language and Linguistic Theory. Dordrecht: Kluwer, 1989.
13. R Byrd, J Klavans, M Aronoff, F Anshen. Computer methods for morphological analysis. ACL Proceedings, 24th Annual Meeting, Morristown, NJ; Association for Computational Linguistics, 1986, pp. 120–127.
14. E. Tzoukermann and M Liberman. A finite-state morphological processor for Spanish. COLING-90, Vol 3. COLING, 1990, pp, 3:277–286.
15. J Goldsmith. Autosegmental and Lexical Phonology: An Introduction. Basil: Blackwell, 1989.
16. J Coleman. Phonological representations—Their names, forms and powers. PhD dissertation, University of York, 1992.
17. S Bird, E Klein. Phonological analysis in typed feature structures. Comput Ling 20:455–491, 1994.
18. A Prince, P Smolensky. Optimality theory. Tech Rep 2, Rutgers University, Piscataway, NJ, 1993.
19. C Culy. The complexity of the vocabulary of Bambara. Linguist Philos. 8:345–351, 1985.
20. CD Johnson. Formal Aspects of Phonological Description. The Hague: Mouton, 1972.
21. R Kaplan, M Kay. Regular models of phonological rule systems. Comput Linguist 20:331–378, 1994.
22. J Hopcroft, J Ullman. Introduction to Automata Theory, Languages and Computation. Reading, MA: Addison–Wesley, 1979.

23. M Mohri, M Riley, R Sproat. Algorithms for speech recognition and language processing. Tutorial presented at COLING-96, 1996. Available as xxx.lanl.gov/ps/cmp-lg/9608018.
24. K Koskenniemi. Two-level morphology: a general computational model for word-form recognition and production. PhD dissertation, University of Helsinki, Helsinki, 1983.
25. L Karttunen, K Beesley. Two-level rule compiler. Tech Rep P92-00149, Xerox Palo Alto Research Center, 1992.
26. E Grimley-Evans, G. Kiraz, S Pulman. Compiling a partition-based two-level formalism. Proceedings of COLING-96, Copenhagen, Denmark, COLING, 1996, pp 454–459.
27. M Mohri, R Sproat. An efficient compiler for weighted rewrite rules. In 34th Annual Meeting of the Association for Computational Linguistics. Morristown, NJ: Association for Computational Linguistics, 1996, pp 231–238.
28. GA Kiraz. SemHe: a generalized two-level system. In: 34th Annual Meeting of the Association for Computational Linguistics. Morristown, NJ: Association for Computational Linguistics, 1996, pp 159–166.
29. L Karttunen KIMMO: a general morphological processor. In L Karttunen, ed., Texas Linguistic Forum, 22. Austin, TX: University of Texas, 1983, pp 165–186.
30. L Karttunen. Directed replacement. 34th Annual Meeting of the Association for Computational Linguistics, Morristown, NJ: Association for Computational Linguistics, 1996, pp 108–115.
31. L Karttunen, R Kaplan, A Zaenen. Two-level morphology with composition. COLING-92. COLING, 1992, pp 141–148.
32. GA Kiraz. Computational nonlinear morphology with emphasis on semitic language. PhD dissertation, University of Cambridge, 1996. Forthcoming, Cambridge University Press.
33. S Bird, T. Ellison. One-level phonology: autosegmental representations and rules as finite automata. Comput Linguist 20:55–90, 1994.
34. FCN Pereira, RN Wright. Finite-state approximation of phrase–structure grammars. 29th Annual Meeting of the Association for Computational Linguistics. Morristown, NJ: Association for Computational Linguistics, 1991. pp, 246–255.
35. M Kay. Nonconcatenative finite-state morphology. ACL Proceedings, 3rd European Meeting. Morristown, NJ. Association for Computational Linguistics, 1987, pp. 2–10.
36. L Kataja, K Koskenniemi. Finite-state description of Semitic morphology: a case study of ancient Akkadian. COLING-88. Morristown, NJ: Association for Computational Linguistics, 1988, pp 313–315.
37. K Beesley. Computer analysis of Arabic morphology: a two-level approach with detours. Proceedings of the Third Annual Symposium on Arabic Linguistics, University of Utah, 1989.
38. F Pereira, M Riley. Speech recognition by composition of weighted finite automata. CMP-LG archive paper 9603001, 1996. www.lanl.gov/ps/cmp-lg/9603001.
39. M Mohri. Finite-state transducers in language and speech processing. Comput Linguist 23: 269–311, 1997.
40. S Shieber. An Introduction to Unification-Based Approaches to Grammar. Chicago: University of Chicago Press, 1986. Center for the Study of Language and Information.
41. J Bear. A morphological recognizer with syntactic and phonological rules. COLING-86. Morristown, NJ: Association for Computational Linguistics, 1986, pp 272–276.
42. H Trost. The application of two-level morphology to non-concatenative German morphology. COLING-90. Vol 2. Morristown, NJ: Association for Computational Linguistics, 1990, pp 371–376.
43. G Russel, S Pulman, G Ritchie, A Black, A dictionary and morphological analyser for English. COLING-86, Morristown, NJ: Association for Computational Linguistics, 1986, pp 277–279.

44. G Ritchie, S Pulman, A Black, G Russel. A computational framework for lexical description. Comput Linguist 13:290–307, 1987.
45. G Ritchie, G Russell, A Black, S Pulman. Computational Morphology: Practical Mechanisms for the English Lexicon. Cambridge, MA: MIT Press, 1992.
46. G Gazdar, E Klein, G Pullum, I Sag. Generalized Phrase Structure Grammar. Cambridge, MA: Harvard University Press, 1985.
47. J Bear. Morphology with two-level rules and negative rule features. COLING-88. Morristown, NJ: Association for Computational Linguistics, 1988, pp 28–31.
48. J Allen, MS Hunnicutt, D Klatt. From Text to Speech: the MITalk System. Cambridge: Cambridge University Press, 1987.
49. J Heemskerk. A probabilistic context-free grammar for disambiguation in morphological parsing. European ACL, Sixth European Conference. 1993, pp 183–192.
50. R Sproat, C Shih, W Gale, N Chang. A stochastic finite-state word-segmentation algorithm for Chinese. Comput Linguist 22: 1996.
51. F Pereira, M Riley, R Sproat. Weighted rational transductions and their application to human language processing. In: ARPA Workshop on Human Language Technology, Advanced Research Projects Agency, March 8–11, 1994, pp 249–254.
52. KW Church, W Gale. A comparison of the enhanced Good–Turing and deleted estimation methods for estimating probabilities of English bigrams. Comput Speech Lang 5: 19–54, 1991.
53. L Karttunen, J Chanod, G Grefenstette, A Schiller. Regular expressions for language engineering. J Nat Lang Eng 2:305–328, 1996.
54. H van Leeuwen. Too$_L$iP: a development tool for linguistic rules. PhD dissertation, Technical University Eindhoven, 1989.
55. C Traber. SVOX: the implementation of a text-to-speech system for German. Tech Rep 7, Swiss Federal Institute of Technology, Zurich, 1995.
56. R Sproat. Multilingual text analysis for text-to-speech synthesis. J Nat Lang Eng 2:369–380, 1996.

4

Parsing Techniques

CHRISTER SAMUELSSON

Xerox Research Centre Europe, Grenoble, France

MATS WIRÉN

Telia Research, Stockholm, Sweden

1. INTRODUCTION

This chapter gives an overview of rule-based natural language parsing and describes a set of techniques commonly used for handling this problem. By *parsing* we mean the process of analysing a sentence to determine its syntactic structure according to a formal grammar. This is usually not a goal in itself, but rather, an intermediary step for the purpose of further processing, such as the assignment of a meaning to the sentence. We shall thus take the output of parsing to be a hierarchical structure suitable for semantic interpretation (the topic of Chap. 5). As for the input to parsing, we shall assume this to be a string of words, thereby simplifying the problem in two respects (treated in Chaps. 2 and 3, respectively): First, our informal notion of a word disregards the problem of chunking, that is, appropriate segmentation of the input into parsable units. Second, the "words" will usually have been analysed in separate phases of lexical lookup and morphological analysis, which we hence exclude from parsing proper.

As pointed out by Steedman [101], there are two major ways in which processors for natural languages differ from processors for artificial languages, such as compilers. The first is in the power of the grammar formalisms used (the *generative capacity*): Whereas programming languages are mostly specified using carefully restricted subclasses of context-free grammar, Chomsky [20] argued that natural languages require more powerful devices (indeed, this was one of the driving ideas of Transformational Grammar.)[1] One of the main motivations for the need for

[1] For a background on formal grammars and formal-language theory, see Hopcroft and Ullman [38].

expressive power has been the presence of unbounded dependencies, as in English wh-questions:

Who did you give the book to _? {1}

Who do you think that you gave the book to _? {2}

Who do you think that he suspects that you gave the book to _? {3}

In Examples {1}–{3} it is held that the noun phrase "who" is displaced from its canonical position (indicated by "_") as indirect object of "give." Because arbitrary amounts of material may be embedded between the two ends, as suggested by {2} and {3}, these dependencies might hold at unbounded distance. Although it was later shown that phenomena such as this can be described using restricted formalisms [e.g., 2, 26, 30, 41], many natural language systems make use of formalisms that go far beyond context-free power.

The second difference concerns the extreme structural *ambiguity* of natural language. At any point in a pass through a sentence, there will typically be several grammar rules or parsing actions that might apply. Ambiguities that are strictly local will be resolved as the parser proceeds through the input, as illustrated by {4} and {5}:

Who has seen John? {4}

Who has John seen? {5}

If we assume left-to-right processing, the ambiguity between "'who" as a subject or direct object will disappear when the parser encounters the third word of either sentence ("seen" and "John", respectively). However, it is common that local ambiguities propagate in such a way that global ambiguities arise. A classic example is the following:

Put the block in the box on the table {6}

Assuming that "put" subcategorizes for two objects, there are two possible analyses of {6}:

Put the block [*in the box on the table*] {7}

Put [*the block in the box*] *on the table* {8}

If we add another prepositional phrase ("... in the kitchen"), we obtain five analyses; if we add yet another, we obtain fourteen, and so on. Other examples of the same phenomenon are nominal compounding and conjunctions. As discussed in detail by Church and Patil [21], "every-way ambiguous" constructions of this kind have a number of analyses that is superexponential in the number of added components. More specifically, the degree of ambiguity follows a combinatoric series called the Catalan Numbers, in which the nth number is given by

$$C_n = \binom{2n}{n} \frac{1}{n+1}.$$

In other words, even the process of just returning all the possible analyses would lead to a combinatorial explosion. Thus, much of the work on parsing—hence, much of

the following exposition—deals somehow or other with ways in which the potentially enormous search spaces resulting from local and global ambiguity can be efficiently handled.

Finally, an additional aspect in which the problem of processing of natural languages differs from that of artificial languages is that any (manually written) grammar for a natural language will be incomplete—there will always be sentences that are completely reasonable, but that are outside of the coverage of the grammar. This problem is relevant to robust parsing, which will be discussed only briefly in this chapter (see Sec. 8.A).

2. GRAMMAR FORMALISMS

Although the topic of grammar formalisms falls outside of this chapter, we shall introduce some basic devices solely for the purpose of illustrating parsing; namely, context-free grammar and a simple form of constraint-based grammar. The latter, also called unification grammar, currently constitutes a widely adopted class of formalisms in computational linguistics (Fig. 1). In particular, the declarativeness of the constraint-based formalisms provides advantages from the perspective of grammar engineering and reusability, compared with earlier frameworks such as augmented transition networks (ATNs) (see Sec. 5).

A key characteristic of constraint-based formalisms is the use of feature terms (sets of attribute–value pairs) for the description of linguistic units, rather than atomic categories as in phrase-structure grammars.[2] Feature terms can be nested:

DCG Definite Clause Grammar (Pereira and Warren [73])

FUG Functional Unification Grammar (Kay [51, 52])

PATR (strictly speaking PATR-II; Shieber et al. [95], Shieber [91, 93])

CLE Core Language Engine (Alshawi [5])

TDL Type Definition Language (Krieger and Schäfer [58])

ALE Attribute Logic Engine (Carpenter and Penn [18])

TAG Tree-Adjoining Grammar (Joshi et al. [42, 41], Schabes et al. [86], Vijay-Shanker and Joshi [109])

LFG Lexical-Functional Grammar (Bresnan [14])

GPSG Generalized Phrase Structure Grammar (Gazdar et al. [30])

CUG Categorial Unification Grammar (Haddock et al. [34], Uszkoreit [108], Karttunen [47])

HPSG Head-driven Phrase Structure Grammar (Pollard and Sag [75])

Fig. 1 Examples of constraint-based linguistic formalisms (or tools).

[2] A phrase–structure grammar consists entirely of context-free or content-sensitive rewrite rules (with atomic categories).

their values can be either atomic symbols or feature terms. Furthermore, they are partial (underspecified) in the sense that new information may be added as long as it is compatible with old information. The operation for merging and checking compatibility of feature constraints is usually formalized as *unification*. Some formalisms, such as PATR, are restricted to simple unification (of conjunctive terms), whereas others allow disjunctive terms, sets, type hierarchies, or other extensions; for example, LFG and HPSG. In summary, feature terms have proved to be a versatile and powerful device for linguistic description. One example of this is unbounded dependency, as illustrated by Examples {1}–{3}, which can be handled entirely within the feature system by the technique of gap threading [46].

Several constraint-based formalisms are phrase-structure-based in the sense that each rule is factored in a phrase–structure backbone and a set of constraints that specify conditions on the feature terms associated with the rule (e.g., PATR, CLE, TDL, LFG, and TAG, although the latter uses tree substitution and tree adjunction rather than string rewritings). Analogously, when parsers for constraint-based formalisms are built, the starting-point is often a phrase–structure parser that is augmented to handle feature terms. This is also the approach we shall follow here.

We shall thus make use of a context-free grammar $G = \langle N, T, P, S \rangle$, where N and T are finite sets of nonterminal categories and terminal symbols (words), respectively, P is a finite set of production (rewrite) rules, and $S \in N$ is the start category. Each production of P is of the form $X \rightarrow \alpha$ such that $X \in N$ and $\alpha \in (N \cup T)^*$. If α is a word, the production corresponds to a lexical entry, and we call X a lexical or preterminal category.

Any realistic natural language grammar that licenses complete analyses will be *ambiguous*; that is, some sentence will give rise to more than one analysis. When a phrase-structure grammar is used, the analysis produced by the parser typically takes the form of one or several parse trees. Each nonleaf node of a parse tree then corresponds to a rule in the grammar: the label of the node corresponds to the left-hand side of the rule, and the labels of its children correspond to the right-hand-side categories of the rule.

Furthermore, we shall make use of a constraint-based formalism with a context-free backbone and restricted to simple unification, thus corresponding to PATR. A grammar rule in this formalism can be seen as an ordered pair of a production $X_0 \rightarrow X_1 \cdots X_n$ and a set of equational constraints over the feature terms of types $X_0, \cdots, X_n$. A simple example of a rule, encoding agreement between the determiner and the noun in a noun phrase, is the following:

$$
\begin{aligned}
&X_0 \rightarrow X_1 X_2 \\
&\quad \langle X_0 \; \textit{category} \rangle = \textit{NP} \\
&\quad \langle X_1 \; \textit{category} \rangle = \textit{Det} \\
&\quad \langle X_2 \; \textit{category} \rangle = \textit{N} \\
&\quad \langle X_1 \; \textit{agreement} \rangle = \langle X_2 \; \textit{agreement} \rangle
\end{aligned}
$$

A rule may be represented as a feature term implicitly recording both the context-free production and the feature constraints. The following is a feature term corresponding to the previous rule (where $\boxed{1}$ indicates identity between the associated elements):

$$\begin{bmatrix} X_0: & [category: NP\ ] \\ X_1: & \begin{bmatrix} category: Det \\ agreement: \boxed{1}[\ ] \end{bmatrix} \\ X_2: & \begin{bmatrix} category: N \\ agreement: \boxed{1} \end{bmatrix} \end{bmatrix}$$

When using constraint-based grammar, we will assume that the parser manipulates and outputs such feature terms.

3. BASIC CONCEPTS IN PARSING

A *recognizer* is a procedure that determines whether or not an input sentence is grammatical according to the grammar (including the lexicon). A *parser* is a recognizer that produces associated structural analyses according to the grammar (in our case, parse trees or feature terms).[3] A *robust parser* attempts to produce useful output, such as a partial analysis, even if the complete input is not covered by the grammar. More generally, it is also able to output a single analysis even if the input is highly ambiguous (see Sec. 8.A).

We can think of a grammar as inducing a search space consisting of a set of states representing stages of successive grammar-rule rewritings and a set of transitions between these states. When analysing a sentence, the parser (recognizer) must rewrite the grammar rules in some sequence. A sequence that connects the state S, the string consisting of just the start category of the grammar, and a state consisting of exactly the string of input words, is called a *derivation*. Each state in the sequence then consists of a string over $(N \cup T)^*$ and is called a *sentential form*. If such a sequence exists, the sentence is said to be grammatical according to the grammar. A derivation in which only the leftmost nonterminal of a sentential form is replaced at each step is called *leftmost*. Conversely, a *rightmost* derivation replaces the rightmost nonterminal at each step.

Parsers can be classified along several dimensions according to the ways in which they carry out derivations. One such dimension concerns rule invocation: In a *top-down* derivation, each sentential form is produced from its predecessor by replacing one nonterminal symbol X by a string of terminal or nonterminal symbols $X_1 \cdots X_n$, where $X \rightarrow X_1 \cdots X_n$ is a production of P. Conversely, in a *bottom-up* derivation, each sentential form is produced by replacing $X_1 \cdots X_n$ with X given the same production, thus successively applying rules in the reverse direction.

Another dimension concerns the way in which the parser deals with ambiguity, in particular, whether the process is *deterministic* or *nondeterministic*. In the former case, only a single, irrevocable choice may be made when the parser is faced with local ambiguity. This choice is typically based on some form of lookahead or systematic preference (compare Sec. 4.B).

[3] To avoid unnecessary detail, we will not say much about the actual assembling of analyses during parsing. Hence, what we will describe will resemble recognition more than parsing. However, turning a recognizer into a parser is usually not a difficult task, and is described in several of the references given.

A third dimension concerns the way in which the search space is traversed in the case of nondeterministic parsing: A *depth-first* derivation generates one successor of the initial state, then generates one of its successors, and then continues to extend this path in the same way until it terminates. Next, it uses backtracking to generate another successor of the last state generated on the current path, and so on. In contrast, a *breadth-first* derivation generates all the successors of the initial state, then generates all the successors at the second level, and so on. Mixed strategies are also possible.

A fourth dimension concerns whether parsing proceeds from left to right (strictly speaking front to back) through the input or in some other order, for example, inside-out from the heads.[4]

4. BASIC TOP-DOWN AND BOTTOM-UP PARSING

Parsers can be classified as working top-down or bottom-up. This section describes two such basic techniques, which can be said to underlie most of the methods to be presented in the rest of the chapter.

A. Basic Top-Down Parsing: Recursive Descent

A simple method for pure top-down parsing consists in taking a rule the left-hand side of which is the start symbol, rewrite this to its right-hand side symbols, then try to expand the leftmost nonterminal symbol out of these, and so on until the input words are generated. This method is called *recursive descent* [3] or *produce-shift* [97].

For simplicity and generality, we shall use a pseudodeductive format for specifying recursive-descent parsing (loosely modelled on the deductive system of Shieber et al. [94]; see Fig. 2). We thus assume that the parser manipulates a set of items, each of which makes an assertion about the parsing process up to a particular point. An item has the form $\langle \cdot \beta, j \rangle$, where β is a (possibly empty) string of grammar categories or words. Such an item asserts that the sentence up to and including word w_j followed by β is a sentential form according to the grammar. The dot thus marks the division between the part of the sentence that has been covered in the parsing process and the part that has not.[5] As the symbols of the string following the dot are recursively expanded, the item is pushed onto a stack.

To handle nondeterminism, the basic scheme in Fig. 2 needs to be augmented with a control strategy. Following Sec. 3, the two basic alternatives are depth-first search with backtracking or breadth-first search, maintaining several derivations in parallel.

Recursive descent in some form is the most natural approach for parsing with ATNs (see Sec. 5). The top-down, depth-first processing used with Definite Clause Grammars [73], as executed by Prolog, is also an instance of recursive descent. A problem with pure top-down parsing is that left-recursive rules (for example,

[4] A *head* is a designated, syntactically central category of the right-hand side of a grammar rule. For example, in the noun–phrase rule $NP \rightarrow Det\ A\ N$, the noun N is usually considered to be the syntactic head.

[5] Actually, the dot is superfluous and is used here only to indicate the relation with tabular approaches; compare Sec. 6.

Input: A string of words $w_1 \cdots w_n$.

Output: A set of items.

Method: If S is the start category, then generate an initial item $\langle \cdot S, 0 \rangle$. Do the following steps until the item $\langle \cdot, n \rangle$ has been generated or no more items can be generated.

Scan: If there is an item $\langle \cdot w_{j+1}\beta, j \rangle$, then generate an item $\langle \cdot \beta, j+1 \rangle$. (In effect, consume the word w_{j+1}.)

Predict: If there is an item $\langle \cdot Y\beta, j \rangle$ and a rule of the form $Y \rightarrow \gamma$, then generate an item $\langle \cdot \gamma\beta, j \rangle$. (In effect, substitute the element Y with the right-hand side γ of the rule.)

Fig. 2 A basic scheme for recursive–descent parsing.

NP → *NP PP*) lead to infinite recursion. This can be avoided by adopting some additional bookkeeping (compare Sec. 6.2B) or by eliminating left-recursion from the grammar [3]. More generally, however, the strong role of the lexical component in many constraint-based grammar frameworks makes top-down parsing in its pure form a less useful strategy.

B. Basic Bottom-Up Parsing: Shift–Reduce

The main data structure used in shift–reduce parsing is a parser stack, which holds constituents built up during analysis or words that are read from the input. Basically, the parser pushes (*shifts*) words onto the stack until the right-hand side of a production appears as one or several topmost elements of the stack. These elements may then be replaced by (*reduced* to) the left-hand-side category of the production. If successful, the process continues by successive shift and reduce steps until the stack has been reduced to the start category and the input is exhausted. [3].

Again, we shall use a pseudodeductive format for specifying shift–reduce parsing (loosely following Shieber et al. [94]). An item has the form $\langle \alpha \cdot, j \rangle$, where α is a (possibly empty) string of grammar categories or words. This kind of item asserts that α derives the sentence up to and including word w_j. The dot thus marks the division between the part of the sentence that has been covered in the parsing process and the part that has not.[6] Put differently, the string α corresponds to the parser stack, the right end of which is the top of the stack, and j denotes the word last shifted onto the stack.

Figure 3 provides a basic scheme for shift–reduce parsing. However, as it stands, it is impractical with any sizable grammar, since it has no means of (deterministically) choosing steps that will lead to success on a well-formed input. For example, if α appears on the top of the stack and $Y \rightarrow \alpha$ and $Y \rightarrow \alpha\beta$ are productions, it is not clear if the parser should shift or reduce. A *shift–reduce* (*parsing*) *table*, or *oracle*, can then be used for guiding the parser. The purpose of the parsing table is thus to endow the pure bottom-up processing with a certain amount of predictive ability. There are two situations that it must resolve: *shift–reduce conflicts*, in which

[6] Again, the dot is superfluous and is used only to indicate the relation with tabular approaches and LR parsing; compare Secs. 6 and 7.

Input: A string of words $w_1 \cdots w_n$.

Output: A set of items.

Method: If S is the start category, then generate an initial item $\langle \bullet, 0 \rangle$. Do the following steps until the item $\langle S \bullet, n \rangle$ has been generated or no more items can be generated.

> *Shift*: If there is an item of the form $\langle \alpha \bullet, j \rangle$, then generate an item $\langle \alpha w_{j+1} \bullet, j + 1 \rangle$. (In effect, consume the word w_{j+1} and push it onto the stack.)
>
> *Reduce*: If there is an item of the form $\langle \alpha \gamma \bullet, j \rangle$ and a rule of the form $Y \rightarrow \gamma$, then generate an item $\langle \alpha Y \bullet, j \rangle$. (In effect, pop the stack element(s) γ and push the element Y onto the stack.)

Fig. 3 A basic scheme for shift–reduce parsing.

the parser can either shift a word onto the stack or reduce a set of elements on the stack to a new element (as seen in the foregoing), and *reduce–reduce conflicts*, in which it is possible to rewrite one or several elements on the stack by more than one grammar rule.[7]

Again, the nondeterminism can be handled by using breadth-first search or depth-first search with backtracking (in which case the parsing table might order the possible choices such that the more likely paths are processed first). Alternatively, one might try to keep to deterministic processing by ensuring that, as far as possible, a correct decision is made at each choice point. This approach was pioneered by Marcus [62] with his parser PARSIFAL. Marcus' aim was to model a particular aspect of hypothesized human linguistic performance; namely, that (almost all) globally unambiguous sentences appear to be parsed deterministically. With globally ambiguous sentences of the kind illustrated by example {6} PARSIFAL would deliver at most one of the legal analyses, possibly with a flag indicating ambiguity. (Alternatively, a semantic component might provide guidance to the parser.) More importantly, however, Marcus was interested in having the parser reject certain locally ambiguous sentences known as *garden-path sentences* [10]. An example is the following:

The boat floated down the river sank {9}

It has been claimed that the temporary ambiguity in {9} between "floated" as introducing a reduced relative clause and as a main verb forces the human sentence processor to back up when encountering the word "sank," thereby rendering the process nondeterministic (and increasing the effort). To allow processing of sentences such as those in Examples {4} and {5}, but not {9}, PARSIFAL depends on being able to delay its decisions a certain number of steps. It achieves this in two ways: First, PARSIFAL's shift–reduce machinery itself provides a way of delaying decisions until appropriate phrases can be closed by reductions. But PARSIFAL also makes use of a limited lookahead; namely, a three-cell buffer in which the next three words or NPs can be examined.[8]

---

[7] Sometimes preterminal categories, rather than actual words, are shifted to the stack; in this event, there may also be *shift–shift conflicts*.

[8] Under certain conditions, the lookahead may involve up to five buffer cells.

Although PARSIFAL can be seen as a shift–reduce parser, it has a richer set of operations than the basic scheme specified in Fig. 3, and makes use of a procedural rule format rather than phrase–structure rules. In subsequent work, Shieber [90] and Pereira [71] have shown how a similar deterministic behaviour can be obtained based on standard shift–reduce parsing. By adopting suitable parsing tables, they are thus able to account for garden-path phenomena as well as various kinds of parsing preferences discussed in the literature. Furthermore, in contrast with PARSIFAL, the grammars used by their parsers can be ambiguous.

Marcus' approach has been further developed by Hindle [36,37], who has developed an extremely efficient wide-coverage deterministic *partial* parser called FIDDITCH (compare Sec. 8). Furthermore, a semantically oriented account, in which garden-path effects are induced by the semantic context, rather than the sentence structure as such, is provided by Crain and Steedman [23]. For a discussion of Marcus' approach, see also Briscoe [15]. LR parsing, a more elaborate shift–reduce technique, in which the parsing table consists of a particular finite automaton, will be described in Sec. 7.

5. PARSING WITH TRANSITION NETWORKS

During the 1970s and early 1980s, Augmented Transition Networks (ATNs) constituted the dominating framework for natural language processing. To understand ATNs, it is useful to begin by looking at the simplest kind of transition network, which consists of a finite set of states (vertices) and a set of state transitions that occur on input symbols. This is called a *finite-state transition network* (FSTN; [31,111]). In automata theory, the equivalent abstract machine is called a *finite (state) automaton* [38]. Transition networks have a distinguished initial state and a set of distinguished final states. Each encountered state corresponds to a particular stage in the search of a constituent and corresponds to one or several dotted expressions (as used in Secs. 4, 6, and 7). Each arc of an FSTN is labelled with a terminal symbol; traversing the arc corresponds to having found some input matching the label. FSTNs can describe only the set of regular languages. Nevertheless, finite-state frameworks are sometimes adopted in natural language parsing on grounds of efficiency or modelling of human linguistic performance [22,27,57,77,80].

By introducing recursion, in other words, by allowing the label of an arc to be a nonterminal symbol referring to a subnetwork, a *recursive transition network* (RTN) is obtained [31,111]. In automata theory, the equivalent notion is a *pushdown automaton* [38]. An RTN thus consists of a set of subnetworks. Furthermore, it is equipped with a pushdown stack that enables the system to jump to a specified subnetwork and to return appropriately after having traversed it. An RTN can describe the set of context-free languages.

An ATN is an RTN in which each arc has been augmented with a *condition* and a sequence of *actions*. The conditions and actions may be arbitrary. The condition associated with an arc must be satisfied for the arc to be chosen, and the actions are executed as the arc is traversed. The purpose of the actions is to construct pieces of linguistic structure (typically, parts of a parse tree). This structure is stored in *registers*, which are passed along as the parsing proceeds. In addition, a global set of registers called the *hold list*, is commonly used for handling unbounded dependen-

cies. Because of these very general mechanisms, the ATN framework is Turing-equivalent, allowing any recursively enumerable language to be described.

Mainly because of the way registers are typically used, the most straightforward processing regimen for ATN parsing is top-down, left-to-right (compare Sec. 4.A). Furthermore, depth-first search with backtracking is commonly used, but breadth-first is also possible. In practice, the efficiency of depth-first search can be increased by ordering the set of outgoing arcs according to the relative frequencies of the corresponding inputs.

The principal references to ATNs are Woods [113, 114] and Woods et al. [115]. For an excellent introduction, see Bates [8]. For a slightly different conception of ATNs (especially in terms of the structures generated), see Winograd [111, Chap. 5]. A comparison of ATNs with Definite Clause Grammars is made by Pereira and Warren [73], who show that the latter are able to capture everything essential from the former, with the added advantage of providing a declarative formalism.

6. TABULAR PARSING (CHART PARSING)

Because of the high degree of local ambiguity of natural language, the same subproblem often appears more than once during (nondeterministic) parsing of a sentence. One example of this is the analysis of the prepositional phrase "on the table" in Example {6}, which can be attached in two ways in the parse tree for the sentence. However, the prepositional-phrase analysis itself is independent of this attachment decision. It, therefore, can be useful to define equivalence classes of partial analyses, store these analyses in a table, and look them up whenever needed, rather than recomputing them each time as the nondeterministic algorithms in Secs. 4 and 5 would do. This technique, which is called *tabulation* (hence, *tabular parsing*), potentially allows exponential reductions of the search space, and thereby, provides a way of coping with ambiguity in parsing. An additional advantage of the tabular representation is that it is compatible with a high degree of flexibility relative to search and control strategies. The tabular framework in which this property has been most systematically explored is called *chart parsing*.[9]

A. Preliminaries

The table in which partial analyses are stored is often conceived of as a directed graph $C = \langle V, E \rangle$, the *chart*. V is then a finite, nonempty set of vertices, $E \subseteq V \times V \times R$ is a finite set of edges and R is a finite set of dotted context-free rules (see following). We assume that the graph is labelled such that each vertex and edge has a unique label, which we denote by a subscript (e.g., v_i or e_j).

The vertices $v_0, \cdots, v_n \in V$ correspond to the linear positions between each word of an n-word sentence $w_1 \cdots w_n$. An edge $e \in E$ between vertices v_i and v_j carries information about a (partially) analysed constituent between the correspond-

[9] To some extent, the use of the terms "tabular parsing" and "chart parsing" reflects different traditions: tabular parsing was originally developed within the field of compiler design [4,25,48,116], and chart parsing, more or less independently, within computational linguistics [44,49,50,53,103,104]. However, chart parsing often refers to a framework which, as a means to attain flexible control, makes use of *both* a table (in the form of a chart) and an agenda (see Sec. 6.D). More fundamentally, tabulation is a general technique that can be realized by other means than a chart (see Sec. 7.B).

ing positions. However, to avoid unnecessary detail, we have not included any element corresponding to the actual analysis in the edge tuple (but compare Sec. 6.E.) We write an edge as a triple

$$\langle v_s, v_t, X \rightarrow \alpha \cdot \beta \rangle$$

with starting vertex v_s, ending vertex v_t, and dotted context-free rule $X \rightarrow \alpha \cdot \beta$. The dotted rule asserts that X derives α followed by β, where α corresponds to the part of the constituent parsed so far. If β is empty, then the edge represents a completely analysed constituent and is called *inactive* (*passive*); otherwise it represents a partially analysed constituent and is called *active*. An active edge thus represents a state in the search for a constituent, including a hypothesis about the category of the remaining, needed input. If α is empty, then the edge is active and looping. It then represents the initial state, at which no input has been parsed in the search for the constituent.[10]

B. Chart Parsing with Context-Free Grammar

Figure 4 provides a scheme for chart parsing with a context-free grammar. It operates in a bottom-up fashion but makes use of top-down prediction, and thus corresponds to Earley's [4,25] algorithm.[11] Furthermore, it is equivalent to the top-down strategies of Kay [53, p. 60] and Thompson [103, p. 168].

The algorithm involves three repeated steps, which can be carried out in any order:

1. *Scan*: Add an inactive edge for each lexical category of the word.
2. *Predict (top-down)*: Add an edge according to each rule whose first left-hand-side category matches the needed category of the triggering active edge, unless an equivalent edge already exists in the chart.
3. *Complete*: Add an edge whenever the category of the first needed constituent of an active edge matches the category of an inactive edge.

In addition, a top-down initialization of the chart is needed. Note that, because a phrase–structure grammar (with atomic categories) is being used, the redundancy test in the prediction step corresponds to a simple equality test. The parsing scheme is specified in Fig. 4. For an illustration of a chart resulting from this scheme, using the example grammar in Fig. 5, see Fig. 6. The contents of the edges are further specified in Fig. 7. Typically, what is desired as output is not the chart as such, but the set of parse trees that cover the entire input and the top node of which is the start category of the grammar. Such a result can be extracted by a separate procedure, which operates on the set of S edges connecting the first and last vertex.

It is possible to obtain a pure bottom-up behaviour by replacing the top-down predictor with a corresponding bottom-up step:

---

[10] Chart parsing using both active and inactive edges is sometimes called *active chart parsing*. This is to distinguish it from the case when only inactive edges are used; that is, when the chart corresponds to a *well-formed substring table*.

[11] Edges correspond to *states* [25] or *items* [4, p. 320] in Earley's algorithm.

Input: A string of words $w_1 \cdots w_n$.

Output: A chart $C = \langle V, E \rangle$.

Method: Add an initial top-down prediction $\langle v_0, v_0, X_0 \rightarrow \cdot\ \alpha \rangle$ for each rule $X_0 \rightarrow \alpha$ such that $X_0 = S$, where S is the start category of the grammar. Repeat the following steps for each vertex v_j such that $j = 0, \ldots, n$ until no more edges can be added to the chart.

Scan: If $w_j = a$, then for each lexical entry of the form $X_0 \rightarrow a$, add an edge $\langle v_{j-1}, v_j, X_0 \rightarrow a\ \cdot \rangle$.

Predict (top-down): For each edge of the form $\langle v_i, v_j, X_0 \rightarrow \alpha \cdot X_m \beta \rangle$ and each rule of the form $Y_0 \rightarrow \gamma$ such that $Y_0 = X_m$, add an edge $\langle v_j, v_j, Y_0 \rightarrow \cdot\ \gamma \rangle$ unless it already exists.

Complete: For each edge of the form $\langle v_i, v_j, X_0 \rightarrow \alpha \cdot X_m \beta \rangle$ and each edge of the form $\langle v_j, v_k, Y_0 \rightarrow \gamma \cdot \rangle$, add an edge $\langle v_i, v_k, X_0 \rightarrow \alpha X_m \cdot \beta \rangle$ if $X_m = Y_0$.

Fig. 4 A scheme for chart parsing with top-down prediction (Earley-style), using context-free grammar.

2. *Predict (bottom-up)*: Add an edge according to each rule whose first right-hand-side category matches the category of the triggering inactive edge, unless an equivalent edge already exists in the chart.

No initialization of the chart is needed for this. The algorithm is specified in Fig. 8, and an example chart is shown in Fig. 9.[12] It is often referred to as a *left-corner* algorithm, because it performs leftmost derivations (and the left corner of a rule is the leftmost category of its right-hand side). It is equivalent to the bottom-up strategies of Kay [53, p. 58] and Thompson [103, p. 168]. Furthermore, it is related to the Cocke–Kasami–Younger (CKY) algorithm [4, 48, 116], although, in particular, this matches categories from right to left in the right-hand side of a grammar rule and only makes use of inactive edges.

Because of the avoidance of redundant calculation, the problem of chart parsing with a context-free grammar is solvable in polynomial time. More specifically, the worst-case complexity of each of the two foregoing algorithms is $O(|G|^2 \cdot n^3)$, where n is the length of the input and $|G|$ is the size of the grammar expressed in terms of the number of rules and nonterminal categories (see, e.g.,

S	$\rightarrow$	$NP\ VP$	Det	$\rightarrow$	the
NP	$\rightarrow$	$Det\ N$	N	$\rightarrow$	old
NP	$\rightarrow$	$Det\ A\ N$	N	$\rightarrow$	man
VP	$\rightarrow$	V	N	$\rightarrow$	ships
VP	$\rightarrow$	$V\ NP$	A	$\rightarrow$	old
			V	$\rightarrow$	man
			V	$\rightarrow$	ships

Fig. 5 Sample grammar and lexicon.

[12] As formulated here, the algorithm does not work with empty productions.

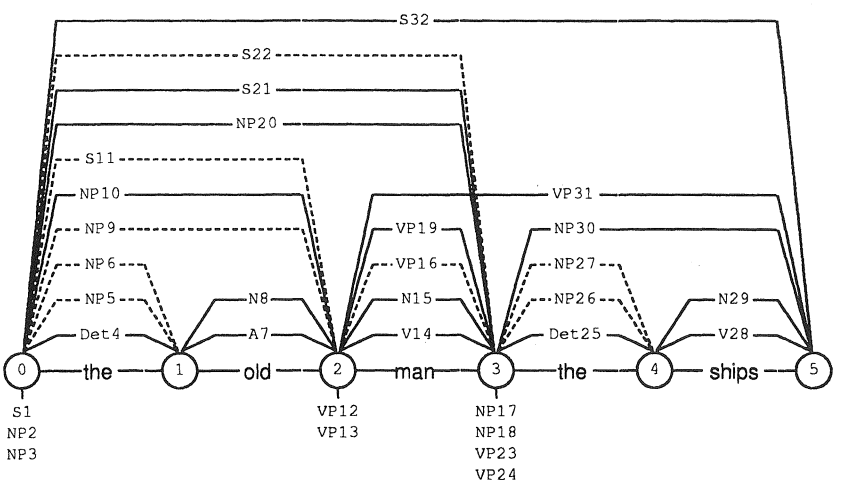

Fig. 6 Chart for the sentence "The old man the ships" using the Earley-style algorithm in Fig. 4 and the grammar given in Fig. 5. Inactive edges are drawn using continuous lines, active edges using dashed lines, and predicted (looping) edges are indicated below the vertices.

Earley [25], Aho and Ullman [4], or Barton et al. [7, Sec. 7.6].) For investigations of practical performance of chart parsing see, e.g., [11, 76, 88, 99, 106, 112].

C. Some Refinements

Various refinements of the foregoing two basic algorithms are possible. One option is filtering, which attempts to reduce the number of edges by making sure that only predictions that in some stronger sense are useful will be added to the chart. For example, pure top-down parsing has the problem that many predictions may be useless, because they are made before the input to which they are meant to apply has been scanned. Moreover, if empty categories are allowed (say, because the

S_1: $\langle v_0, v_0, S \rightarrow \cdot NP\ VP\rangle$
NP_2: $\langle v_0, v_0, NP \rightarrow \cdot Det\ N\rangle$
NP_3: $\langle v_0, v_0, NP \rightarrow \cdot Det\ A\ N\rangle$
Det_4: $\langle v_0, v_1, Det \rightarrow \text{the}\cdot\rangle$
NP_5: $\langle v_0, v_1, NP \rightarrow Det \cdot N\rangle$
NP_6: $\langle v_0, v_1, NP \rightarrow Det \cdot A\ N\rangle$
A_7: $\langle v_1, v_2, A \rightarrow \text{old}\cdot\rangle$
N_8: $\langle v_1, v_2, N \rightarrow \text{old}\cdot\rangle$
NP_9: $\langle v_0, v_2, NP \rightarrow Det\ A \cdot N\rangle$
NP_{10}: $\langle v_0, v_2, NP \rightarrow Det\ N \cdot\rangle$
S_{11}: $\langle v_0, v_2, S \rightarrow NP \cdot VP\rangle$
VP_{12}: $\langle v_2, v_2, VP \rightarrow \cdot V\rangle$
VP_{13}: $\langle v_2, v_2, VP \rightarrow \cdot\ V\ NP\rangle$
V_{14}: $\langle v_2, v_3, V \rightarrow \text{man}\cdot\rangle$
N_{15}: $\langle v_2, v_3, N \rightarrow \text{man}\cdot\rangle$
VP_{16}: $\langle v_2, v_3, VP \rightarrow V \cdot NP\rangle$
NP_{17}: $\langle v_3, v_3, NP \rightarrow \cdot Det\ N\rangle$
NP_{18}: $\langle v_3, v_3, NP \rightarrow \cdot Det\ A\ N\rangle$
VP_{19}: $\langle v_2, v_3, VP \rightarrow V\cdot\rangle$
NP_{20}: $\langle v_0, v_3, NP \rightarrow Det\ A\ N\cdot\rangle$
S_{21}: $\langle v_0, v_3, S \rightarrow NP\ VP\cdot\rangle$
S_{22}: $\langle v_0, v_3, S \rightarrow NP\cdot VP\rangle$
VP_{23}: $\langle v_3, v_3, VP \rightarrow \cdot V\rangle$
VP_{24}: $\langle v_3, v_3, VP \rightarrow \cdot NP\rangle$
Det_{25}: $\langle v_3, v_4, Det \rightarrow \text{the}\cdot\rangle$
NP_{26}: $\langle v_3, v_4, NP \rightarrow Det \cdot N\rangle$
NP_{27}: $\langle v_3, v_4, NP \rightarrow Det \cdot A\ N\rangle$
V_{28}: $\langle v_4, v_5, V \rightarrow \text{ships}\cdot\rangle$
N_{29}: $\langle v_4, v_5, N \rightarrow \text{ships}\cdot\rangle$
NP_{30}: $\langle v_3, v_5, NP \rightarrow Det\ N\cdot\rangle$
VP_{31}: $\langle v_2, v_5, VP \rightarrow V\ NP\cdot\rangle$
S_{32}: $\langle v_0, v_5, S \rightarrow NP\ VP\cdot\rangle$

Fig. 7 Contents of the chart edges obtained in Fig. 6.

Input: A string of words $w_1 \cdots w_n$.

Output: A chart $C = \langle V, E \rangle$.

Method: Repeat the following steps for each vertex v_j such that $j = 0, \ldots, n$ until no more edges can be added to the chart.

Scan: As in Figure 4.

Predict (bottom-up): For each edge of the form $\langle v_j, v_k, X_0 \rightarrow \alpha \bullet \rangle$ and each rule of the form $Y_0 \rightarrow Y_1 \gamma$ such that $Y_1 = X_0$, add an edge $\langle v_j, v_j, Y_0 \rightarrow \bullet Y_1 \beta \rangle$ unless it already exists.

Complete: As in Figure 4.

Fig. 8 A scheme for chart parsing with bottom-up prediction, using context-free grammar.

grammar makes use of gaps), pure bottom-up parsing has the problem that every such category must be predicted at each point in the input string, because the parser cannot know in advance where the gaps occur.

To deal with problems such as these, a reachability relation $\mathcal{R}$ on the set of grammatical categories can be precompiled: $\mathcal{R}$ is defined as the set of ordered pairs $\langle X_0, X_1 \rangle$ such that there is a production $X_0 \rightarrow X_1 \cdots X_n$. If $\mathcal{R}^*$ is the reflexive and transitive closure of $\mathcal{R}$, then $\mathcal{R}^*(X, X')$ holds if and only if $X = X'$ or there exists some derivation from X to $X' \cdots$ such that X' is the leftmost category in a string of categories dominated by X.

Earley-style parsing with *bottom-up filtering* can then be obtained by augmenting the algorithm in Fig. 4 with a condition that checks if some category of a word w_j is reachable from the first needed category in α of an edge $\langle v_{j-1}, v_{j-1}, X \rightarrow \cdot \alpha \rangle$ about to be predicted; if not, this prediction is discarded. This corresponds to Kay's [53, p. 48] directed top-down strategy; it can also be seen as a one-symbol lookahead.

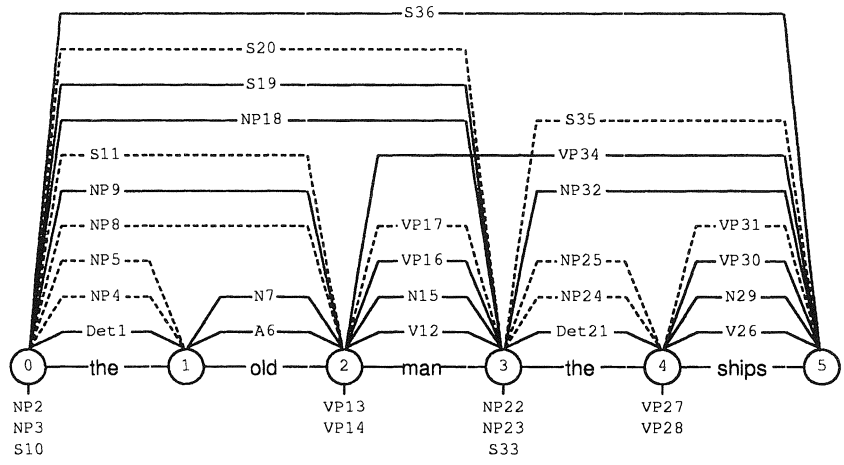

Fig. 9 Chart for the sentence "The old man the ships" using the pure bottom-up algorithm in Fig. 8 and the grammar in Fig. 5.

Analogously, bottom-up parsing with *top-down filtering* can be obtained by augmenting the algorithm in Fig. 8 with a condition that checks if the category of an edge $\langle v_k, v_k, X' \rightarrow \cdot\ \gamma \rangle$ about to be predicted is reachable from the first needed category in β of some incoming active edge $\langle v_i, v_k, X \rightarrow \alpha \cdot \beta \rangle$; if not, the edge about to be predicted is discarded. (This strategy needs an initial top-down prediction in order not to filter out predictions made at the first vertex.) This corresponds to the method described by Pratt [76, p. 424], who refers to the top-down filter as an oracle; it also corresponds to Kay's [53, p. 48] directed bottom-up parsing.

When several ambiguities are localized to the same part of a sentence, it is possible to use a more economic representation than the one above. For example, in the sentence

He put the block in the box on the table {10}

the verb phrase "put the block in the box on the table" has two readings in accordance with example {6}. Two inactive edges will thus be used to represent these readings. However, each of these edges will be treated in exactly the same way by any superior edge using it as a subedge. It is therefore possible to collapse the two edges into a single one in which two parse trees are shared. This technique is called (*local-ambiguity*) *packing* [106]; compare the end of Sec. 7.B.

Furthermore, other forms of rule invocation are possible. For example, in head-driven chart parsing the predictions are triggered from the heads instead of from the left corners, as in Fig. 8. For some work on this, see Bouma and van Noord [13], Kay [54], Satta and Stock [85] and Nederhof and Satta [68]. Another variant is bidirectional chart parsing [84], in which edges may be completed both to the left and to the right (in other words, the chart is here an undirected graph).

Chart (or tabular) parsing appears in several incarnations: For example, it can be seen as an instance of dynamic programming [9], and it has been formalized as deduction [74, 94], in which tabulation corresponds to maintaining a cache of lemmas. For general discussion of tabular parsing, see also Refs. 67, 89, 96.

D. Controlling Search

The algorithms in Figs. 4 and 8 do not specify any order in which to carry out the three parsing steps; in other words, they lack an explicit search strategy. To control search, an *agenda* can be used [53,104]. The idea is as follows: Every newly generated edge is added to the agenda. Edges are moved from the agenda to the chart one by one until the agenda is empty. Whenever an edge is moved to the chart, its consequences in terms of new (predicted or completed) edges are generated and themselves added to the agenda for later consideration.

Different search strategies can be obtained by removing agenda items in different orders: For example, treating the agenda as a (last-in-first-out stack results in a behaviour similar to depth-first search; treating it instead as a queue first-in-first-out stack) results in a behaviour similar to breadth-first search. Furthermore, by introducing a preference metric with respect to agenda elements, more fine-grained control is possible [28].

E. Chart Parsing with Constraint-Based Grammar

Basically, the chart parsers in Figs. 4 and 8 can be adapted to constraint-based grammar by keeping a feature term D in each edge, and by replacing the matching of atomic categories with unification. We thus take an edge to be a tuple

$$\langle v_s, v_t, X_0 \rightarrow \alpha \cdot \beta, D \rangle$$

starting from vertex v_s and ending at vertex v_t with dotted context-free rule $X_0 \rightarrow \alpha \cdot \beta$ and feature term D.

A problem in chart parsing with constraint-based grammar, as opposed to phrase-structure grammar, is that the edge-redundancy test involves comparing complex feature terms instead of testing for equality between atomic symbols. For this reason, we need to make sure that no previously added edge *subsumes* a new edge to be added [72, p. 199; 92]. One edge subsumes another edge if and only if the first three elements of the edges are identical and the fourth element of the first edge subsumes that of the second edge. Roughly, a feature term D subsumes another feature term D' if D contains a subset of the information in D'. The rationale for using this test is that we are interested only in adding edges that are less specific than the old ones, for everything we could do with a more specific edge, we could also do with a more general one. In general, the test must be carried out both in the prediction and completion step. (If an agenda is used, this needs analogous testing for the same reason. For efficiency it is also desirable that any existing edge that is subsumed by a new edge be removed, although this introduces some extra bookeeping.)

Figure 10 provides a scheme for tabular parsing with a constraint-based grammar formalism (for its context-free counterpart, see Fig. 4). It corresponds to Shieber's [92] generalization of Earley's [25] algorithm. Here, for example, $D(\langle X_0$ *category*$\rangle)$ denotes the substructure at the end of the path $\langle X_0$ *category*$\rangle$ from the root of the feature structure D, and $D \sqcup E$ denotes the unification of the feature structures D and E.

As in the context-free case, alternative rule-invocation strategies can be obtained by changing the predictor. To generate more selective predictions, the feature terms of the left-hand-side category of the rule are unified with the needed category of the triggering active edge (rather than to just matching their *category* features, as would be the case if only the context-free backbone was used). Now, unification grammars may contain rules that lead to prediction of ever more specific feature terms that do not subsume each other, thereby resulting in infinite sequences of predictions. In logic programming, the occur check is used for circumventing a corresponding circularity problem. In constraint-based grammar, Shieber [92] introduced the notion of *restriction* for the same purpose. A restrictor removes those portions of a feature term that could potentially lead to nontermination. This is in general done by replacing those portions with free (newly instantiated) variables, which typically removes some coreference. The purpose of restriction is to ensure that terms to be predicted are only instantiated to a certain depth, so that terms will eventually subsume each other.

Input: A string of words $w_1 \cdots w_n$.

Output: A chart $C = \langle V, E \rangle$.

Method: Initialize the chart with an edge $\langle v_0, v_0, X_0 \rightarrow \cdot\, \alpha, D \rangle$ for each rule $\langle X_0 \rightarrow \alpha, D \rangle$ such that $D(\langle X_0\ \textit{category} \rangle) = S$, where S is the start category of the grammar. Repeat the following steps for each vertex v_j such that $j = 0, \ldots, n$ until no more edges can be added to the chart.

Scan: If $w_j = a$, then for each lexical entry of the form $\langle X_0 \rightarrow \cdot\, a, D \rangle$ add an edge $\langle v_{j-1}, v_j, X_0 \rightarrow a\cdot, D \rangle$.

Predict (*top-down*): For each edge of the form $\langle v_i, v_j, X_0 \rightarrow \alpha \cdot X_m \beta, D \rangle$ and each rule of the form $\langle Y_0 \rightarrow \gamma, E \rangle$ such that the unification $E(\langle Y_0 \rangle) \sqcup D(\langle X_m \rangle)$ succeeds, add an edge $\langle v_j, v_j, Y_0 \rightarrow \cdot\, \gamma, E' \rangle$ unless it is subsumed by any existing edge. E' is the result on E of the restricted unification $E(\langle Y_0 \rangle) \sqcup D(\langle X_m \rangle)$.

Complete: For each edge of the form $\langle v_i, v_j, X_0 \rightarrow \alpha \cdot X_m \beta, D \rangle$ and each edge of the form $\langle v_j, v_k, Y_0 \rightarrow \gamma\cdot, E \rangle$, add the edge $\langle v_i, v_k, X_0 \rightarrow \alpha X_m \cdot \beta, D' \rangle$ if the unification $E(\langle Y_0 \rangle) \sqcup D(\langle X_m \rangle)$ succeeds and this edge is not subsumed by any existing edge. D' is the result on D of the unification $E(\langle Y_0 \rangle) \sqcup D(\langle X_m \rangle)$.

Fig. 10 A scheme for chart parsing with top-down prediction (Earley-style), using constraint-based grammar.

7. LR PARSING

An LR parser is a type of shift–reduce parser that was originally devised for programming languages [56]. An LR parser constructs the rightmost derivation of a syntactic analysis in reverse. In fact, the "R" in "LR" stands for rightmost-derivation in reverse. The "L" stands for left-to-right scanning of the input string. The success of LR parsing lies in handling a number of production rules simultaneously by the use of prefix merging (compare, the end of the next section), rather than attempting to apply one rule at a time.

The use of LR parsing techniques for natural languages involves adapting them to high-coverage grammars that allow large amounts of syntactic ambiguity, and thus result in nondeterminism when parsing. The most important extensions to the original LR-parsing scheme addressing these issues are:

- Generalized LR (GLR) parsing, developed by Tomita [106], but whose theoretical foundation was provided by Lang [59], avoids exponential search by adopting a quasiparallel deterministic processing scheme, packing local ambiguity in the process. This is discussed in Sec. 7.B.
- Using constraint-based grammars employing complex-valued features to reduce the amount of (spurious) ambiguity [see, e.g., 66, 82]. This is discussed in Sec. 7.C.

Useful references to deterministic LR parsing and LR compilation are Sippu and Soisalon-Soininen [98] and Aho et al. [3; pp. 215–266]. We will here first discuss simple backtracking LR parsing.

A. Basic LR Parsing

An LR parser is basically a pushdown automaton, that is, it has a stack in addition to a finite set of internal states and a reader head for scanning the input string from left to right, one symbol at a time. The stack is used in a characteristic way: the items on the stack consist of alternating grammar symbols and states. The current state is simply the state on top of the stack. The most distinguishing feature of an LR parser, however, is the form of the transition relation—the action and goto tables. A nondeterministic LR parser can in each step perform one of four basic actions. In state S_i with lookahead symbol[13] X it can:

1. `accept`: Halt and signal success.
2. `error`: Fail and backtrack.
3. `shift` S_k: Consume the input symbol X, push it onto the stack, and transit to state S_k by pushing it onto the stack.
4. `reduce`: R_n: Pop off two items from the stack for each phrase in the RHS of grammar rule R_n, inspect the stack for the old state S_j now on top of the stack, push the LHS Y of the rule R_n onto the stack, and transit to state S_k determined by `goto` S_j Y S_k by pushing S_k onto the stack.

This is not a pushdown automaton according to some formal definitions found in the literature [e.g., 70, pp. 491–492], because the reduce actions do not consume any input symbols. From a theoretical point of view, this is not important, for it does not change the expressive power of the machine. From a practical point of view, this gives the parser the possibility of not halting by reducing empty (ϵ) productions and transiting back to the same state; care must be taken to avoid this. This is a variant of the well-known problem of bottom-up parsing in conjunction with empty productions.

Prefix merging is accomplished by having each internal state correspond to a set of dotted rules (as defined in Sec. 6), also called *items*[14]. For example, if the grammar contains the following three rules,

$$VP \rightarrow V \quad VP \rightarrow V\ NP \quad VP \rightarrow V\ NP\ NP$$

there will be a state containing the items

$$VP \rightarrow V \cdot \quad VP \rightarrow V \cdot NP \quad VP \rightarrow V \cdot NP\ NP$$

This state corresponds to just having found a verb (V). Which of the three rules to apply in the end will be determined by the rest of the input string; at this point no commitment has been made to any of them.

1. LR-Parsed Example

The example grammar of Fig. 11 will generate the internal states of Fig. 12. These in turn give rise to the parsing tables of Fig. 13. The entry `s2` in the action table, for example, should be interpreted as "shift the lookahead symbol onto the stack and transit to state S_2." The action entry `r7` should be interpreted as "reduce by rule R_7." The goto entries simply indicate what state to transit to once a phrase of that

[13] The lookahead symbol is the next symbol in the input string, that is, the symbol under the reader head.
[14] Not to be confused with stack items, or Earley items, compare the beginning of Sec. 6.B.