Cognitive Technologies

Pierre M. Nugues

Language Processing with Perl and Prolog

Theories, Implementation, and Application

Second Edition

Springer

Cognitive Technologies

For further volumes:
http://www.springer.com/series/5216

Pierre M. Nugues

Language Processing with Perl and Prolog

Theories, Implementation, and Application

Second Edition

Springer

Pierre M. Nugues
Department of Computer Science
Lund University
Lund, Sweden

ISSN 1611-2482 Cognitive Technologies ISSN 2197-6635 (electronic)
ISBN 978-3-642-41463-3 ISBN 978-3-642-41464-0 (eBook)
DOI 10.1007/978-3-642-41464-0
Springer Heidelberg New York Dordrecht London

Library of Congress Control Number: 2014945101

À mes parents,
À Madeleine

Preface to the Second Edition

Eight years, from 2006 to 2014, is a very long time in computer science. The trends I described in the preface of the first edition have not only been confirmed, but accelerated. I tried to reflect this with a complete revision of the techniques exposed in this book: I redesigned or updated all the chapters, I introduced two new ones, and, most notably, I considerably expanded the sections using machine-learning techniques. To make place for them, I removed a few algorithms of lesser interest. This enabled me to keep the size of the book to ca. 700 pages. The programs and companion slides are available from the book web site at http://ilppp.cs.lth.se/.

This book corresponds to a course in natural language processing offered at Lund University. I am grateful to all the students who took it and helped me write this new edition through their comments and questions. Curious readers can visit the course site at http://cs.lth.se/EDAN20/ and see how we use this book in a teaching context.

I would like to thank the many readers of the first edition who gave me feedback or reported errors, the anonymous copy editor of the first and second editions, Richard Johansson and Michael Covington for their suggestions, as well as Peter Exner, the PhD candidate I supervised during this period, for his enthusiasm. Special thanks go to Ronan Nugent, my editor at Springer, for his thorough review and copyediting along with his advice on style and content.

This preface would not be complete without a word to those who passed away, my aunt, Madeleine, and my father, Pierre. There is never a day I do not think of you.

Lund, Sweden
April 2014

Pierre Nugues

Preface

In the past 20 years, natural language processing and computational linguistics have considerably matured. The move has mainly been driven by the massive increase of textual and spoken data and the need to process them automatically. This dramatic growth of available data spurred the design of new concepts and methods, or their improvement, so that they could scale up from a few laboratory prototypes to proven applications used by billions of people. Concurrently, the speed and capacity of machines became an order of magnitude larger, enabling us to process gigabytes of data and billions of words in a reasonable time, to train, test, retrain, and retest algorithms like never before. Although systems entirely dedicated to language processing remain scarce, there are now scores of applications that, to some extent, embed language processing techniques.

The industry trend, as well as the user's wishes, toward information systems able to process textual data has made language processing a new requirement for many computer science students. This has shifted the focus of textbooks from readers being mostly researchers or graduate students to a larger public, from readings by specialists to pragmatism and applied programming. Natural language processing techniques are not completely stable, however. They consist of a mix that ranges from well-mastered and routine to rapidly changing. This makes the existence of a new book an opportunity as well as a challenge.

This book tries to take on this challenge and find the right balance. It adopts a hands-on approach. It is a basic observation that many students have difficulties going from an algorithm exposed using pseudocode to a runnable program. I did my best to bridge the gap and provide the students with programs and ready-made solutions. The book contains real code the reader can study, run, modify, and run again. I chose to write examples in two languages to make the algorithms easy to understand and encode: Perl and Prolog.

One of the major driving forces behind the recent improvements in natural language processing is the increase of text resources and annotated data. The huge amount of texts made available by the Internet and never-ending digitization led many practitioners to evolve from theory-oriented, armchair linguists to frantic empiricists. This books attempts as well as it can to pay attention to this trend and

stresses the importance of corpora, annotation, and annotated corpora. It also tries to go beyond English only and expose examples in two other languages, namely French and German.

The book was designed and written for a quarter or semester course. At Lund, I used it when it was still in the form of lecture notes in the EDA171 course. It comes with a companion web site where slides, programs, corrections, an additional chapter, and Internet pointers are available: http://www.cs.lth.se/~pierre/ilppp/. All the computer programs should run with Perl (available from www.perl.com) or Prolog. Although I only tested the programs with SWI Prolog available from www.swi-prolog.org, any Prolog compatible with the ISO reference should apply.

Many people helped me during the last 10 years when this book took shape, step-by-step. I am deeply indebted to my colleagues and to my students in classes at Caen, Nottingham, Stafford, Constance, and now in Lund. Without them, it could never have existed. I would like most specifically to thank the PhD students I supervised, in chronological order, Pierre-Olivier El Guedj, Christophe Godéreaux, Dominique Dutoit, and Richard Johansson.

Finally, my acknowledgments would not be complete without the names of the people I most cherish and who give meaning to my life: my wife, Charlotte, and my children, Andreas and Louise.

Lund, Sweden Pierre Nugues
January 2006

Contents

Chapter 1
An Overview of Language Processing

Γνῶθι σεαυτόν
'Know thyself'
Inscription at the entrance to Apollo's Temple at Delphi

1.1 Linguistics and Language Processing

Linguistics is the study and the description of human languages. Linguistic theories on grammar and meaning have developed since ancient times and the Middle Ages. However, modern linguistics originated at the end of the nineteenth century and the beginning of the twentieth century. Its founder and most prominent figure was probably Ferdinand de Saussure (1916). Over time, modern linguistics has produced an impressive set of descriptions and theories.

Computational linguistics is a subset of both linguistics and computer science. Its goal is to design mathematical models of language structures enabling the automation of language processing by a computer. From a linguist's viewpoint, we can consider computational linguistics as the formalization of linguistic theories and models or their implementation in a machine. We can also view it as a means to develop new linguistic theories with the aid of a computer.

From an applied and industrial viewpoint, language and speech processing, which is sometimes referred to as natural language processing (NLP), natural language understanding (NLU), or language technology, is the mechanization of human language faculties. People use language every day in conversations by listening and talking, or by reading and writing. It is probably our preferred mode of communication and interaction. Ideally, automated language processing would enable a computer to understand texts or speech and to interact accordingly with human beings.

Understanding or translating texts automatically and talking to an artificial conversational assistant are major challenges for the computer industry. Although this final goal has not been reached yet, in spite of constant research, it is being

P.M. Nugues, *Language Processing with Perl and Prolog*, Cognitive Technologies,
DOI 10.1007/978-3-642-41464-0_1, © Springer-Verlag Berlin Heidelberg 2014

approached every day, step-by-step. Even if we have missed Stanley Kubrick's prediction of talking electronic creatures in the year 2001, language processing and understanding techniques have already achieved results ranging from very promising to near-perfect. The description of these techniques is the subject of this book.

1.2 Applications of Language Processing

At first, language processing is probably easier understood by the description of a result to be attained rather than by the analytical definition of techniques. Ideally, language processing would enable a computer to analyze huge amounts of text and to understand them; to communicate with us in a written or a spoken way; to capture our words whatever the entry mode: through a keyboard or through a speech recognition device; to parse our sentences; to understand our utterances, to answer our questions, and possibly to have a discussion with us – the human beings.

Language processing has a history nearly as old as that of computers, and it comprises a large body of work. However, many early attempts remained in the stage of laboratory demonstrations or simply failed. Significant applications have been slow to come, and they are still relatively scarce compared with the universal deployment of some other technologies such as operating systems, databases, and networks. Nevertheless, the number of commercial applications or significant laboratory prototypes embedding language processing techniques is increasing. Examples include:

Spelling and grammar checkers. These programs are now ubiquitous in text processors, and hundred of millions of people use them every day. Spelling checkers are based primarily on computerized dictionaries, and they remove most misspellings that occur in documents. Grammar checkers, although not perfect, have improved to a point that many users could not write a single e-mail without them. Grammar checkers use rules to detect common grammar and style errors (Jensen et al. 1993).

Text indexing and information retrieval from the Internet. These programs are among the most popular of the Web. They are based on crawlers that visit internet sites and that download texts they contain. Crawlers track the links occurring on the pages and thus explore the Web. Many of these systems carry out a full text indexing of the pages. Users ask questions and text retrieval systems return the internet addresses of documents containing words of the question. Using statistics on words or popularity measures, text retrieval systems are able to rank the documents (Brin and Page 1998; Salton 1988).

Speech transcription. These systems are based on speech recognition. Instead of typing using a keyboard, speech dictation systems allow a user to dictate reports and transcribe them automatically into a written text. Systems like Microsoft's *Windows Speech Recognition* or Google's *Voice Search* have high performance and recognize English, French, German, Spanish, Italian, Japanese, Chinese, etc.

Some systems transcribe radio and TV broadcast news with a word-error rate lower than 10 % (Nguyen et al. 2004).

Voice control of domestic devices such as videocassette recorders or disc changers (Ball et al. 1997). These systems are embedded in objects to provide them with a friendlier interface. Many people find electronic devices complicated and are unable to use them satisfactorily. A spoken interface would certainly be an easier means to control them. Although there are commercial systems available, few of them are fully usable. One challenge they still have to overcome is to operate in noisy environments that impair speech recognition.

Interactive voice response applications. These systems deliver information over the telephone using speech synthesis or prerecorded messages. In more traditional systems, users interact with the application using touch-tone telephones. More advanced servers have a speech recognition module that enables them to understand spoken questions or commands from users. Early examples of speech servers include travel information and reservation services (Mast et al. 1994; Sorin et al. 1995). Although most servers are just interfaces to existing databases and have limited reasoning capabilities, they have spurred significant research on dialogue, speech recognition, and synthesis.

Machine translation. Research on machine translation is one of the oldest domains of language processing. One of its outcomes is the venerable SYSTRAN program that started with translations between English and Russian for the US Department of Defense. Since then, machine translation has been extended to many other languages and has become a mainstream NLP application: *Google Translate* now supports more than 60 languages and is used by more than 200 million people every month (Och 2012). Another pioneer example is the *Spoken Language Translator* that translated spoken English into spoken Swedish in a restricted domain in real time (Agnäs et al. 1994; Rayner et al. 2000).

Conversational agents. Conversational agents are elaborate dialogue systems that have understanding faculties. An example is TRAINS that helps a user plan a route and the assembling trains: boxcars and engines to ship oranges from a warehouse to an orange juice factory (Allen et al. 1995). Ulysse is another example that uses speech to navigate into virtual worlds (Godéreaux et al. 1996, 1998).

Question answering. Question answering systems reached a milestone in 2011 when *IBM Watson* outperformed all its human contestants in the *Jeopardy!* quiz show (Ferrucci 2012). Watson answers questions in any domain posed in natural language using knowledge extracted from Wikipedia and other textual sources, encyclopedias, dictionaries, as well as databases such as WordNet, DBpedia, and Yago (Fan et al. 2012).

Some of these applications are widespread, like spelling and grammar checkers. Others are not yet ready for industrial exploitation or are still too expensive for popular use. They generally have a much lower distribution. Unlike other computer programs, results of language processing techniques rarely hit a 100 % success rate. Speech recognition systems are a typical example. Their accuracy is assessed in statistical terms. Language processing techniques become mature and usable when

they operate above a certain precision and at an acceptable cost. However, common to these techniques is that they are continuously improving and they are rapidly changing our way of interacting with machines.

1.3 The Different Domains of Language Processing

Historically linguistics has been divided into disciplines or levels, which go from sounds to meaning. Computational processing of each level involves different techniques such as signal and speech processing, statistics and machine learning, automaton theory, parsing, first-order logic, and automated reasoning.

A first discipline of linguistics is **phonetics**. It concerns the production and perception of acoustic sounds that form the speech signal. In each language, sounds can be classified into a finite set of **phonemes**. Traditionally, they include **vowels**: *a*, *e*, *i*, *o*; and **consonants**: *p*, *f*, *r*, *m*. Phonemes are assembled into **syllables**: *pa*, *pi*, *po*, to build up the words.

A second level concerns the **words**. The word set of a language is called a **lexicon**. Words can appear in several forms, for instance, the singular and the plural forms. **Morphology** is the study of the structure and the forms of a word. Usually a lexicon consists of root words. Morphological rules can modify or transform the root words to produce the whole vocabulary.

Syntax is a third discipline in which the order of words in a sentence and their relationships is studied. Syntax defines word categories and functions. Subject, verb, object is a sequence of functions that corresponds to a common order in many European languages including English and French. However, this order may vary, and the verb is often located at the end of the sentence in German. **Parsing** determines the structure of a sentence and assigns functions to words or groups of words.

Semantics is a fourth domain of linguistics. It considers the meaning of words and sentences. The concept of "meaning" or "signification" can be controversial. Semantics is differently understood by researchers and is sometimes difficult to describe and process. In a general context, semantics could be envisioned as a medium of our thought. In applications, semantics often corresponds to the determination of the sense of a word or the representation of a sentence in a logical format.

Pragmatics is a fifth discipline. While semantics is related to universal definitions and understandings, pragmatics restricts it – or complements it – by adding a contextual interpretation. Pragmatics is the meaning of words and sentences in specific situations.

The production of language consists of a stream of sentences that are linked together to form a **discourse**. This discourse is usually aimed at other people who can answer – it is to be hoped – through a **dialogue**. A dialogue is a set of linguistic interactions that enables the exchange of information and sometimes eliminates misunderstandings or ambiguities.

Fig. 1.1 A speech signal corresponding to *This is* [ðɪs ɪz]

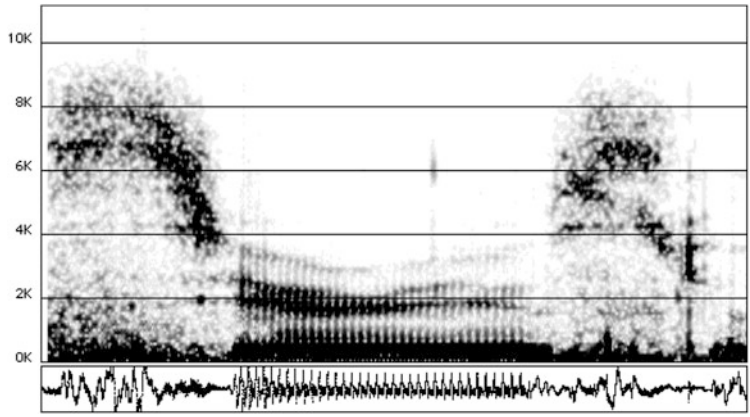


Fig. 1.2 A spectrogram corresponding to the word *serious* [sɪəriəs]

1.4 Phonetics

Sounds are produced through vibrations of the vocal cords. Several cavities and organs modify vibrations: the vocal tract, the nose, the mouth, the tongue, and the teeth. Sounds can be captured using a microphone. They result in signals such as that in Fig. 1.1.

A speech signal can be sampled and digitized by an analog-to-digital converter. It can then be processed and transformed by a Fourier analysis (FFT) in a moving window, resulting in spectrograms (Figs. 1.2 and 1.3). Spectrograms represent the distribution of speech power within a frequency domain ranging from 0 to 10,000 Hz over time. This frequency domain corresponds roughly to the sound production possibilities of human beings.

Phoneticians can "read" spectrograms, that is, split them into a sequence of relatively regular – stationary – patterns. They can then annotate the corresponding segments with phonemes by recognizing their typical patterns.

A descriptive classification of phonemes includes:

- Simple vowels such as /ɪ/, /a/, and /ɛ/, and nasal vowels in French such as /ɑ̃/ and /ɔ̃/, which appear on the spectrogram as a horizontal bar – the fundamental frequency – and several superimposed horizontal bars – the harmonics.

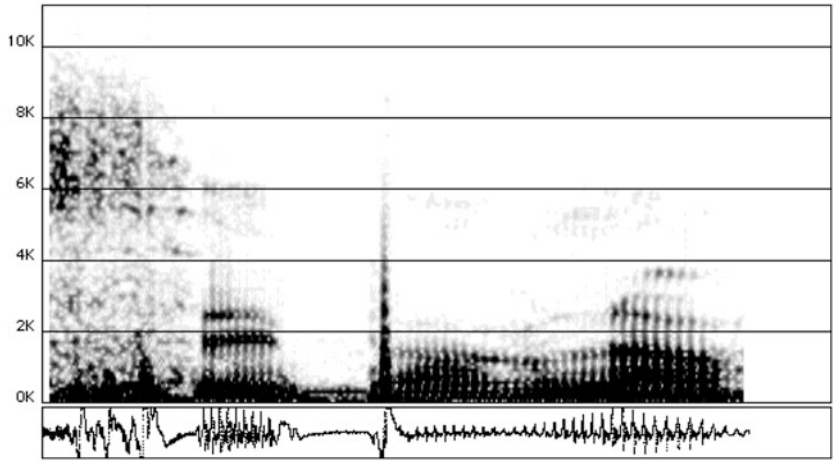


Fig. 1.3 A spectrogram of the French phrase *C'est par là* [separla] 'It is that way'

- Plosives such as /p/ and /b/ correspond to a stop in the airflow and then a very short and brisk emission of air from the mouth. The air release appears as a vertical bar from 0 to 5,000 Hz.
- Fricatives such as /s/ and /f/ that appear as white noise on the spectrogram, that is, as a uniform gray distribution. Fricatives sound a bit like a loudspeaker with an unplugged signal cable.
- Nasals and approximants such as /m/, /l/, and /r/ are more difficult to spot and are subject to modifications according to their left and right neighbors.

The pronunciation of a word is basically carried out through its **syllables**, phonetic segments formed of a vowel and one or more consonants. These syllables are more or less stressed or emphasized, and are influenced by neighboring syllables.

The general rhythm of the sentence is the **prosody**. Prosody is quite different from English to French and German and is an open subject of research. It is related to the length and structure of sentences, to questions, and to the meaning of the words.

Speech synthesis uses signal processing techniques, phoneme models, and letter-to-phoneme rules to convert a text into speech and to read it in a loud voice. **Speech recognition** does the reverse and transcribes speech into a computer-readable text. It also uses signal processing and statistical techniques including hidden Markov models (HMM) and language models.

1.5 Lexicon and Morphology

The set of available words in a given context makes up a lexicon. It varies from language to language and within a language according to the context or genre: fiction, news, scientific literature, jargon, slang, or gobbledygook. Every word can

Table 1.1 Grammatical features that modify the form of a word

Features	Values	English	French	German
Number	Singular	*a car*	*une voiture*	*ein Auto*
	Plural	*two cars*	*deux voitures*	*zwei Autos*
Gender	Masculine	*he*	*il*	*er*
	Feminine	*she*	*elle*	*sie*
	Neuter	*it*		*es*
Conjugation	Infinitive	*to work*	*travailler*	*arbeiten*
and	Finite	*he works*	*il travaille*	*er arbeitet*
tense	Gerund	*working*	*travaillant*	*arbeitend*

Table 1.2 Examples of word derivations

	Words	Derived words
English	*real*/adjective	*really*/adverb
French	*courage*/noun	*courageux*/adjective
German	*Der Mut*/noun	*mutig*/adjective

be classified through a lexical category or **part of speech** such as article, noun, verb, adjective, adverb, conjunction, preposition, or pronoun. Most of the lexical entities come from four categories: noun, verb, adjective, and adverb. Other categories such as articles, pronouns, or conjunctions have a limited and stable number of elements. Words in a sentence can be annotated – tagged – with their part of speech.

For instance, the simple sentences in English, French, and German:

The big cat ate the gray mouse
Le gros chat mange la souris grise
Die große Katze ißt die graue Maus

are annotated as:

The/article *big*/adjective *cat*/noun *ate*/verb *the*/article *gray*/adjective *mouse*/noun
Le/article *gros*/adjectif *chat*/nom *mange*/verbe *la*/article *souris*/nom *grise*/adjectif
Die/Artikel *große*/Adjektiv *Katze*/Substantiv *ißt*/Verb *die*/Artikel *graue*/Adjektiv *Maus*/Substantiv

Morphology is the study of how root words and affixes – the **morphemes** – are composed to form words. Morphology can be divided into **inflection** and **derivation**:

- Inflection is the form variation of a word under certain grammatical conditions. In European languages, these conditions consist notably of the number, gender, conjugation, or tense (Table 1.1).
- Derivation combines affixes to an existing root or stem to form a new word. Derivation is more irregular and complex than inflection. It often results in a change in the part of speech for the derived word (Table 1.2).

Most of the inflectional morphology of words can be described through morphological rules, possibly with a set of exceptions. According to these rules,

Table 1.3 Decomposition of inflected words into a root and affixes

	Words	**Roots and affixes**	**Lemmas and grammatical interpretations**
English	*worked*	*work* + *ed*	*work* + verb + preterit
French	*travaillé*	*travaill* + *é*	*travailler* + verb + past participle
German	*gearbeitet*	*ge* + *arbeit* + *et*	*arbeiten* + verb + past participle

a morphological parser splits each word as it occurs in a text into morphemes – the root word and the affixes. When affixes have a grammatical content, morphological parsers generally deliver this content instead of the raw affixes (Table 1.3).

Morphological parsing operates on single words and does not consider the surrounding words. Sometimes, the form of a word is ambiguous. For instance, *worked* can be found in *he worked* (*to work* and preterit) or *he has worked* (*to work* and past participle). Another processing stage is necessary to remove the ambiguity and to assign (to annotate) each word with a single part-of-speech tag.

A lexicon may simply be a list of all the **inflected** word forms – a wordlist – as they occur in running texts. However, keeping all the forms, for instance, *work*, *works*, *worked*, generates a useless duplication. For this reason, many lexicons retain only a list of canonical words: the **lemmas**. Lemmas correspond to the entries of most ordinary dictionaries. Lexicons generally contain other features, such as the phonetic transcription, part of speech, morphological type, and definition, to facilitate additional processing. Lexicon building involves collecting most of the words of a language or of a domain. Nonetheless, it is probably impossible to build an exhaustive dictionary since new words are appearing every day.

Morphological rules enable us to generate all the word forms from a lexicon. Morphological parsers do the reverse operation and retrieve the word root and its affixes from its inflected or derived form in a text. Morphological parsers use finite-state automaton techniques. Part-of-speech taggers disambiguate the possible multiple readings of a word. They also use finite-state automata or statistical techniques.

1.6 Syntax

Syntax governs the formation of a sentence from words. Syntax is sometimes combined with morphology under the term morphosyntax. Syntax has been a central point of interest of linguistics since the Middle Ages, but it probably reached an apex in the 1970s, when it captured an overwhelming amount of attention in the linguistics community.

1.6.1 Syntax as Defined by Noam Chomsky

Chomsky (1957) had a determining influence in the study of language, and his views still fashion the way syntactic formalisms are taught and used today. Chomsky's

theory postulates that syntax is independent from semantics and can be expressed in terms of logic grammars. These grammars consist of a set of rules that describe the sentence structure of a language. In addition, grammar rules can generate the whole sentence set – possibly infinite – of a definite language.

Generative grammars consist of syntactic rules that fractionate a phrase into subphrases and hence describe a sentence composition in terms of phrase structure. Such rules are called **phrase-structure rules**. An English sentence typically comprises two main phrases: a first one built around a noun called the noun phrase, and a second one around the main verb called the verb phrase. Noun and verb phrases are rewritten into other phrases using other rules and by a set of terminal symbols representing the words.

Formally, a grammar describing a very restricted subset of English, French, or German phrases could be the following rule set:

- A **sentence** consists of a **noun phrase** and a **verb phrase**.
- A **noun phrase** consists of an **article** and a **noun**.
- A **verb phrase** consists of a **verb** and a **noun phrase**.

A very limited lexicon of the English, French, or German words could be made of:

- Articles such as *the*, *le*, *la*, *der, den*
- Nouns such as *boy*, *garçon*, *Knabe*
- Verbs such as *hit*, *frappe*, *trifft*

This grammar generates sentences such as:

The boy hit the ball
Le garçon frappe la balle
Der Knabe trifft den Ball

but also incorrect or implausible sequences such as:

The ball hit the ball
*Le balle frappe la garçon
*Das Ball trifft den Knabe

Linguists use an asterisk (*) to indicate an ill-formed grammatical construction or a nonexistent word. In the French and German sentences, the articles must agree with their nouns in gender, number, and case (for German). The correct sentences are:

La balle frappe le garçon
Der Ball trifft den Knaben

Trees can represent the syntactic structure of sentences (Figs. 1.4–1.6) and reflect the rules involved in sentence generation. Moreover, Chomsky's formalism enables some transformations: rules can be set to carry out the building of an interrogative sentence from a declaration, or the building of a passive form from an active one.

Parsing is the reverse of generation, where a grammar, a set of phrase-structure rules, accepts syntactically correct sentences and determines their structure. Parsing

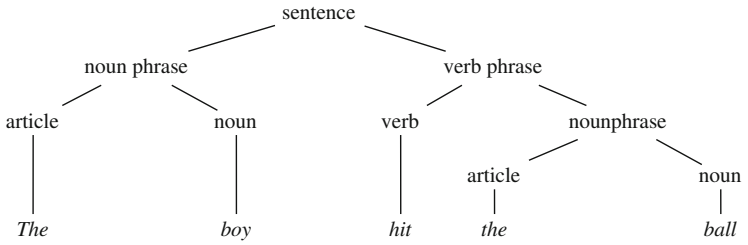


Fig. 1.4 Tree structure of *The boy hit the ball*

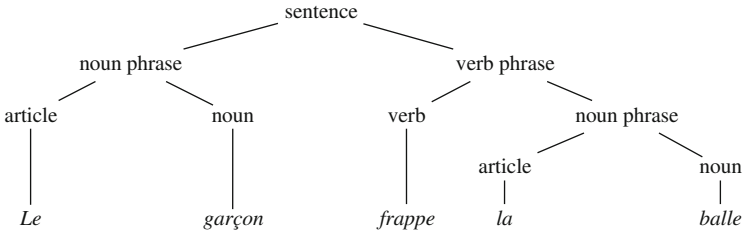


Fig. 1.5 Tree structure of *Le garçon frappe la balle*

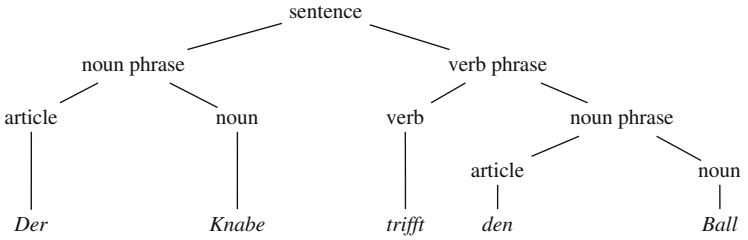


Fig. 1.6 Tree structure of *Der Knabe trifft den Ball*

requires a mechanism to search the rules that describe the sentence's structure. This mechanism can be applied from the sentence's words up to a rule describing the sentence's structure. This is **bottom-up parsing**. Rules can also be searched from a sentence structure rule down to the sentence's words. This corresponds to **top-down parsing**.

1.6.2 Syntax as Relations and Dependencies

Before Chomsky, pupils and students learned syntax (and still do so) mainly in terms of functions and relations between the words. A sentence's classical parsing consists in annotating words using parts of speech and in identifying the main verb. The main verb is the pivot of the sentence, and the principal grammatical functions

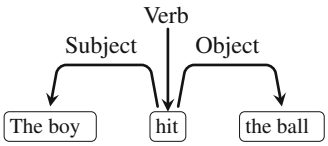


Fig. 1.7 Grammatical relations in the sentence *The boy hit the ball*

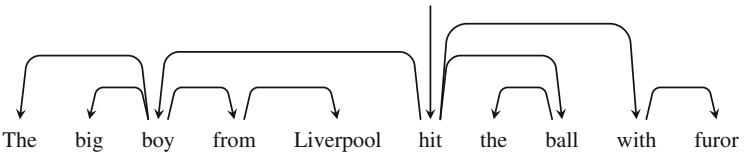


Fig. 1.8 Dependency relations in the sentence *The big boy from Liverpool hit the ball with furor*

are determined relative to it. Parsing consists then in grouping words to form the subject and the object, which are the two most significant functions in addition to the verb.

In the sentence *The boy hit the ball*, the main verb is *hit*, the subject of *hit* is *the boy*, and its object is *the ball* (Fig. 1.7).

Other grammatical functions (or relations) involve notably articles, adjectives, and adjuncts. We see this in the sentence

The big boy from Liverpool hit the ball with furor.

where the adjective *big* is related to the noun *boy*, and the adjuncts *from Liverpool* and *with furor* are related respectively to *boy* and *hit*.

We can picture these relations as a dependency net, where each word is said to modify exactly another word up to the main verb (Fig. 1.8). The main verb is the head of the sentence and modifies no other word. Tesnière (1966) has extensively described dependency theory.

Recently, **dependency grammars** have enjoyed a growing popularity as they can efficiently handle multiple languages and have a good interface to the semantic level. They provide a theoretical framework to many current parsing techniques and have numerous applications.

1.7 Semantics

The semantic level is more difficult to capture, and there are numerous viewpoints on how to define and to process it. A possible viewpoint is to oppose it to syntax: there are sentences that are syntactically correct but that cannot make sense. Such a description of semantics would encompass sentences that make sense. Classical examples by Chomsky (1957) – sentences 1 and 2 – and Tesnière (1966) – sentence 3 – include:

Table 1.4 Correspondence between sentences and logical forms

Sentences	Logical forms (predicates)
Pierre wrote notes	`wrote(pierre, notes).`
Pierre a écrit des notes	`a_écrit(pierre, notes).`
Pierre schrieb Notizen	`schrieb(pierre, notizen).`

1. *Colorless green ideas sleep furiously*.
2. **Furiously sleep ideas green colorless*.
3. *Le silence vertébral indispose la voile licite*.
   'The vertebral silence embarrasses the licit sail.'

Sentences 1 and 3 and are syntactically correct but have no meaning, while sentence 2 is neither syntactically nor semantically correct.

In computational linguistics, semantics is often related to logic and to predicate calculus. Determining the semantic representation of a sentence then involves turning it into a predicate–argument structure, where the predicate is the main verb and the arguments correspond to phrases accompanying the verb such as the subject and the object. This type of logical representation is called a **logical form**. Table 1.4 shows examples of sentences together with their logical forms.

Representation is only one facet of semantics. Once sentence representations have been built, they can be interpreted to check what they mean. *Notes* in the sentence *Pierre wrote notes* can be linked to a dictionary **definition**. If we look up *notes* in the *Cambridge International Dictionary of English* (Procter 1995), we find as many as five possible senses for it (abridged from p. 963):

1. **note** [WRITING], *noun*, a short piece of writing;
2. **note** [SOUND], *noun*, a single sound at a particular level;
3. **note** [MONEY], *noun*, a piece of paper money;
4. **note** [NOTICE], *verb*, to take notice of;
5. **note** [IMPORTANCE], *noun*, of note: of importance.

So linking a word meaning to a definition is not straightforward because of possible ambiguities. Among these definitions, the intended sense of *notes* is a specialization of the first entry:

> **notes**, *plural noun*, notes are written information.

Finally, we can interpret *notes* as what they refer to concretely, that is, a specific object: a set of bound paper sheets with written text on them or a file on a computer drive. The word *notes* is then the mention of an object of the real world, here a file on a computer, and linking the mention and the object is called **reference resolution**.

The **referent** of the word *notes*, that is, the designated object or **entity**, could be the path /users/pierre/language_processing.html in Unix parlance. As for the definition of a word, the designated entity can be ambiguous. Let us suppose that a database contains the locations of the lecture notes Pierre wrote. In Prolog, listing its content could yield:

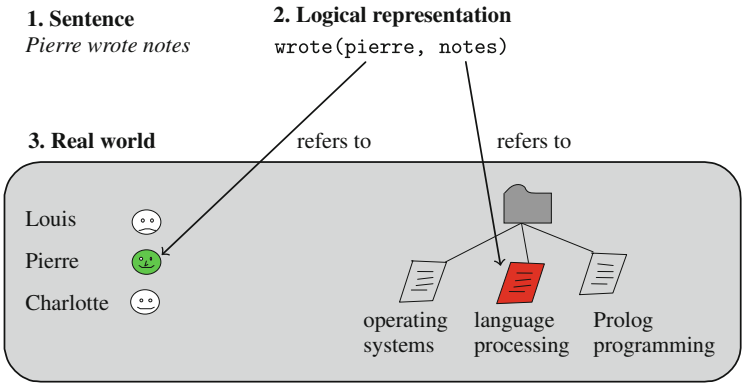


Fig. 1.9 Resolving references of *Pierre wrote notes*

```
notes('/users/pierre/operating_systems.html').
notes('/users/pierre/language_processing.html').
notes('/users/pierre/prolog_programming.html').
```

Here this would mean that finding the referent of *notes* consists in choosing a document among three possible ones (Fig. 1.9).

Obtaining the semantic structure of a sentence has been discussed abundantly in the literature. This is not surprising, given the uncertain nature of semantics. Building a logical form sometimes calls on the **composition** of the semantic representation of the phrases that constitute a sentence. To carry it out, we must assume that sentences and phrases have an internal representation that can be expressed in terms of a logical formula.

Once a representation has been built, a reasoning process is applied to resolve references and to determine whether a sentence is true or not. It generally involves rules of deduction, or **inferences**.

Pragmatics is semantics restricted to a specific context and relies on facts that are external to the sentence. These facts contribute to the inference of a sentence's meaning or prove its truth or falsity. For instance, the pragmatics of

> Methuselah lived to be 969 years old. (Genesis 5:27)

can make sense in the Bible but not elsewhere, given the current possibilities of medicine.

1.8 Discourse and Dialogue

An interactive conversational agent cannot be envisioned without considering the whole **discourse** of (human) users – or parts of it – and apart from a **dialogue** between a user and the agent. Discourse refers to a sequence of sentences, to

a sentence context in relation with other sentences, or with some background situation. It is often linked with pragmatics.

Discourse study also enables us to resolve references that are not self-explainable in single sentences. Pronouns are good examples of such missing information. In the sentence

John took it

the pronoun *it* can probably be related to an entity mentioned in a previous sentence, or is obvious given the context where this sentence was said. These references are given the name of **anaphors**.

Dialogue provides a means of communication. It is the result of two intermingled – and, we hope, interacting – discourses: one from the user and the other from the machine. It enables a conversation between the two parties, the assertion of new results, and the cooperative search for solutions.

Dialogue is also a tool to repair communication failures or to complete interactively missing data. It may clarify information and mitigate misunderstandings that impair communication. Through a dialogue a computer can respond and ask the user:

I didn't understand what you said! Can you repeat (rephrase)?

Dialogue easily replaces some hazardous guesses. When an agent has to find the potential reference of a pronoun or to solve reference ambiguities, the best option is simply to ask the user to clarify what s/he means:

Tracy? Do you mean James' brother or your mother?

Discourse processing splits texts and sentences into segments. It then sets links between segments to chain them rationally and to map them onto a sort of structure of the text. Discourse studies often make use of **rhetoric** as a background model of this structure.

Dialogue processing classifies the segments into what are called **speech acts**. At a first level, speech acts comprise dialogue turns: the user turn and the system turn. Then turns are split into sentences, and sentences into questions, declarations, requests, answers, etc. Speech acts can be modeled using finite-state automata or more elaborate schemes using **intention** and **planning** theories.

1.9 Why Speech and Language Processing Are Difficult

So far, for all the linguistic levels mentioned in the previous sections, we outlined models and techniques to process speech and language. They often enable machines to obtain excellent results compared to the performance of human beings. However, for most levels, language processing rarely hits the ideal score of 100 %. Among the

hurdles that often prevent the machine from reaching this figure, two recur at any level: ambiguity and the absence of a perfect model.

1.9.1 Ambiguity

Ambiguity is a major obstacle in language processing, and it may be the most significant. Although as human beings we are not aware of it most of the time, ambiguity is ubiquitous in language and plagues any stage of automated analysis. We saw examples of ambiguous morphological analysis and part-of-speech annotation, word senses, and references. Ambiguity also occurs in speech recognition, parsing, anaphora solving, and dialogue.

McMahon and Smith (1996) illustrate strikingly ambiguity in speech recognition with the sentence

The boys eat the sandwiches.

Speech recognition comprises generally two stages: first, a phoneme recognition, and then a concatenation of phoneme substrings into words. Using the International Phonetic Association (IPA) symbols, a perfect phonemic transcription of this utterance would yield the transcription:

[ˈðəbˈoɪzˈiːtˈðəsˈændwɪdʒɪz],

which shows eight other alternative readings at the word decoding stage:

*The boy seat the sandwiches.
*The boy seat this and which is.
*The boys eat this and which is.
The buoys eat the sandwiches.
*The buoys eat this and which is.
The boys eat the sand which is.
*The buoys seat this and which is.

This includes the strange sentence

The buoys eat the sand which is.

For syntactic and semantic layers, a broad classification occurs between lexical and structural ambiguity. Lexical ambiguity refers to multiple senses of words, while structural ambiguity describes a parsing alternative, as with the frequently quoted sentence

I saw the boy with a telescope,

which can mean either that I used a telescope to see the boy or that I saw the boy who had a telescope.

A way to resolve ambiguity is to use a conjunction of language processing components and techniques. In the example given by McMahon and Smith, five

out of eight possible interpretations are not grammatical. These are flagged with an asterisk. A further syntactic analysis could discard them.

Probabilistic models of word sequences can also address disambiguation. Statistics on word occurrences drawn from large quantities of texts – corpora – can capture grammatical as well as semantic patterns. Improbable alternatives <boys eat sand> and <buoys eat sand> are also highly unlikely in corpora and will not be retained (McMahon and Smith 1996). In the same vein, probabilistic parsing is a very powerful tool to rank alternative parse trees, that is, to retain the most probable and reject the others.

In some applications, logical rules model the context, reflect common sense, and discard impossible configurations. Knowing the physical context may help disambiguate some structures, as in the boy and the telescope, where both interpretations of the isolated sentence are correct and reasonable. Finally, when a machine interacts with a user, it can ask her/him to clarify an ambiguous utterance or situation.

1.9.2 Models and Their Implementation

Processing a linguistic phenomenon or layer starts with the choice or the development of a formal model and its algorithmic implementation. In any scientific discipline, good models are difficult to design. This is specifically the case with language. Language is closely tied to human thought and understanding, and in some instances models in computational linguistics also involve the study of the human mind. This gives a measure of the complexity of the description and the representation of language.

As noted in the introduction, linguists have produced many theories and models. Unfortunately, few of them have been elaborate enough to encompass and describe language effectively. Some models have also been misleading. This explains somewhat the failures of early attempts in language processing. In addition, many of the potential theories require massive computing power. Processors and storage able to support the implementation of complex models with substantial dictionaries, corpora, and parsers were not widely available until recently.

However, in the last decade models have matured, and computing power has become inexpensive. Although models and implementations are rarely (never?) perfect, they now enable us to obtain exploitable results. Most use a limited set of techniques that we will consider throughout this book, namely finite-state automata, logic grammars, and first-order logic. These tools are easily implemented in Prolog. Another set of tools pertains to the theory of probability, statistics, and machine learning. The combination of logic, statistics and machine-learning techniques now enables us to parse running-text sentences in multiple languages with an accuracy rate of more than 90 %, a figure that would have been unimaginable 15 years ago.

Table 1.5 An excerpt of a Persona dialogue (After Ball et al. (1997))

Turns	Utterances
	[Peedy is asleep on his perch]
User:	Good morning, Peedy.
	[Peedy rouses]
Peedy:	**Good morning.**
User:	Let's do a demo.
	[Peedy stands up, smiles]
Peedy:	**Your wish is my command, what would you like to hear?**
User:	What have you got by Bonnie Raitt?
	[Peedy waves in a stream of notes, and grabs one as they rush by.]
Peedy:	**I have "The Bonnie Raitt Collection" from 1990.**
User:	Pick something from that.
Peedy:	**How about "Angel from Montgomery"?**
User:	Sounds good.
	[Peedy drops note on pile]
Peedy:	**OK.**
User:	Play some rock after that.
	[Peedy scans the notes again, selects one]
Peedy:	**How about "Fools in Love"?**
User:	Who wrote that?
	[Peedy cups one wing to his 'ear']
Peedy:	**Huh?**
User:	Who wrote that?
	[Peedy looks up, scrunches his brow]
Peedy:	**Joe Jackson**
User:	Fine.
	[Drops note on pile]
Peedy:	**OK.**

1.10 An Example of Language Technology in Action: The Persona Project

1.10.1 Overview of Persona

The Persona prototype from Microsoft Research (Ball et al. 1997) illustrates a user interface that is based on a variety of language processing techniques. Persona is a conversational agent that helps a user select songs and music tracks from a record database. Peedy, an animated cartoonlike parrot, embodies the agent that interacts with the user. It contains speech recognition, parsing, and semantic analysis modules to listen and to respond to the user and to play the songs. Table 1.5 shows an example of a dialogue with Peedy.

Certain interactive talking assistants consider a limited set of the linguistic levels we have presented before. Simple systems bypass syntax, for example, and have

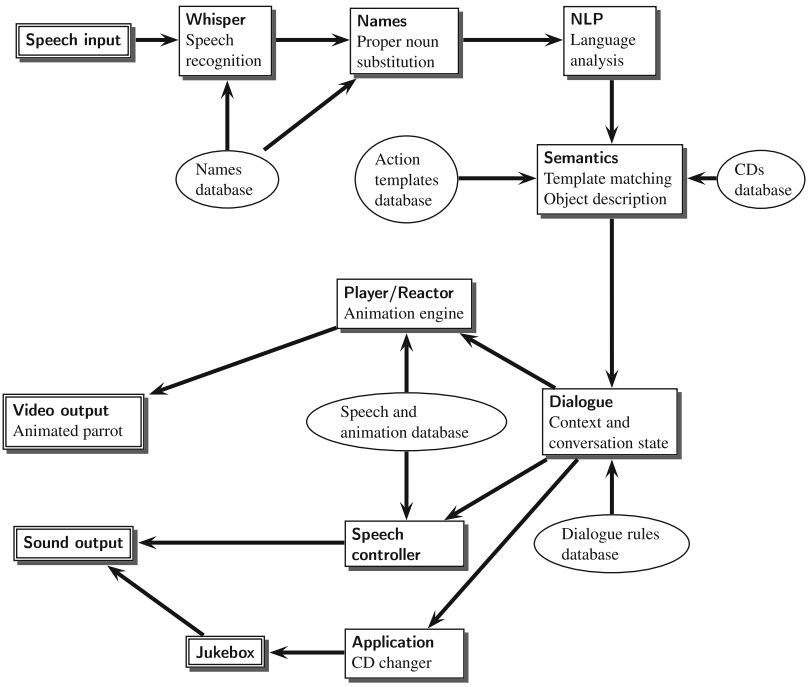


Fig. 1.10 Architecture of the Persona conversational assistant (After Ball et al. (1997))

only a speech recognition device to detect a couple of key words. In contrast, Persona has components to process more layers. They are organized in modules carrying out speech recognition, speech synthesis, parsing, semantics analysis, and dialogue. In addition, Persona has components specific to the application such as a name substitution module to find proper nouns like *Madonna* or *Debussy* and an animation module to play the Peedy character.

Persona's architecture organizes its modules into a pipeline processing flow (Fig. 1.10). Many other instances of dialogue systems adopt a similar architecture.

1.10.2 The Persona's Modules

Persona's first component is the Whisper speech recognition module (Huang et al. 1995). Whisper uses signal processing techniques to compare phoneme models to the acoustic waves, and it assembles the recognized phonemes into words. It also uses a grammar to constrain the recognition possibilities. Whisper transcribes continuous speech into a stream of words in real time. It is a speaker-independent system. This means that it operates with any speaker without training.

The user's orders to select music often contain names: artists, titles of songs, or titles of albums. The Names module extracts them from the text before they are passed on to further analysis. Names uses a pattern matcher that attempts to substitute all the names and titles contained in the input sentence with placeholders. The utterance *Play before you accuse me by Clapton* is transformed into *Play* `track1` *by* `artist1`.

The NLP module parses the input in which names have been substituted. It uses a grammar with rules similar to that of Sect. 1.6.1 and produces a tree structure. It creates a logical form whose predicate is the verb, and the arguments are the subject and the object: `verb(subject, object)`. The sentence *I would like to hear something* is transformed into the form `like(i, hear(i, something))`.

The logical forms are converted into a task graph representing the utterance in terms of actions the agent can do and objects of the task domain. It uses an application-dependent notation to map English words to symbols. It also reverses the viewpoint from the user to the agent. The logical form of *I would like to hear something* is transformed into the task graph: `verbPlay(you, objectTrack)` – *You play* (`verbPlay`) *a track* (`objectTrack`).

Each possible request Peedy understands has possible variations – paraphrases. The mapping of logical forms to task graphs uses transformation rules to reduce them to a limited set of 17 canonical requests. The transformation rules deal with synonyms, syntactic variation, and colloquialisms. The forms corresponding to

> I'd like to hear some Madonna.
> I want to hear some Madonna.
> It would be nice to hear some Madonna.

are transformed into a form equivalent to

> Let me hear some Madonna.

The resulting graph is matched against actions templates the jukebox can carry out.

The dialogue module controls Peedy's answers and reactions. It consists of a state machine that models a sequence of interactions. Depending on the state of the conversation and an input event – what the user says – Peedy will react: trigger an animation, utter a spoken sentence or play music, and move to another conversational state.

1.11 Further Reading

Introductory textbooks on linguistics include *An Introduction to Language* (Fromkin et al. 2010) and *Linguistics: An Introduction to Linguistics Theory* (Fromkin 2000). The *Nouveau dictionnaire encyclopédique des sciences du langage* (Ducrot and Schaeffer 1995) is an encyclopedic presentation of linguistics in French, and *Studienbuch Linguistik* (Linke et al. 2004) is an introduction in German. *Fondamenti*

di linguistica (Simone 2007) is an outstandingly clear and concise work in Italian that describes most fundamental concepts of linguistics.

Concepts and theories in linguistics evolved continuously from their origins to the present time. Historical perspectives are useful to understand the development of central issues. *A Short History of Linguistics* (Robins 1997) is a very readable introduction to linguistics history. *Histoire de la linguistique de Sumer à Saussure* (Malmberg 1991) and *Analyse du langage au XX^{e} siècle* (Malmberg 1983) are comprehensive and accessible books that review linguistic theories from the ancient Near East to the end of the twentieth century. *Landmarks in Linguistic Thought, The Western Tradition from Socrates to Saussure* (Harris and Taylor 1997) are extracts of founding classical texts followed by a commentary.

Available books on natural language processing include (in English): *Natural Language Processing in Prolog* (Gazdar and Mellish 1989), *Prolog for Natural Language Analysis* (Gal et al. 1991), *Natural Language Processing for Prolog Programmers* (Covington 1994b), *Natural Language Understanding* (Allen 1994), *Foundations of Statistical Natural Language Processing* (Manning and Schütze 1999), *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition* (Jurafsky and Martin 2008), *Foundations of Computational Linguistics: Human–Computer Communication in Natural Language* (Hausser 2014). Available books in French include: *Prolog pour l'analyse du langage naturel* (Gal et al. 1989), *L'intelligence artificielle et le langage* (Sabah 1990). And in German *Grundlagen der Computerlinguistik. Mensch-Maschine-Kommunikation in natürlicher Sprache* (Hausser 2000).

The Internet offers a wealth of resources: digital libraries, general references, corpus, lexical, and software resources, together with registries and portals. A starting point is the official home page of the Association for Computational Linguistics (ACL), which provides many links (http://www.aclweb.org/). The ACL anthology (http://www.aclweb.org/anthology/) is an extremely valuable anthology of research papers (journal and conferences) published under the auspices of the ACL. The French journal *Traitement automatique des langues* is also a source of interesting papers. It is published by the Association de traitement automatique des langues (http://www.atala.org/). Wikipedia (http://www.wikipedia.org/) is a free encyclopedia that contains definitions and general articles on concepts and theories used in computational linguistics and natural language processing.

Many source programs are available for download, either for free or under a license. They include speech synthesis and recognition, morphological analysis, parsing, and so on. The Natural Language Toolkit (NLTK) is an example that features a comprehensive suite of open source Python programs, data sets, and tutorials (http://nltk.org/). It has a companion book: *Natural Language Processing with Python* by Bird et al. (2009). The German Institute for Artificial Intelligence Research maintains a list of available software and related resources at the Natural Language Software Registry (http://registry.dfki.de/).

Lexical and corpus resources are now available in many languages. Valuable sites include the Linguistic Data Consortium of the University of Pennsylvania

(http://www.ldc.upenn.edu/) and the European Language Resources Association (http://www.elra.info/).

There are nice interactive online demonstrations covering speech synthesis, parsing, translation, and so on. Since sites are sometimes transient, we do not list them here. A good way to find them is to use search engines or directories like Google, Bing, or Yahoo.

Finally, some companies and laboratories are very active in language processing research. They include major software powerhouses like Google, IBM, Microsoft, Yahoo, and Xerox. The paper describing the Peedy animated character can be found at the Microsoft Research website (http://www.research.microsoft.com/).

Exercises

1.1. List some computer applications that are relevant to the domain of language processing.

1.2. Tag the following sentences using parts of speech you know:

The cat caught the mouse.
Le chat attrape la souris.
Die Katze fängt die Maus.

1.3. Give the morpheme list of: *sings*, *sung*, *chante*, *chantiez*, *singt*, *sang*. List all the possible ambiguities.

1.4. Give the morpheme list of: *unpleasant*, *déplaisant*, *unangenehm*.

1.5. Draw the tree structures of the sentences:

The cat caught the mouse.
Le chat attrape la souris.
Die Katze fängt die Maus.

1.6. Identify the main functions of these sentences and draw the corresponding dependency graph linking the words:

The cat caught the mouse.
Le chat attrape la souris.
Die Katze fängt die Maus.

1.7. Draw the dependency graphs of the sentences:

The mean cat caught the gray mouse on the table.
Le chat méchant attrape la souris grise sur la table.
Die böse Katze fängt die graue Maus auf dem Tisch.

1.8. Give examples of sentences that are:

- Syntactically incorrect
- Syntactically correct
- Syntactically and semantically correct

1.9. Give the logical forms of these sentences:

The cat catches the mouse.
Le chat attrape la souris.
Die Katze fängt die Maus.

1.10. List the components you think necessary to build a spoken dialogue system.

Chapter 2
Corpus Processing Tools

A. a. a.	Je.	I.a.	[A, a, a,] domine deus, ecce nescio loqui.
		XIIII.b.	[A, a, a, domine deus,] prophete dicunt eis.
	Eze.	IIII.d.	[A, a, a,] domine deus ecce anima mea non est
		XXI.a.	[A, a, a,] domine deus.
	Joel	I.c.	[A, a, a,] diei.
Aaron	exo.	IIII.c	[Aaron. . .] egredietur in occursum
		VII.a.	[Aaron frater tuus] erit propheta tuus.
		XVII.d.	[Aaron autem et] Hur sustentabant manus.
		XXIIII.d.	habetis Aaron et Hur vobiscum.
	...		

First lines from the third concordance to the Vulgate.
Abbreviations are spelled out for clarity.
Bibliothèque nationale de France. Manuscrit latin 515. Thirteenth century.

2.1 Corpora

A corpus, plural corpora, is a collection of texts or speech stored in an electronic machine-readable format. A few years ago, large electronic corpora of more than a million of words were rare, expensive, or simply not available. At present, huge quantities of texts are accessible in many languages of the world. They can easily be collected from a variety of sources, most notably the Internet, where corpora of hundreds of millions of words are within the reach of most computational linguists.

2.1.1 Types of Corpora

Some corpora focus on specific genres: law, science, novels, news broadcasts, electronic correspondence, or transcriptions of telephone calls or conversations.

P.M. Nugues, *Language Processing with Perl and Prolog*, Cognitive Technologies,
DOI 10.1007/978-3-642-41464-0_2, © Springer-Verlag Berlin Heidelberg 2014

Table 2.1 List of the most frequent words in present texts and in the book of Genesis (After Crystal (1997))

	English	French	German
Most frequent words in a collection of contemporary running texts	*the*	*de*	*der*
	of	*le* (article)	*die*
	to	*la* (article)	*und*
	in	*et*	*in*
	and	*les*	*des*
Most frequent words in Genesis	*and*	*et*	*und*
	the	*de*	*die*
	of	*la*	*der*
	his	*à*	*da*
	he	*il*	*er*

Others try to gather a wider variety of running texts. Texts collected from a unique source, say from scientific magazines, will probably be slanted toward some specific words that do not appear in everyday life. Table 2.1 compares the most frequent words in the book of Genesis and in a collection of contemporary running texts. It gives an example of such a discrepancy. The choice of documents to include in a corpus must then be varied to survey comprehensively and accurately a language usage. This process is referred to as balancing a corpus.

Balancing a corpus is a difficult and costly task. It requires collecting data from a wide range of sources: fiction, newspapers, technical, and popular literature. Balanced corpora extend to spoken data. The Linguistic Data Consortium (LDC) from the University of Pennsylvania and the European Language Resources Association (ELRA), among other organizations, distribute written and spoken corpus collections. They feature samples of magazines, laws, parallel texts in English, French, German, Spanish, Chinese, Arabic, telephone calls, radio broadcasts, etc.

In addition to raw texts, some corpora are annotated. Each of their words is labeled with a linguistic tag such as a part of speech or a semantic category. The annotation is done either manually or semiautomatically. Spoken corpora contain the transcription of spoken conversations. This transcription may be aligned with the speech signal and sometimes includes prosodic annotation: pause, stress, etc. Annotation tags, paragraph and sentence boundaries, parts of speech, syntactic or semantic categories follow a variety of standards, which are called markup languages.

Among annotated corpora, treebanks deserve a specific mention. They are collections of parse trees or more generally syntactic structures of sentences. The production of a treebank generally requires a team of linguists to parenthesize the constituents of a corpus or to arrange them in a dependency structure. Annotated corpora require a fair amount of handwork and are therefore more expensive than raw texts. Treebanks involve even more clerical work and are relatively rare. The

Penn Treebank (Marcus et al. 1993) from the University of Pennsylvania is a widely cited example for English.

A last word on annotated corpora: in tests, we will benchmark automatic methods against manual annotation, which is often called the gold standard. We will assume the hand annotation perfect, although this is not true in practice. Some errors slip into hand-annotated corpora, even in those of the best quality, and the annotators may not agree between them. The scope of agreement varies depending on the annotation task. The inter-annotator agreement is generally high for parts of speech that are relatively well defined. It is lower when determining the sense of a word, for which annotators may have different interpretations. This inter-annotator agreement defines then a sort of upper bound of the human performance. It is a useful figure to conduct a reasonable assessment of results obtained by automatic methods as well as their potential for improvements.

2.1.2 Corpora and Lexicon Building

Lexicons and dictionaries are intended to give word lists, to provide a reader with word senses and meanings, and to outline their usage. Dictionaries' main purpose is related to lexical semantics. Lexicography is the science of building lexicons and writing dictionaries. It uses electronic corpora extensively.

The basic data of a dictionary is a word list. Such lists can be drawn manually or automatically from corpora. Then, lexicographers write the word definitions and choose citations illustrating the words. Since most of the time, current meanings are obvious to the reader, meticulous lexicographers tended to collect examples – citations – reflecting a rare usage. Computerized corpora can help lexicographers avoid this pitfall by extracting all the citations that exemplify a word. An experienced lexicographer will then select the most representative examples that reflect the language with more relevance. S/he will prefer and describe more frequent usage and possibly set aside others.

Finding a citation involves sampling a fragment of text surrounding a given word. In addition, the context of a word can be more precisely measured by finding recurrent pairs of words, or most-frequent neighbors. The first process results in concordance tables, and the second one in collocations.

A **concordance** is an alphabetical index of all the words in a text, or the most significant ones, where each word is related to a comprehensive list of passages where the word is present. Passages may start with the word or be centered on it and surrounded by a limited number of words before and after it (Table 2.2 and incipit of this chapter). Furthermore, concordances feature a system of reference to connect each passage to the book, chapter, page, paragraph, or verse, where it occurs.

Concordance tables were first produced for antiquity and religious studies. Hugh of St-Cher is known to have directed the first concordance to the scriptures in

Table 2.2 Concordance of *miracle* in the Gospel of John. English text: King James version; French text: Augustin Crampon; German text: Luther's Bible

Language	Concordances
English	l now. This beginning of miracles did Jesus in Cana of Ga
	name, when they saw the miracles which he did. But Jesus
	for no man can do these miracles that thou doest, except
	This is again the second miracle that Jesus did, when he
	im, because they saw his miracles which he did on them th
French	Galilée, le premier des miracles que fit Jésus, et il ma
	que, beaucoup voyant les miracles qu'il faisait, crurent
	nne ne saurait faire les miracles que vous faites, si Die
	maison. Ce fut le second miracle que fit Jésus en revenan
	parce qu'elle voyait les miracles qu'il opérait sur ceux
German	alten. Das ist das erste Zeichen, das Jesus tat, geschehe
	as zeigst du uns für ein Zeichen, daß du dies tun darfst?
	seinen Namen, da sie die Zeichen sahen, die er tat. Aber
	n; denn niemand kann die Zeichen tun, die du tust, es sei
	h zu ihm: Wenn ihr nicht Zeichen und Wunder seht, so glau

Table 2.3 Comparing *strong* and *powerful*. The German words *eng* and *schmal* 'narrow' are near-synonyms, but have different collocates

	English	French	German
You say	*Strong tea*	*Thé fort*	*Schmales Gesicht*
	Powerful computer	*Ordinateur puissant*	*Enge Kleidung*
You don't say	*Strong computer*	*Thé puissant*	*Schmale Kleidung*
	Powerful tea	*Ordinateur fort*	*Enges Gesicht*

the thirteenth century. It comprised about 11,800 words ranging from *A, a, a.* to *Zorobabel* and 130,000 references (Rouse and Rouse 1974). Other more elaborate concordances take word morphology into account or group words together into semantic themes. Sœur Jeanne d'Arc (1970) produced an example of such a concordance for Bible studies.

Concordancing is a powerful tool to study usage patterns and to write definitions. It also provides evidence on certain preferences between verbs and prepositions, adjectives and nouns, recurring expressions, or common syntactic forms. These couples are referred to as **collocations**. Church and Mercer (1993) cite a striking example of idiosyncratic collocations of *strong* and *powerful*. While *strong* and *powerful* have similar definitions, they occur in different contexts, as shown in Table 2.3.

Table 2.4 shows additional collocations of *strong* and *powerful*. These word preferences cannot be explained using rational definitions, but can be observed in corpora. A variety of statistical tests can measure the strength of pairs, and we can extract them automatically from a corpus.

Table 2.4 Word preferences of *strong* and *powerful* collected from the Associated Press corpus. Numbers in columns indicate the number of collocation occurrences with word *w* (After Church and Mercer (1993))

Preference for *strong* over *powerful*			**Preference for *powerful* over *strong***		
strong w	*powerful w*	*w*	*strong w*	*powerful w*	*w*
161	0	*showing*	1	32	*than*
175	2	*support*	1	32	*figure*
106	0	*defense*	3	31	*minority*
...					

2.1.3 Corpora as Knowledge Sources for the Linguist

In the early 1990s, computer-based corpus analysis completely renewed empirical methods in linguistics. It helped design and implement many of the techniques presented in this book. As we saw with dictionaries, corpus analysis helps lexicographers acquire lexical knowledge and describe language usage. More generally, corpora enable us to experiment with tools and to confront theories and models on real data. For most language analysis programs, collecting relevant corpora of texts is then a necessary step to define specifications and measure performances. Let us take the examples of part-of-speech taggers, parsers, and dialogue systems.

Annotated corpora are essential tools to develop part-of-speech taggers or parsers. A first purpose is to measure the tagging or parsing performance. The tagger or parser is run on texts and their result is compared to hand annotation, which serves as a reference. A linguist or an engineer can then determine the accuracy, the robustness of an algorithm or a parsing model and see how well it scales up by applying it to a variety of texts.

A second purpose of annotated corpora is to be a knowledge source to refine tagging techniques and improve grammars. While developing a grammar, a linguist can see if changing a rule improves or deteriorates results. The tool tuning is then done manually. Using statistical or machine-learning techniques, annotated corpora also enable researchers to create models, and identify parameters automatically or semiautomatically to tag or parse a text. We will see this in Chap. 8.

A dialogue corpus between a user and a machine is also critical to develop an interactive spoken system. The corpus is usually collected through fake dialogues between a real user and a person simulating the machine answers. Repeating such experiments with a reasonable number of users enables us to acquire a text set covering what the machine can expect from potential users. It is then easier to determine the vocabulary of an application, to have a precise idea of word frequencies, and to know the average length of sentences. In addition, the dialogue corpus enables the analyst to understand what the user expects from the machine and how s/he interacts with it.

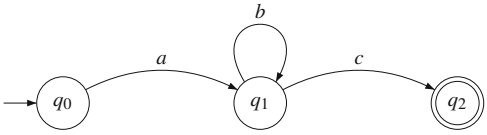


Fig. 2.1 A finite-state automaton

2.2 Finite-State Automata

2.2.1 A Description

The most frequent operation we do with corpora consists in searching for words or phrases. To be convenient, search must extend beyond fixed strings. We may want to search for a word or its plural form, strings consisting of uppercase or lowercase letters, expressions containing numbers, etc. This is made possible using finite-state automata (FSA), which we introduce now. FSA are flexible tools to process texts and are one of the most adequate ways to search strings.

FSA theory was designed in the beginning of computer science as a model of abstract computing machines. It forms a well-defined formalism that has been tested and used by generations of programmers. FSA stem from a simple idea. These are devices that accept – recognize – or reject an input stream of characters. FSA are very efficient in terms of speed and memory occupation and are easy to implement in Prolog. In addition to text searching, they have many other applications: morphological parsing, part-of-speech annotation, and speech processing.

Figure 2.1 shows an automaton with three states numbered from 0 to 2, where state q_0 is called the start state, and q_2, the final state. An automaton has a single start state and any number of final states, indicated by double circles. Arcs between states designate the possible transitions. Each arc is annotated by a label, which means that the transition accepts or generates the corresponding character.

An automaton accepts an input string in the following way: it starts in the initial state, follows a transition where the arc character matches the first character of the string, consumes the corresponding string character, and reaches the destination state. It then makes a second transition with the second character of the string, and continues in this way until it ends up in one of the final states and there is no character left. The automaton in Fig. 2.1 accepts or generates strings such as: ac, abc, $abbc$, $abbbc$, $abbbbbbbbbbbbc$, etc. If the automaton fails to reach a final state, either because it has no more characters in the input string or because it is trapped in a nonfinal state, it rejects the string.

As an example, let us see how the automaton accepts string $abbc$ and rejects $abbcb$. The input $abbc$ is presented to the start state q_0. The first character of the string matches that of the outgoing arc. The automaton consumes character a and moves to state q_1. The remaining string is bbc. Then, the automaton loops twice on state q_1 and consumes bb. The resulting string is character c. Finally, the automaton consumes c and reaches state q_2, which is the final state. On the contrary,

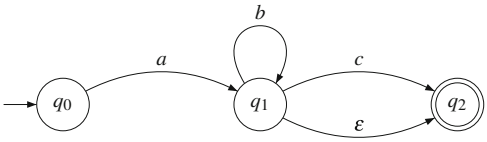


Fig. 2.2 A finite-state automaton with an ε-transition

Table 2.5 A state-transition table where Ø denotes nonexistent or impossible transitions

State\Input	a	b	c
q_0	q_1	Ø	Ø
q_1	Ø	q_1	q_2
q_2	Ø	Ø	Ø

the automaton does not accept string $abbcb$. It moves to states q_0, q_1, and q_2, and consumes $abbc$. The remaining string is letter b. Since there is no outgoing arc with a matching symbol, the automaton is stuck in state q_2 and rejects the string.

Automata may contain ε-transitions from one state to another. In this case, the automaton makes a transition without consuming any character of the input string. The automaton in Fig. 2.2 accepts strings a, ab, abb, etc., as well as ac, abc, $abbc$, etc.

2.2.2 Mathematical Definition of Finite-State Automata

FSA have a formal definition. An FSA consists of five components $(Q, \Sigma, q_0, F, \delta)$, where:

1. Q is a finite set of states.
2. Σ is a finite set of symbols or characters: the input alphabet.
3. q_0 is the start state, $q_0 \in Q$.
4. F is the set of final states, $F \subseteq Q$.
5. δ is the transition function $Q \times \Sigma \rightarrow Q$, where $\delta(q, i)$ returns the state where the automaton moves when it is in state q and consumes the input symbol i.

The quintuple defining the automaton in Fig. 2.1 is $Q = \{q_0, q_1, q_2\}$, $\Sigma = \{a, b, c\}$, $F = \{q_2\}$, and $\delta = \{\delta(q_0, a) = q_1, \delta(q_1, b) = q_1, \delta(q_1, c) = q_2\}$. The state-transition table in Table 2.5 is an alternate representation of the δ function.

2.2.3 Finite-State Automata in Prolog

A finite-state automaton has a straightforward implementation in Prolog. It is merely the transcription of the quintuplet definition. The following code describes the transitions, the start, and the final states of the automaton in Fig. 2.1:

```
% The start state
start(q0).

% The final states
final(q2).

% The transitions
% transition(SourceState, Symbol, DestinationState)
transition(q0, a, q1).
transition(q1, b, q1).
transition(q1, c, q2).
```

The predicate `accept/1` selects the start state and runs the automaton using `accept/2`. The predicate `accept/2` is recursive. It succeeds when it reaches a final state, or consumes a symbol of the input string and makes a transition otherwise.

```
accept(Symbols) :-
  start(StartState),
  accept(Symbols, StartState).

% accept(+Symbols, +State)
accept([], State) :-
  final(State).
accept([Symbol | Symbols], State) :-
  transition(State, Symbol, NextState),
  accept(Symbols, NextState).
```

`accept/1` either accepts an input symbol string or fails:

```
?- accept([a, b, b, c]).
true

?- accept([a, b, b, c, b]).
false
```

The automaton in Fig. 2.2 contains ε-transitions. They are introduced in the database as facts:

```
epsilon(q1, q2).
```

To take them into account, the `accept/2` predicate should be modified so that there are two possible sorts of transitions. A first rule consumes a character and a second one, corresponding to an ε-transition, passes the string unchanged to the next state:

```
accept([], State) :-
  final(State).
accept([Symbol | Symbols], State) :-
  transition(State, Symbol, NextState),
  accept(Symbols, NextState).
```

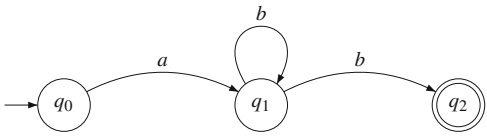


Fig. 2.3 A nondeterministic automaton

```
accept(Symbols, State) :-
  epsilon(State, NextState),
  accept(Symbols, NextState).
```

2.2.4 Deterministic and Nondeterministic Automata

The automaton in Fig. 2.1 is said to be deterministic (DFSA) because given a state and an input, there is one single possible destination state. On the contrary, a nondeterministic automaton (NFSA) has states where it has a choice: the path is not determined in advance.

Figure 2.3 shows an example of an NFSA that accepts the strings ab, abb, $abbb$, $abbbb$, etc. Taking abb as input, the automaton reaches the state q_1 consuming the letter a. Then, it has a choice between two states. The automaton can either move to state q_2 or stay in state q_1. If it first moves to state q_2, there will be one character left, and the automaton will fail. The right path is to loop onto q_1 and then to move to q_2. ε-transitions also cause automata to be nondeterministic as in Fig. 2.2, where any string that has reached state q_1 can also reach state q_2.

A possible strategy to deal with nondeterminism is to use backtracking. When an automaton has the choice between two or more states, it selects one of them and remembers the state where it made the decision: the choice point. If it subsequently fails, the automaton backtracks to the choice point and selects another state to go to. In our example in Fig. 2.3, if the automaton moves first to state q_2 with the string bb, it will end up in a state without outgoing transition. It will have to backtrack and select state q_1. Backtracking is precisely the strategy that Prolog uses automatically.

2.2.5 Building a Deterministic Automaton from a Nondeterministic One

Although surprising, it is possible to convert any nondeterministic automaton into an equivalent deterministic automaton. We outline here an informal description of the determinization algorithm. See Hopcroft et al. (2007) for a complete description of this algorithm.

The algorithm starts from an NFSA $(Q_N, \Sigma, q_0, F_N, \delta_N)$ and builds an equivalent DFSA $(Q_D, \Sigma, \{q_0\}, F_D, \delta_D)$, where:

Table 2.6 The state-transition table of the nondeterministic automaton shown in Fig. 2.3

State\Input	a	b
q_0	q_1	Ø
q_1	Ø	q_1, q_2
q_2	Ø	Ø

Table 2.7 The state-transition table of the determinized automaton in Fig. 2.3

State\Input	a	b
Ø	Ø	Ø
$\{q_0\}$	$\{q_1\}$	Ø
$\{q_1\}$	Ø	$\{q_1, q_2\}$
$\{q_2\}$	Ø	Ø
$\{q_0, q_1\}$	$\{q_1\}$	$\{q_1, q_2\}$
$\{q_0, q_2\}$	$\{q_1\}$	Ø
$\{q_1, q_2\}$	Ø	$\{q_1, q_2\}$
$\{q_0, q_1, q_2\}$	$\{q_1\}$	$\{q_1, q_2\}$

- Q_D is the set of all the possible state subsets of Q_N. It is called the power set. The set of states of the automaton in Fig. 2.3 is $Q_N = \{q_0, q_1, q_2\}$. The corresponding set of sets is $Q_D = \{\emptyset, \{q_0\}, \{q_1\}, \{q_2\}, \{q_0, q_1\}, \{q_0, q_2\}, \{q_1, q_2\}, \{q_0, q_1, q_2\}\}$. If Q_N has n states, Q_D will have 2^n states. In general, many of these states will be inaccessible and will be discarded.
- F_D is the set of sets that include at least one final state of Q_D. In our example, $F_D = \{\{q_2\}, \{q_0, q_2\}, \{q_1, q_2\}, \{q_0, q_1, q_2\}\}$.
- For each set $S \subset Q_N$ and for each input symbol a, $\delta_D(S, a) = \bigcup_{s \in S}^{\delta_N(s,a)}$. The state-transition table in Table 2.6 represents the automaton in Fig. 2.3. Table 2.7 represents the determinized version of it.

2.2.6 *Searching a String with a Finite-State Automaton*

Searching the occurrences of a string in a text corresponds to recognizing them with an automaton, where the string characters label the sequence of transitions. However, the automaton must skip chunks in the beginning, between the occurrences, and at the end of the text. The automaton consists then of a core accepting the searched string and of loops to process the remaining pieces. Consider again the automaton in Fig. 2.1 and modify it to search strings ac, abc, $abbc$, $abbbc$, etc., in a text. We add two loops: one in the beginning and the other to come back and start the search again (Fig. 2.4).

In doing this, we have built an NFSA that it is preferable to convert into a DFSA. Hopcroft et al. (2007) describe the mathematical properties of such automata and an algorithm to automatically build an automaton for a given set of patterns to search. They notably report that resulting DFSA have exactly the same number of

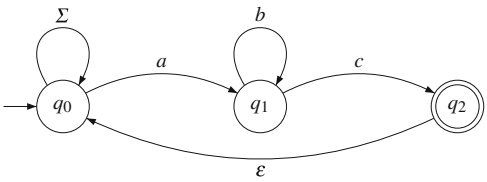


Fig. 2.4 Searching strings ac, abc, $abbc$, $abbbc$, etc.

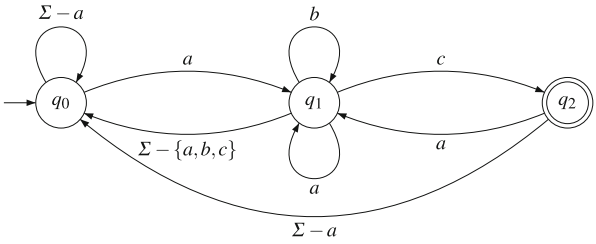


Fig. 2.5 An automaton to search strings ac, abc, $abbc$, $abbbc$, etc., in a text

states as the corresponding NFSA. We present an informal solution to determine the transitions of the automaton in Fig. 2.4.

If the input text does not begin with an a, the automaton must consume the beginning characters and loop on the start state until it finds one. Figure 2.5 expresses this with an outgoing transition from state 0 to state 1 labeled with an a and a loop for the rest of the characters. $\Sigma - a$ denotes the finite set of symbols except a. From state 1, the automaton proceeds if the text continues with either a b or a c. If it is an a, the preceding a is not the beginning of the string, but there is still a chance because it can start again. This corresponds to the second loop on state 1. Otherwise, if the next character falls in the set $\Sigma - \{a, b, c\}$, the automaton goes back to state 0. The automaton successfully recognizes the string if it reaches state 2. Then it goes back to state 0 and starts the search again, except if the next character is an a, for which it can go directly to state 1.

2.2.7 Operations on Finite-State Automata

FSA can be combined using a set of operations. The most useful are the union, the concatenation, and the closure.

The union or sum of two automata A and B accepts or generates all the strings of A and all the strings of B. It is denoted $A \cup B$. We obtain it by adding a new initial state that we link to the initial states of A and B (Fig. 2.6) using ε-transitions (Fig. 2.7).

The concatenation or product of A and B accepts all the strings that are concatenations of two strings, the first one being accepted by A and the second one by B. It is denoted $A.B$. We obtain the resulting automaton by connecting all the final states of A to the initial state of B using ε-transitions (Fig. 2.8).

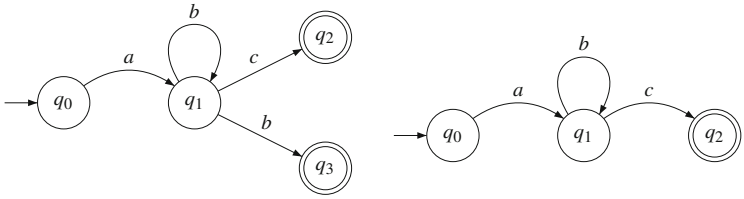


Fig. 2.6 Automata A (*left*) and B (*right*)

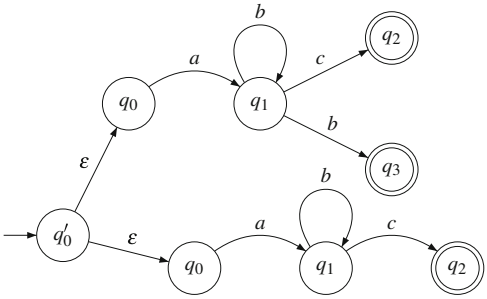


Fig. 2.7 The union of two automata: $A \cup B$

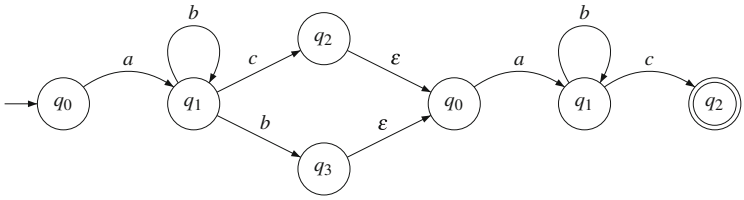


Fig. 2.8 The concatenation of two automata: $A.B$

The iteration or Kleene closure of an automaton A accepts the concatenations of any number of its strings and the empty string. It is denoted A^*, where $A^* = \{\varepsilon\} \cup A \cup A.A \cup A.A.A \cup A.A.A.A \cup \ldots$. We obtain the resulting automaton by linking the final states of A to its initial state using ε-transitions and adding a new initial state, as shown in Fig. 2.9. The new initial state enables us to obtain the empty string.

The notation Σ^* designates the infinite set of all possible strings generated from the alphabet Σ. Other significant operations are:

- The intersection of two automata $A \cap B$ that accepts all the strings accepted both by A and by B. If $A = (\Sigma, Q_1, q_1, F_1, \delta_1)$ and $B = (\Sigma, Q_2, q_2, F_2, \delta_2)$, the resulting automaton is obtained from the Cartesian product of states $(\Sigma, Q_1 \times Q_2, \langle q_1, q_2 \rangle, F_1 \times F_2, \delta_3)$ with the transition function $\delta_3(\langle s_1, s_2 \rangle, i) = \{\langle t_1, t_2 \rangle \mid t_1 \in \delta_1(s_1, i) \wedge t_2 \in \delta_2(s_2, i)\}$.

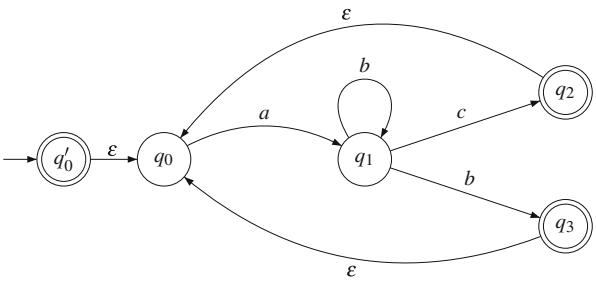


Fig. 2.9 The closure of A

- The difference of two automata $A - B$ that accepts all the strings accepted by A but not by B.
- The complementation of the automaton A in Σ^* that accepts all the strings that are not accepted by A. It is denoted $\bar{A}$, where $\bar{A} = \Sigma^* - A$.
- The reversal of the automaton A that accepts all the reversed strings accepted by A.

Two automata are said to be equivalent when they accept or generate exactly the same set of strings. Useful equivalence transformations optimize computation speed or memory requirements. They include:

- ε-removal, which transforms an initial automaton into an equivalent one without ε-transitions;
- Determinization, which transforms a nondeterministic automaton into a deterministic one;
- Minimization, which determines among equivalent automata the one that has the smallest number of states.

Optimization algorithms are outside the scope of this book. Hopcroft et al. (2007) as well as Roche and Schabes (1997) describe them in detail.

2.3 Regular Expressions

The automaton in Fig. 2.1 generates or accepts strings composed of one a, zero or more b's, and one c. We can represent this set of strings using a compact notation: `ab*c`, where the star symbol means any number of the preceding character. Such a notation is called a regular expression or regex. Regular expressions are very powerful devices to describe patterns to search in a text. Although their notation is different, regular expressions can always be implemented in the form of automata, and vice versa. However, regular expressions are generally easier to use.

Table 2.8 Examples of simple patterns and matching results

Pattern	String
`regular`	"A section on regular expressions"
`Prolog`	"The Prolog language"
`the`	"The book of the life"

Regular expressions are composed of literal characters, that is, ordinary text characters, like `abc`, and of metacharacters, like `*`, that have a special meaning. The simplest form of regular expressions is a sequence of literal characters: letters, numbers, spaces, or punctuation signs. The regexes `regular` and `Prolog` match, respectively, the strings *regular* or *Prolog* contained in a text. Table 2.8 shows examples of pattern matching with literal characters. Regular expressions are case-sensitive and match the first instance of the string or all its instances in a text, depending on the regex language that is used.

There are currently a dozen major regular expression dialects freely available. Their common ancestor is `grep`, which stands for global/regular expression/print. `grep`, together with `egrep`, a modern version of it, is a standard Unix tool that prints out all the lines of a file that contain a given pattern. The `grep` user interface conforms to the Unix command-line style. It consists of the command name, here `grep`, options, and the arguments. The first argument is the regular expression delimited by single straight quotes. The next arguments are the files where to search the pattern:

```
grep 'regular expression' file1 file2 ... filen
```

The Unix command:

```
grep 'abc' myFile
```

prints all the lines of file `myFile` containing the string *abc* and

```
grep 'ab*c' myFile1 myFile2
```

prints all the lines of file `myFile1` and `myFile2` containing the strings *ac*, *abc*, *abbc*, *abbbc*, etc.

`grep` had a considerable influence, and most programming languages, including Perl, Python, Java, and C#, have now some support for regexes. All the regex variants – or flavors – adhere to an analog syntax, with some differences, however, that hinder a universal compatibility.

In the following sections, we will use the syntax defined by Perl. Because of its built-in support for regexes and its simplicity, Perl was immediately recognized as a real innovation in the world of scripting languages and was adopted by millions of programmers. It is probably Perl that made regular expressions a mainstream programming technique and, in return, it explains why the Perl regex syntax became a sort of *de facto* standard that inspires most modern regex flavors.

Table 2.9 Repetition metacharacters (quantifiers)

Metachar	Description	Example
`*`	Matches any number of occurrences of the previous character – zero or more	`ac*e` matches strings `ae`, `ace`,`acce`, `accce`, etc. as in "The aerial acceleration alerted the ace pilot"
`?`	Matches at most one occurrence of the previous character – zero or one	`ac?e` matches `ae` and `ace` as in "The aerial acceleration alerted the ace pilot"
`+`	Matches one or more occurrences of the previous character	`ac+e` matches `ace`, `acce`, `accce`, etc. as in as in "The aerial acceleration alerted the ace pilot"
`{n}`	Matches exactly *n* occurrences of the previous character	`ac{2}e` matches `acce` as in "The aerial acceleration alerted the ace pilot"
`{n,}`	Matches *n* or more occurrences of the previous character	`ac{2,}e` matches `acce`, `accce`, etc.
`{n,m}`	Matches from *n* to *m* occurrences of the previous character	`ac{2,4}e` matches `acce`, `accce`, and `acccce`.

2.3.1 Repetition Metacharacters

We saw that the metacharacter `*` expressed a repetition of zero or more characters, as in `ab*c`. Other characters that describe repetitions are the question mark, `?`, the plus, `+`, and the range quantifiers `{n,m}` matching a specified range of occurrences (Table 2.9). The star symbol is also called the closure operator or the Kleene star.

2.3.2 The Dot Metacharacter

The dot `.` is also a metacharacter that matches one occurrence of any character of the alphabet except a new line. For example, `a.e` matches the strings *ale* and *ace* in the sentence:

The aerial acceleration alerted the ace pilot

as well as *age*, *ape*, *are*, *ate*, *awe*, *axe*, or *aae*, *aAe*, *abe*, *aBe*, *a1e*, etc. We can combine the dot and the star in the expression `.*` to match any string of characters until we encounter a new line.

2.3.3 The Escape Character

If the pattern to search contains a character that is also a metacharacter, for instance, "?", we need to indicate it to the regex engine using a backslash `\` before it. We saw that `abc?` matches *ab* and *abc*. The expression `abc\?` matches the string *abc*?. In the same vein, `abc\.` matches the string *abc*., and `a\*bc` matches *a*bc*.

We call the backslash an escape character. It transforms a metacharacter into a literal symbol. We can also say that we "quote" a metacharacter with a backslash. In Perl, we must use a backslash escape with the 14 following characters:

```
* + ? . \ | ( ) [ ] { } ^ $
```

to search them literally.

As a matter of fact, the backslash is not always necessary as sometimes Perl can guess from the context that a character has a literal meaning. This is the case for the braces, for instance, that Perl interprets as literals outside the expressions, `{n}`, `{n,}`, and `{n,m}`. Anyway, it is always safer to use a backslash escape to avoid ambiguities.

2.3.4 The Longest Match

The description of repetition metacharacters in Table 2.9 sometimes makes string matching ambiguous, as with the string `aabbc` and the regex `a+b*`, which could have six possible matches: *a*, *aa*, *ab*, *aab*, *abb*, and *aabb*. In fact, matching algorithms use two rules that are common to all the regex languages:

1. They match as early as they can in a string.
2. They match as many characters as they can.

Hence, `a+b*` matches *aabb*, which is the longest possible match. The matching strategy of repetition metacharacters is said to be greedy.

In some cases, the greedy strategy is not appropriate. To display the sentence

> They match **as early** and **as many** characters as they can.

in a web page with two phrases set in bold, we need specific tags that we will insert in the source file. Using HTML, the language of the web, the sentence will probably be annotated as

```
They match <b>as early</b> and <b>as many</b> characters as
they can.
```

where `<b>` and `</b>` mark respectively the beginning and the end of a phrase set in bold. (We will see annotation frameworks in more detail in Chap. 3.)

A regular expression to search and extract phrases in bold could be:

```
<b>.*</b>
```

Unfortunately, applying this regex to the sentence will match one single string:

```
<b>as early</b> and <b>as many</b>
```

which is not what we wanted. In fact, this is not a surprise. As we saw, the regex engine matches as early as it can, i.e., from the first `<b>` and as many characters as it can up to the second `</b>`.

Table 2.10 Lazy metacharacters

Metachar	Description
`*?`	Matches any number of occurrences of the previous character – zero or more
`??`	Matches at most one occurrence of the previous character – zero or one
`+?`	Matches one or more occurrences of the previous character
`{n}?`	Matches exactly *n* occurrences of the previous character
`{n,}?`	Matches *n* or more occurrences of the previous character
`{n,m}?`	Matches from *n* to *m* occurrences of the previous character

A possible solution is to modify the behavior of repetition metacharacters and make them "lazy." They will then consume as few characters as possible. We create the lazy variant of a repetition metacharacter by appending a question mark to it (Table 2.10). The regex

```
<b>.*?</b>
```

will then match the two intended strings,

`<b>as early</b>` and `<b>as many</b>`.

2.3.5 Character Classes

We saw that the dot, `.`, represented any character of the alphabet. It is possible to define smaller subsets or **classes**. A list of characters between square brackets `[...]` matches any character contained in the list. The expression `[abc]` means one occurrence of either `a`, `b`, or `c`; `[ABCDEFGHIJKLMNOPQRSTUVWXYZ]` means one uppercase unaccented letter; and `[0123456789]` means one digit. We can concatenate character classes, literal characters, and metacharacters, as in the expressions `[0123456789]+` and `[0123456789]+\.[0123456789]+`, that match, respectively, integers and decimal numbers.

Character classes are useful to search patterns with spelling differences, such as `[Cc]omputer [Ss]cience`, which matches four different strings:

Computer Science
Computer science
computer Science
computer science

Negated Character Classes

We can define the complement of a character class, that is, the characters of the alphabet that are not member of the class, using the caret symbol, `^`, as the first symbol inside the square brackets. For example:

- The expression `[^a]` means any character that is not an *a*;
- `[^0123456789]` means any character that is not a digit;
- `[^ABCD]+` means any string that does not contain *A*, *B*, *C*, or *D*.

Such classes are also called negated character classes.

Range of Characters

Inside square brackets, we can also specify ranges using the hyphen character: `-`. For example:

- The expression `[1-4]` means any of the digits *1*, *2*, *3*, or *4*, and `a[1-4]b` matches *a1b*, *a2b*, *a3c*, and *a4b*.
- The expression `[a-zàâäæçéèêëîïôöœßùûüÿ]` matches any lowercase accented or unaccented letter of French and German.

Metacharacters

Inside a character class, the hyphen is a metacharacter describing a range. If we want to search it like an ordinary character and include it in a class, we need to quote it with a backslash like this: `\-`. The expression `[1\-4]` means any of the characters *1*, -, or *4*.

In addition to the hyphen, the other metacharacters used in character classes are: the closing square bracket, `]`, the backslash, `\`, the caret, `^`, and the dollar sign, `$`. As for carets, they need to be quoted to be treated as normal characters in a character class. However, when they are in an unambiguous position, Perl will interpret them correctly even without the escape sign. For instance, if the caret is not the first character after the opening bracket, Perl will recognize it as a normal character. The expression `[a^b]` matches either *a*, ^, or *b*.

Predefined Character Classes

Most regex flavors have predefined classes. Table 2.11 lists some useful ones in Perl. Some classes are adopted by all the flavors, while some others are specific to Perl. In case of doubt, refer to the appropriate documentation. Perl also defines classes as properties using the `\p{class}` construct that matches the symbols in `class` and `\P{class}` that matches symbols not in `class`. To name the properties or classes, Perl uses its own categories as well as those defined by the Unicode standard that we will review in Chap. 3. This enables the programmer to handle non-Latin scripts more easily.

Table 2.11 Predefined character classes in Perl (After Wall et al. (2000))

Expression	Description	Equivalent	\p{...} equiv.
\d	Any digit	[0-9]	\p{IsDigit}
\D	Any nondigit	[^0-9]	\P{IsDigit}
\s	Any whitespace character: space, tabulation, new line, carriage return, or form feed	[\t\n\r\f]	\p{IsSpace}
\S	Any nonwhitespace character	[^\s]	\P{IsSpace}
\w	Any word character: letter, digit, or underscore	[a-zA-Z0-9_]	\p{IsWord}
\W	Any nonword character	[^\w]	\P{IsWord}
\p{IsAlpha}	Any alphabetic character. It includes accented characters		
\p{IsAlnum}	Any alphanumeric character. It includes accented characters	[\p{IsAlpha} \p{IsDigit}]	
\p{IsPunct}	Any punctuation sign		
\p{IsLower}	Any lowercase character. It includes accented characters		
\p{IsUpper}	Any uppercase character. It includes accented characters		

2.3.6 Nonprintable Symbols or Positions

Some metacharacters match positions and nonprintable symbols. Positions or **anchors** enable one to search a pattern with a specific location in a text. They encode the start and end of a line using, respectively, the caret, ^, and the dollar symbol, $.

The expression ^Chapter matches lines beginning with *Chapter* and [0-9]+$ matches lines ending with a number. We can combine both in ^Chapter [0-9]+$, which matches lines consisting only of the *Chapter* word and a number as *Chapter 3*, for example.

The command line

```
egrep '^[aeiou]+$' myFile
```

matches the lines of myFile containing only vowels.

Similarly, in Perl, the anchor \b matches word boundaries. The expression \bace matches *aces* and *acetylene* but not *place*. Conversely, ace\b matches *place* but neither *aces* nor *acetylene*. The expression \bact\b matches exactly the word *act* and not *react* or *acted*. Table 2.12 summarizes anchors and some nonprintable characters.

2.3.7 Union and Boolean Operators

We reviewed the basic constructs to write regular expressions. A powerful feature is that we can also combine expressions with operators, as with automata. Using a

Table 2.12 Some metacharacters matching nonprintable characters in Perl

Metachar	Description	Example
^	Matches the start of a line	^ab*c matches ac, abc, abbc, abbbc, etc., when they are located at the beginning of a new line
$	Matches the end of a line	ab?c$ matches ac and abc when they are located at the end of a line
\b	Matches word boundaries	\babc matches abcd but not dabc bcd\b matches abcd but not abcde

mathematical term, we say that they define an algebra. Using a simpler analogy, this means that we can arrange regular expressions just like arithmetic expressions. This means, for instance, that it will be possible to apply the repetition metacharacters * or + not only to the previous character, but to a previous regular expression. This greatly eases the design of complex expressions and makes them very versatile.

Regex languages use three main operators. Two of them are already familiar to us. The first one is the Kleene star or closure, denoted *. The second one is the concatenation, which is usually not represented. It is implicit in strings like abc, which is the concatenation of characters *a*, *b*, and *c*. To concatenate the word *computer*, a space symbol, and *science*, we just write them in a row: computer science.

The third operation is the union and is denoted "|". The expression a|b means either *a* or *b*. We saw that the regular expression [Cc]omputer [Ss]cience could match four strings. We can rewrite an equivalent expression using the union operator: Computer Science|Computer science|computer Science|computerscience. A union is also called an alternation because the corresponding expression can match any of the alternatives, here four.

2.3.8 Operator Combination and Precedence

Regular expressions and operators are grouped using parentheses. If we omit them, expressions are governed by rules of precedence and associativity. The expression a|bc matches the strings *a* and *bc* because the concatenation operator takes precedence over the union. In other words, the concatenation binds the characters stronger than the union. If we want an expression that matches the strings *ac* and *bc*, we need parentheses (a|b)c.

Let us examine another example of precedence. We rewrote the expression [Cc]omputer [Ss]cience using a union of four strings. Since the difference between expressions lies in the first letters only, we can try to revise this union into something more compact. The character class [Cc] is equivalent to the alternation C|c, which matches either *C* or *c*. A tentative expression could then be C|computer S|science. But it would not match the

desired strings; it would find occurrences of either *C*, *computer S*, or *science* because of the operator precedence. We need parentheses to group the alternations `(C|c)omputer (S|s)cience` and thus match the four intended strings.

The order of precedence of the three main operators union, concatenation, and closure is as follows:

1. Closure and other repetition operator (highest);
2. Concatenation, line and word boundaries;
3. Union (lowest).

This entails that `abc*` describes the set ab, abc, $abcc$, $abccc$, etc. To repeat the pattern abc, we need parentheses; and the expression `(abc)*` corresponds to abc, $abcabc$, $abcabcabc$, etc.

2.4 Programming with Regular Expressions

We saw that regular expressions were devices to define and search patterns in texts. If we want to use them for more elaborate text processing such as translating characters, substituting words, or counting them, we need a full-fledged programming language, for example, Perl, Python, C#, or Java with its `java.util.regex` package. They enable the design of powerful regexes and at the same time, they are complete programming languages.

This section, as well as the next chapter, discusses features of Perl. This intends to give you a glimpse of Perl programming. Further references include Christiansen et al. (2012) and Schwartz et al. (2011).

2.4.1 Perl

Perl has constructs similar to those of the C language. It has analogous control flow statements and the assignment operator is denoted =. However, variables begin with an initial symbol, which is the `$` character for an individual number or string. Such variables are called scalars in Perl and are not typed. Comments start with the # symbol. The short program

```
# A first program
$integer = 30;
$pattern = "My string";
print $integer, " ", $pattern, "\n";
```

prints the line

```
30 My string
```

We run it with the command:

```
perl -w program.pl
```

where the option `-w` asks Perl to check syntax errors.

2.4.2 Strings and Regular Expressions in Perl

Perl represents strings as sequences of characters or symbols enclosed within single or double quotes as, respectively, `'my string'` and `"my string"`. There is no limit to the length of strings; we can use them to store a whole corpus, provided that our machine has enough memory.

Single-Quoted Strings

Strings delimited by single quotes are interpreted literally by Perl, except the single quotes themselves and backslashes. To create strings containing these two characters, Perl defines two *escape sequences*: `\'` to represent a single quote and `\\` to represent a backslash as in:

```
$pattern = 'Perl\'s strings';
```

This instruction assigns the string *Perl's strings* to `$pattern`; the backslash escape character tells Perl to read the quote literally instead of interpreting it as an end-of-string delimiter.

The sequences consisting of a backslash and any other character, like `\n` or `\t`, are not escape sequences in single-quoted strings. They are literal parts of a string.

Double-Quoted Strings

As opposed to single quotes, double quotes tell Perl to interpolate the string when it contains variables or certain backslashed sequences. For example, like in Java or C, `\n` is interpreted as a new line and `\t` as a tabulation. Table 2.13 shows a list of escape sequences that have an altered meaning in double-quoted strings.

The right column in Table 2.13 lists the numerical representations of characters using the ASCII and Unicode standards. The `\N{name}` and `\x{hexcode}` sequences enable us to designate any character, like *Ö* and *Œ*, by its Unicode name, respectively, `\N{LATIN CAPITAL LETTER O WITH DIAERESIS}` and `\N{LATIN CAPITAL LIGATURE OE}`, or its code point, `\x{00D6}` and `\x{0152}`. We will review both the ASCII and Unicode schemes in Chap. 3.

Table 2.13 Escape sequences in double quoted strings

Sequence	Description	Sequence	Description
\t	Tabulation	\100	Octal ASCII, three digits, here @
\n	New line	\x40	Hexadecimal ASCII, two digits, here @
\r	Carriage return	\x{0152}	Unicode code point, here Œ
\f	Form feed	\N{COMMERCIAL AT}	Unicode name, here @
\e	Escape		
\b	Backspace		
\a	Bell		

To use Unicode escape sequences, we must include this directive at the beginning of the Perl program:

```
use charnames ':full';
```

Perl also interpolates variables inside double quotes as in

```
$begin = "my";
$pattern = "${begin} string";
```

where it replaces `${begin}` with *my* and assigns `$pattern` with *my string*. We surround the `$begin` variable with braces to tell Perl what the variable name exactly is and avoid ambiguities. Braces are not always necessary, but it is always safer to use them.

As \ and $ are special characters inside double quotes, as well as @, as we will see in Sect. 2.5.1, we need to use the escape sequences \\, \$, and \@ to insert these signs literally in strings.

Regular Expressions and Strings

Regular expressions and double-quoted strings are very similar constructs in Perl. We already examined the syntax of regex literals and their metacharacters in Sect. 2.3. In addition, as in double-quoted strings, regexes can use the escape sequences defined in Table 2.13 to match nonprintable or numerically-encoded characters as well as interpolate variables.

For example, Perl replaces the variables

```
$pattern = "my string";
$width = 20;
```

with their values in the regex literal

```
(.{0,$width}$pattern.{0,$width})
```

to produce

```
(.{0,20}my string.{0,20})
```

that matches the pattern *my string* with 0 to 20 characters to the left and to the right.

From Tables 2.12 and 2.13, you may have noted that the metacharacter \b was used with two different meanings: word boundary or backspace. In fact, its interpretation depends on the context: it is a backspace in character classes; otherwise, it matches word boundaries.

2.4.3 Matching

Perl's regex engine is supported by the language itself, and the matching operation has a dedicated construct to denote a regular expression: m/regex/. The next program reads the input line and searches the expression ab*c. If it finds the expression, it prints the line:

```
while ($line = <>) {
  if ($line =~ m/ab*c/) {
    print $line;
  }
}
```

The program uses repeat and conditional statements. The symbol <> designates the standard input, and the instruction $line = <> assigns the current line from the input to the $line variable. The while instruction reads all the lines until it encounters an end of file. The m/.../ instruction delimits the regular expression to match, and the =~ operator instructs Perl to search it in the $line variable. If the expression matches a string in $line, the =~ operator returns true, or false otherwise. The if instruction tells the program to print the input when it contains the pattern. We run the program to search the file file_name with the command:

```
perl -w program.pl file_name
```

The match operator supports a set of options also called modifiers. Their syntax is m/regex/modifiers. Useful modifiers are:

- Case insensitive: i. The instruction m/regex/i searches regex in the target string regardless of its case.
- Multiple lines: m. By default, the anchors ^ and $ match the start and the end of the input string. The instruction m/regex/m considers the input string as multiple lines separated by new line characters, where the anchors ^ and $ match the start and the end of any line in the string.
- Single line: s. Normally, a dot symbol "." does not match new line characters. The /s modifier makes a dot in the instruction m/regex/s match any character, including new lines.

Modifiers can be grouped in any order as in m/regex/im, for instance, or m/regex/sm, where a dot in regex matches any character and the anchors ^ and $ match just after and before new line characters.

2.4.4 Substitutions

One of the powerful features of Perl is pattern substitution. It uses a construct similar to the match instruction: `s/regex/replacement/`. The instruction:

```
$line =~ s/regex/replacement/
```

matches the first occurrence of `regex` and replaces it by the `replacement` string in the `$line` variable. If we want to replace all the occurrences of a pattern, we use the `g` modifier, where `g` stands for globally:

```
$line =~ s/regex/replacement/g
```

We shall write a program to replace the occurrences of `ab*c` by `ABC` in a file and print them. We read all the lines of the input. We use the instruction `m/ab*c/` to check whether they match the regular expression `ab*c`. We then print the old line and we substitute the matched pattern using the construct `s/ab*c/ABC/`:

```
while ($line = <>) {
  if ($line =~ m/ab*c/) {
    print "Old: ", $line;
    $line =~ s/ab*c/ABC/g;
    print "New: ", $line;
  }
}
```

2.4.5 Translating Characters

The transliteration instruction `tr/search_list/replacement_list/` replaces all the occurrences of the characters in `search_list` by the corresponding character in
`replacement_list`. The instruction `tr/ABC/abc/` replaces the occurrences of *A*, *B*, and *C* by *a*, *b*, and *c*, respectively. The string

```
AbCdEfGhIjKlMnOpQrStUvWxYzÉö
```

results in

```
abcdEfGhIjKlMnOpQrStUvWxYzÉö
```

The hyphen specifies a character range, as in the instruction

```
$line =~ tr/A-Z/a-z/;
```

which converts the uppercase characters to their lowercase equivalents. The instruction `tr` has useful modifiers:

- `d` deletes any characters of the search list that are not found in the replacement list.
- `c` translates characters that belong to the complement of the search list.
- `s` reduces – squeezes, squashes – sequences of characters translated to an identical character to a single instance.

The instruction

```
$line =~ tr/AEIOUaeiou//d;
```

deletes all the vowels in `$line` and

```
$line =~ tr/AEIOUaeiou/$/cs;
```

replaces all nonvowel characters by a $ sign. The contiguous sequences of translated dollar signs are reduced to a single sign.

2.4.6 String Operators

Perl operators are similar to those of the C and Java languages. They are summarized in Table 2.14. The string operators are notable differences. They enable us to concatenate and compare strings.

The Boolean operators `eq` (equal) and `ne` (not equal) compare two strings. The dot is the concatenation operator:

```
$string1 = "abc";
$string 2 = "def";
$string3 = $string1 . $string2;
print $string3;
#prints abcdef
```

As with the C and Java operators, the shorthand notation `$var1 .= $var2` is equivalent to `$var1 = $var1 . $var2`. The following program reads the content of the input line by line, concatenates it in the `$text` variable, and prints it:

```
while ($line = <>) {
  $text .= $line;
}
print $text;
```

2.4.7 Back References

It is sometimes useful to keep a reference to matched patterns or parts of them. Let us imagine that we want to find a sequence of three identical characters, which corresponds to matching a character and checking if the next two characters are

Table 2.14 Summary of the main Perl operators

Unary operators		
	`!`	Logical not
	`+` and `-`	Arithmetic plus sign and negation
Binding operators		
	`=~`	Returns true in case of match success
	`!~`	Returns false in case of match success
Arithmetic operators		
	`*` and `/`	Multiplication and division
	`+` and `-`	Addition and subtraction
String operator		
	`.`	String concatenation
Arithmetic comparison operators		
	`>` and `<`	Greater than and less than
	`>=` and `<=`	Greater than or equal and less than or equal
	`==` and `!=`	Equal and not equal
String comparison operators		
	`ge` and `le`	Greater than or equal and less than or equal
	`gt` and `lt`	Greater than and less than
	`eq` and `ne`	Equal and not equal
Logical operators		
	`&&`	Logical and
	`||`	Logical or

identical to the first character. To do this, we first tell Perl to remember the matched pattern and we put parentheses around it. This creates a buffer to hold the pattern and we refer back to it by the sequence `\1`. The instruction

```
$line =~ s/(.)\1\1/***/g;
```

matches sequences of three identical characters and replaces them by three stars in `$line`.

Perl can create as many buffers as we need. It allocates a new one when it encounters a left parenthesis and refers back to it by references `\1`, `\2`, `\3`, etc. The first pair of parentheses corresponds to `\1`, the second pair to `\2`, the third to `\3`, etc.

Outside the regular expression, the `\<digit>` reference is denoted by `$<digit>`: `$1`, `$2`, `$3`, etc. As an example, the next instruction matches the decimal amounts of money expressed with the dollar sign and substitutes them with the words *dollars* and *cents* in clear in the replacement string:

```
$line =~ s/\$ *([0-9]+)\.?([0-9]*)/$1 dollars and $2 cents/g;
```

Perl will keep these references until the next pattern matching instruction. The program below uses them in a separate instruction and prints the dollars and cents of money amount occurring in `$line`:

```
while ($line = <>) {
  while ($line =~ m/\$ *([0-9]+)\.?([0-9]*)/g) {
    print "Dollars: ", $1, " Cents: ", $2, "\n";
  }
}
```

In the inner loop, the combination of `while` and the `/g` modifier enables the `m/.../` instruction to match a pattern and to start a new search from its current position – where the previous match ended. When `m/.../g` fails to match, the start position is reset to the beginning of the string and we exit the loop. Without this `/g` modifier, we would have looped onto the first occurrence of the pattern and printed it infinitely.

2.4.8 Predefined Variables

Perl has a large set of predefined variables, which are assigned without the need for us to do it explicitly. The most frequently used is default input variable: `$_`. When writing an input instruction in a program, we can omit the variable that would store the input as well as the = operator. Perl will automatically assign it to `$_`. In the same way, you can also leave out the left value of the pattern matching operations `m//`, `s///`, and `tr///`, as well as their =~ operator or the variable of some one-place functions (unary functions). Perl will use `$_` without us having to specify it in the program:

```
while (<>) {
  if (m/ab*c/) {
    print;
  }
}
```

which is equivalent to:

```
while ($_ = <>) {
  if ($_ =~ m/ab*c/) {
    print $_;
  }
}
```

The triple `$‘`, `$&`, and `$’` is another set of useful predefined variables, whose values are assigned by a successful match:

- `$&` is automatically assigned to the string that last matched a regular expression as in this program:

```
$line = "Tell me, O muse, of that ingenious hero
  who travelled far and wide after he had sacked
```

```
  the famous town of Troy.";
$line =~ m/,.*,/;
print $&, "\n";
```

which prints

```
, O muse,
```

- `$`` and `$'` contain, respectively, the strings before and after the matched pattern and the program:

```
$line = "Tell me, O muse, of that ingenious hero
  who travelled far and wide after he had sacked
  the famous town of Troy.";
$line =~ m/,.*,/;
print "Before: ", $`, "\n";
print "After: ", $', "\n";
```

prints

```
Before: Tell me
After:  of that ingenious hero
  who travelled far and wide after he had sacked
  the famous town of Troy.
```

There are a couple of other predefined variables. The complete reference on them is the perlvar section of the Perl manual. A word of caution, however: many people consider the predefined variables dangerous, especially `$_`. It is a good practice, at least for beginners, to try to avoid them. They make programs hard to read and may introduce bugs.

2.5 Finding Concordances

Concordances of a word, an expression, or more generally any string in a corpus are easy to obtain with Perl or Prolog. In our programs, we will represent the corpus as one single big string, and concordancing will simply consist in matching the pattern we are searching as a substring of the whole list. There will be no need then to consider the corpus structure, that is, whether it is made of blanks, words, sentences, or paragraphs.

2.5.1 Concordances in Perl

To have a convenient input of the concordance parameters – the file name, the pattern to search, and the span size of the concordance – we will design the Perl program so that it can read them from the command line as in

```
perl -w concordance.pl corpus.txt pattern_to_search 15
```

These arguments are passed to Perl by the operating system in the form of an array. Before writing the program, we introduce this data type now.

Arrays in Perl

Arrays in Perl are data structures that can hold any number of elements of any type. Their name begins with an at sign, `@`, for example, `@array`. Each element has a position where the programmer can store and read data using the position index.

An array grows or shrinks automatically when elements are appended, inserted, or deleted. Perl manages the memory without any intervention from the programmer. Here are some examples of arrays:

```
@array1 = ();         # The empty array
@array2 = (1, 2, 3); # Array containing 1, 2, and 3

$var1 = 3.14;
$var2 = "my string";
@array3 = (1, $var1, "Prolog", $var2);
# Array containing four elements of different type

@array4 = (@array2,@array3);
#Same as (1, 2, 3, 1, 3.14, "Prolog", "my string")
```

Reading or assigning a value to a position of the array is done using its index between square brackets starting from 0:

```
print $array2[1];  # prints 2
```

If an element is assigned to a position that did not exist before, Perl grows the array to store it. The positions in-between are not initialized. They hold the value `undef`:

```
$array4[10] = 10;
print $array4[10]; # prints 10
print $array4[9];
# prints a message telling it is undefined
```

The existence of a variable can be tested using the `defined` Boolean function as in:

```
if (defined($array4[9])) {
  print "yes", "\n";
} else {
  print "no", "\n";
}
```

If an `undef` value is used as a number, it is considered to be a zero. The next two lines print 1.

```
$array4[9]++;
print $array4[9];
```

The variable `$#array` is the index of the last element of the array. It can be assigned to grow or shrink the array:

```
$length4 = $#array4;
print $length4;    # prints 10
print $#array2;    # prints 2
$#array4 = 5;      # shrinks the array to 6 elements.
                   # Other elements are lost.
print $array4[10];
# prints a message telling it is undefined
$#array2 = 10;     # extends the array to 11 elements.
                   # Indices 3..10 are undefined.
```

You can also assign a complete array to an array and an array to a list of variables as in:

```
@array5 = @array2;
($v1, $v2, $v3) = @array2;
```

where `@array5` contains a copy of `@array2`, and `$v1`, `$v2`, `$v3` contain, respectively, 1, 2, and 3.

Printing Concordances in Perl

Now let us write a concordance program modified from Cooper (1999). We use three arguments in the command line: the file name, the pattern to search, and the span size. Peal reads them and stores them in an array with the reserved name: `@ARGV`. We assign these arguments, respectively, to `$file_name`, `$pattern`, and `$width`.

We open the file using the `open` function, which assigns the stream to the `FILE` identifier. If `open` fails, the program exits using `die` and prints a message to inform us that it could not open the file. The notation `<FILE>` designates the input stream, which is assigned to the `$line` variable. We read all the text and we assign it to the `$text` variable.

In addition to single words, we may want to search concordances of a phrase such as *the Achaeans*. Depending on the text formatting, the phrase's words can be on the same line or spread on two lines of text as in:

```
I see that the Achaeans are subject to you in great
multitudes.
...
the banks of the river Sangarius; I was their ally,
and with them when the Amazons, peers of men, came up
against them, but even they were not so many as the
Achaeans."
```

The Perl string `the Achaeans` matches the first occurrence of the phrase in the text, but not the second one as the two words are separated by a line break.

There are two ways to cope with that:

- We can modify `$pattern`, the phrase to search, so that it matches across sequences of line breaks, tabulations, or spaces. To do this, we replace the sequences of spaces in `$pattern` with the generic white space character class: `$pattern =~ s/ +/\\s+/g;`
- The second possibility is to normalize the text, `$text`, so that the line breaks and all kinds white spaces in the text are replaced with a standard space: `$text =~ s/\s+/ /g;`

Both solutions can deal with the multiple conventions to mark line breaks, the two most common ones being `\n` and `\r\n` adopted, respectively, by Unix and Windows. Moreover, the text normalization makes it easier to format the concordance output and print the results. In our program, we will keep both instructions, although they are somewhat redundant.

Finally, we use a `while` loop to match the pattern with `$width` characters to the left and to the right. We create a back reference by setting parentheses around the regular expression and we print its value stored in `$1`. We do this for all the occurrences of the pattern in `$text` using the combination of `while` and the `g` modifier as we saw in Sect. 2.4.7.

```
($file_name, $pattern, $width) = @ARGV;
open(FILE, "$file_name") ||
      die "Could not open file $file_name.";
while ($line = <FILE>) {
  $text .= $line;
}
$pattern =~ s/ +/\\s+/g;
      # spaces match tabs and new lines
$text =~ s/\s+/ /g;
      # line breaks and blank sequences are replaced
      # by spaces
while ($text =~ m/(.{0,$width}$pattern.{0,$width})/g) {
      # matches the pattern with 0..width
      # to the right and left
  print "$1\n"; #$1 contains the match
}
```

Now let us run the command:

```
perl -w concordance.pl odyssey.txt Penelope 25

he suitors of his mother Penelope, who persist in eating u
ace dying out yet, while Penelope has such a fine son as y
 laid upon the Achaeans. Penelope, daughter of Icarius, he
 blood of Ulysses and of Penelope in your veins I see no l
 his long-suffering wife Penelope, and his son Telemachus,
```

```
ngs. It was not long ere Penelope came to know what the su
he threshold of her room Penelope said: "Medon, what have
```

2.5.2 Concordances in Prolog

Writing a basic concordance program is also relatively easy in Prolog. Before conducting the search, and just as in the Perl program, it is preferable to normalize all kinds of white spaces in the text.

Normalizing White Spaces

The normalization of white spaces corresponds to the substitution of the `\s+` expression with a blank space. We implement it with the `normalize/2` predicate that replaces sequences of contiguous white spaces with one single blank space. We use `memberchk/2`, a faster, nonbacktracking version of `member/2`, to determine whether a character is a white space.

```
% normalize(+List, -NormalizedList)
% replaces contiguous white spaces with one blank

normalize([C1, C2 | L1], [' ' | L2]) :-
  memberchk(C1, [' ', '\t', '\n', '\r', '\f']),
  memberchk(C2, [' ', '\t', '\n', '\r', '\f']),
  !,
  normalize([C2 | L1], [' ' | L2]).
normalize([C1 | L1], [' ' | L2]) :-
  memberchk(C1, [' ', '\t', '\n', '\r', '\f']),
  !,
  normalize(L1, L2).
normalize([C1 | L1], [C1 | L2]) :-
  \+ memberchk(C1, [' ', '\t', '\n', '\r', '\f']),
  !,
  normalize(L1, L2).
normalize([], []).
```

Searching the Pattern

We implement the concordance search with two auxiliary predicates:

```
prefix(+List, +Span, -Prefix)
```

that extracts the prefix of a list with up to `Span` characters, and

```
prepend(+List, +Span, -PrependedList)
```

that adds `Span` variables onto the beginning of a list.

The top-level predicate, `concordance/4`, finds `Pattern` in `List` and returns the first `Line` where it occurs. `Span` is the window size, for example, 25 characters to the left and to the right, within which `Pattern` will be displayed. We first prepend `Pattern` with `Span` variables before it to match the pattern and its right context. We find it with a combination of two `append/3` calls; then we use `prefix/3` to extract up to `Span` characters after it.

```
% concordance(+Pattern, +List, +Span, -Line)
% finds Pattern in List and displays the Line
% where it appears within Span characters surrounding it.

concordance(Pattern, List, Span, Line) :-
  atom_chars(Pattern, LPattern),
  prepend(LPattern, Span, LeftPattern),
  append(_, Rest, List),
  append(LeftPattern, End, Rest),
  prefix(End, Span, Suffix),
  append(LeftPattern, Suffix, LLine),
  atom_chars(Line, LLine).

% prefix(+List, +Span, -Prefix) extracts the prefix
% of List with up to Span characters.
% The second rule is to check the case where there
% are less than Span character in List.

prefix(List, Span, Prefix) :-
  append(Prefix, _, List),
  length(Prefix, Span),
  !.
prefix(Prefix, Span, Prefix) :-
  length(Prefix, L),
  L < Span.

% prepend(+List, +Span, -Prefix) adds Span variables
% to the beginning of List.

prepend(Pattern, Span, List) :-
  prepend(Pattern, Span, Pattern, List).

prepend(_, 0, List, List) :- !.
prepend(Pattern, Span, List, FList) :-
  Span1 is Span - 1,
  prepend(Pattern, Span1, [X | List], FList).
```

Let us apply this program to retrieve the concordances of *Helen* in the *Iliad*. We use `read_file/2` defined in Sect. A.16.2, and we make `concordance/4` backtrack until all the occurrences have been found:

```
?- read_file('iliad.txt', L), normalize(L, L2),
concordance('Helen', L2, 25, C), write(C), nl, fail.

e glory of still keeping Helen, for whose sake so many
e glory of still keeping Helen, for whose sake so many
suffered for the sake of Helen. Nevertheless, if any ma
suffered for the sake of Helen. The men of Pylos and Ar
fight in their midst for Helen and all her wealth. Let
 in the midst of you for Helen and all her wealth. Let
. Meanwhile Iris went to Helen in the form of her siste
s spoke the goddess, and Helen's heart yearned after he
in a wood. When they saw Helen coming towards the tower
a king." "Sir," answered Helen, "father of my husband,
...
No
```

Because the pattern is prepended with exactly `Span` variables, the concordance program will not examine the first `Span` characters of the file. This means that it will not find a possible pattern in this sublist. In our example above, the program finds all the occurrences of *Helen* except the ones that could occur in the first 25 characters of the text. This is easily corrected in the program and is left as an exercise.

2.6 Approximate String Matching

So far, we have used regular expressions to match exact patterns. However, in many applications, such as in spell checkers, we need to extend the match span to search a set of related patterns or strings. In this section, we review techniques to carry out approximate or inexact string matching.

2.6.1 Edit Operations

A common method to create a set of related strings is to apply a sequence of edit operations that transforms a source string s into a target string t. The operations are carried out from left to right using two pointers that mark the position of the next character to edit in both strings:

- The copy operation is the simplest. It copies the current character of the source string to the target string. Evidently, the repetition of copy operations produces equal source and target strings.
- Substitution replaces one character from the source string by a new character in the target string. The pointers are incremented by one in both the source and target strings.

Table 2.15 Typographical errors (typos) and corrections. Strings differ by one operation. The *correction* is the source and the *typo* is the target. Unless specified, other operations are just copies (After Kernighan et al. (1990))

Typo	Correction	Source	Target	Position	Operation
acress	actress	–	t	2	Deletion
acress	cress	a	–	0	Insertion
acress	caress	ac	ca	0	Transposition
acress	access	r	c	2	Substitution
acress	across	e	o	3	Substitution
acress	acres	s	–	4	Insertion
acress	acres	s	–	5	Insertion

- Insertion inserts a new character in the target string. The pointer in the target string is incremented by one, but the pointer in the source string is not.
- Deletion deletes the current character in the target string, i.e., the current character is not copied in the target string. The pointer in the source string is incremented by one, but the pointer in the target string is not.
- Reversal (or transposition) copies two adjacent characters of the source string and transposes them in the target string. The pointers are incremented by two characters.

Kernighan et al. (1990) illustrate these operations with the misspelled word *acress* and its possible corrections (Table 2.15).

If we allow only one edit operation on a source string of length n, and if we consider an alphabet of 26 unaccented letters, the deletion will generate n new strings; the insertion, $(n + 1) \times 26$ strings; the substitution, $n \times 25$; and the transposition, $n - 1$ new strings.

2.6.2 Minimum Edit Distance

Complementary to edit operations, edit distances measure the similarity between strings. They assign a cost to each edit operation, usually 0 to copies and 1 to deletions and insertions. Substitutions and transpositions correspond both to an insertion and a deletion. We can derive from this that they each have a cost of 2. Edit distances tell how far a source string is from a target string: the lower the distance, the closer the strings.

Given a set of edit operations, the minimum edit distance is the operation sequence that has the minimal cost needed to transform the source string into the target string. If we restrict the operations to copy/substitute, insert, and delete, we can represent the edit operations using a table, where the distance at a certain position in the table is derived from distances in adjacent positions already computed. This is expressed by the formula:

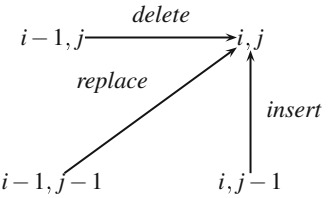


Fig. 2.10 Edit operations

$$edit\_distance(i, j) = \min \begin{pmatrix} edit\_distance(i-1, j) + del\_cost \\ edit\_distance(i-1, j-1) + subst\_cost \\ edit\_distance(i, j-1) + ins\_cost \end{pmatrix}.$$

The boundary conditions for the first row and the first column correspond to a sequence of deletions and of insertions. They are defined as $edit\_distance(i, 0) = i$ and $edit\_distance(0, j) = j$.

We compute the cell values as a walk through the table from the beginning of the strings at the bottom left corner, and we proceed upward and rightward to fill adjacent cells from those where the value is already known. Arrows in Fig. 2.10 represent the three edit operations, and Table 2.16 shows the distances to transform *language* into *lineage*. The value of the minimum edit distance is 5 and is shown at the upper right corner of the table.

The minimum edit distance algorithm is part of the **dynamic programming** techniques. Their principles are relatively simple. They use a table to represent data, and they solve a problem at a certain point by combining solutions to subproblems. Dynamic programming is a generic term that covers a set of widely used methods in optimization.

2.6.3 Computing the Minimum Edit Distance in Perl

To implement the minimum edit distance in Perl, we use the `length` built-in function to compute the length of the source and target, and `split(//, $string)` to convert a string into an array of characters. The instruction

```
@array = split(regex, $string)
```

breaks up the `$string` variable as many times as `regex` matches in `$string`. The `regex` expression acts as a separator, and the string pieces are assigned sequentially to `@array`. In the minimum edit distance program, `regex` is reduced to nothing and assigns all the characters `$string` as elements of `@array`.

```
($source, $target) = @ARGV;
$length_s = length($source);
$length_t = length($target);
# Initialize first row and column
```

Table 2.16 Distances between *language* and *lineage*

e	7	6	5	6	5	6	7	6	**5**
g	6	5	4	5	4	5	6	5	6
a	5	4	3	4	5	6	5	6	7
e	4	3	4	3	4	5	6	7	6
n	3	2	3	2	3	4	5	6	7
i	2	1	2	3	4	5	6	7	8
l	1	0	1	2	3	4	5	6	7
Start	0	1	2	3	4	5	6	7	8
–	**Start**	**l**	**a**	**n**	**g**	**u**	**a**	**g**	**e**

```
for ($i = 0; $i <= $length_s; $i++) {
  $table[$i][0] = $i;
}
for ($j = 0; $j <= $length_t; $j++) {
  $table[0][$j] = $j;
}
# Get the characters. Start index is 0
@source = split(//, $source);
@target = split(//, $target);
# Fills the table.
# Start index of rows and columns is 1
for ($i = 1; $i <= $length_s; $i++) {
  for ($j = 1; $j <= $length_t; $j++) {
    # Is it a copy or a substitution?
    $cost = ($source[$i-1] eq $target[$j-1]) ? 0: 2;
    # Computes the minimum
    $min = $table[$i-1][$j-1] + $cost;
    if ($min > $table[$i][$j-1] + 1) {
      $min = $table[$i][$j-1] + 1;
    }
    if ($min > $table[$i-1][$j] + 1) {
      $min = $table[$i-1][$j] + 1;
    }
    $table[$i][$j] = $min;
  }
}
print "Minimum distance: ",
$table[$length_s][$length_t], "\n";
```

2.6.4 Searching Edits in Prolog

Once we have filled the table, we can search the operation sequences that correspond to the minimum edit distance. Such a sequence is also called an **alignment**.

The depth-first strategy is an economical way to traverse a search space. It is easy to implement in Prolog and has low memory requirements. The problem with it is that it blindly selects the paths to follow and can explore very deep nodes while ignoring shallow ones. To avoid this, we apply a variation of the depth-first search where we fix the depth in advance to the minimum edit distance. We assign it to the call parameter `Cost` of `edit_distance/4`.

The code of the depth-limited search is similar to the depth-first program (see Appendix A, Sect. A.15.2). We add a counter in the recursive case that represents the current search depth and we increment it until we have reached the depth limit. We compute each individual edit operation and its cost with the `edit_operation/6` predicate.

```
% edit_distance(+Source, +Target, -Edits, +Cost).
edit_distance(Source, Target, Edits, Cost) :-
  edit_distance(Source, Target, Edits, 0, Cost).

edit_distance([], [], [], Cost, Cost).
edit_distance(Source, Target, [EditOp | Edits], Cost,
    FinalCost) :-
  edit_operation(Source, Target, NewSource, NewTarget,
      EditOp, CostOp),
  Cost1 is Cost + CostOp,
  edit_distance(NewSource, NewTarget, Edits, Cost1,
      FinalCost).

% edit_operation carries out one edit operation
% between a source string and a target string.
edit_operation([Char | Source], [Char | Target], Source,
    Target, ident, 0).
edit_operation([SChar | Source], [TChar | Target], Source,
    Target, sub(SChar,TChar), 2) :-
  SChar \= TChar.
edit_operation([SChar | Source], Target, Source, Target,
    del(SChar), 1).
edit_operation(Source, [TChar | Target], Source, Target,
    ins(TChar), 1).
```

Using backtracking, Prolog finds all the alignments. We obtain with the minimum distance of 5:

```
?- edit_distance([l,a,n,g,u,a,g,e], [l,i,n,e,a,g,e], E, 5).

E = [ident, sub(a, i), ident, sub(g, e), del(u), ident,
     ident, ident] ;

E = [ident, sub(a, i), ident, del(g), sub(u, e), ident,
     ident, ident] ;
```

	First alignment	Third alignment
Without epsilon symbols	l a n g *u* a g e | | | | / / / l i n e a g e	l a n *g* *u* a g e | | | / / / l i n *e* a g e
With epsilon symbols	l a n g u a g e | | | | | | | | l i n e ε a g e	l a n g u ε a g e | | | | | | | | | l i n ε ε e a g e

Fig. 2.11 Alignments of *lineage* and *language*. The figure contains two possible representations of them. In the *upper row*, the deletions in the source string are in italics, as are the insertions in the target string. The *lower row* shows a synchronized alignment, where deletions in the source string as well as the insertions in the target string are aligned with epsilon symbols (null symbols)

```
E = [ident, sub(a, i), ident, del(g), del(u), ins(e), ident,
     ident, ident]
...
```

with 15 possible alignments in total. Figure 2.11 shows the first and third ones.

We can apply this Prolog search program alone to find the edit distance. We avoid going down an infinite path with an iterative deepening. We start with an edit distance of 0 (the `Cost` parameter) and we increment it – 1, 2, 3, 4 – until we find the minimum edit distance. The first searches will fail, and the first one that succeeds corresponds to the minimum distance.

We can also compute these alignments in Perl. A frequently used technique is to consider each cell in Table 2.16 and to store the coordinates of all the adjacent cells that enabled us to fill it. For instance, the program filled the last cell of coordinates (8, 7), containing 5 (`$table[8][7]`), using the content of cell (7, 6). The storage can be a parallel table, where each cell contains the coordinates of the immediately preceding positions (the backpointers). Starting from the last cell down to the bottom left cell, (0, 0), we traverse the table from adjacent cell to adjacent cell to recover all the alignments. This program is left as an exercise (Exercise 2.9).

2.7 Further Reading

Corpora are now easy to obtain. Organizations such as the Linguistic Data Consortium and ELRA collect and distribute texts in many languages. Although not widely cited, Busa (1974, 1996) is the author of the first large computerized corpus, the *Index Thomisticus*, a complete edition of the works of Saint Thomas Aquinas. The corpus, which is entirely lemmatized, is available online (http://www.corpusthomisticum.org/). FranText is also a notable early corpus of more than 100 million words. It helped write the *Trésor de la langue française* (Imbs and Quemada 1971–1994), a comprehensive French dictionary. Other early corpora include the

Bank of English, which contributed to the *Collins COBUILD Dictionary* (Sinclair 1987).

Concordancing plays a role today that goes well beyond lexicography. Google, Bing, and other web search engines can be considered as modern avatars of concordancers as they return a small passage – a snippet – of a document, where a phrase or words are cited. The Dominicans who created the first concordances in the thirteenth century surely did not forecast the future of their brainchild and the billions of searches per day it would entail. For a history of early concordances to the scriptures, see Rouse and Rouse (1974).

The code examples in these chapter enabled us to search strings and patterns in a corpus. For large volumes of text, a more realistic application would first index all the words before a user can search them. Manning et al. (2008) is a good review of indexing techniques. Lucene (http://lucene.apache.org/) is a widely used system to carry out text indexing and search.

Text and corpus analysis are an active focus of research in computational linguistics. Kaeding (1897) and Estoup (1912), the latter cited in Petruszewycz (1973), were among the pioneers in this field, at the turn of the twentieth century, when they used corpora to carry out systematic studies on letter and word frequencies for stenography. Paradoxically, natural language processing conducted by computer scientists largely ignored corpora until the 1990s, when they rediscovered techniques routinely used in the humanities. For a short history, see Zampolli (2003) and Busa (2009).

Roche and Schabes (1997, Chap. 1) is a concise and clear introduction to automata theory. It makes extensive use of mathematical notations, however. Hopcroft et al. (2007) is a standard and comprehensive textbook on automata and regular expressions. Friedl (2006) is a thorough presentation of regular expressions oriented toward programming techniques and applications.

Although the idea of automata underlies some mathematical theories of the nineteenth century (such as those of Markov, Gödel, or Turing), Kleene (1956) was the first to give a formal definition. He also proved the equivalence between regular expressions and FSA. Thompson (1968) was the first to implement a widely used editor embedding a regular expression tool: Global/Regular Expression/Print, better known as `grep`.

There are several FSA toolkits available from the Internet. The Perl Compatible Regular Expressions (PCRE) library is an open-source set of functions that implements the Perl regex syntax. It is written in C by Philip Hazel (http://www.pcre.org/). The FSA utilities (van Noord and Gerdemann 2001) is a Prolog package to manipulate regular expressions, automata, and transducers (http://odur.let.rug.nl/~vannoord/Fsa/). The OpenFst library (Allauzen et al. 2007; Mohri et al. 2000) is another set of tools (http://www.openfst.org/). Both include rational operations – union, concatenation, closure, reversal – and equivalence transformation – ε-elimination, determinization, and minimization.

Exercises

2.1. Implement the automaton in Fig. 2.5.

2.2. Implement a Prolog program to automatically construct an automaton to search a given input string.

2.3. Write a regular expression that finds occurrences of *honour* and *honor* in a text.

2.4. Write a regular expression that finds lines containing all the vowels *a*, *e*, *i*, *o*, *u*, in that order.

2.5. Write a regular expression that finds lines consisting only of letters *a*, *b*, or *c*.

2.6. List the strings generated by the expressions:

```
(ab)*c
(a.)*c
(a|b)*
a|b*|(a|b)*a
a|bc*d
```

2.7. Complement the Prolog concordance program to sort the lines according to words appearing on the right of the string to search.

2.8. Write the iterative deepening search in Prolog to find the minimum edit distance.

2.9. Extend the Perl program in Sect. 2.6.3 to find the alignments. See the last paragraph of Sect. 2.6.4 for an idea of the algorithm.

Chapter 3
Encoding and Annotation Schemes

ʽΕλλάδι φωνήεντα καὶ ἔμφρονα δῶρα κομίζων
γλώσσης ὄργανα τεῦξεν ὁμόθροα, συμφυέος δὲ
ἁρμονίης στοιχηδὸν ἐς ἄζυγα σύζυγα μίξας
γραπτὸν ἀσιγήτοιο τύπον τορνώσατο σιγῆς,
πάτρια θεσπεσίης δεδαημένος ὄργια τέχνης,

Nonnus Panopolitanus, *Dionysiaca*, Book IV, verses 261–265. Fifth century.

> *But Cadmos [from Sidon in Phoenicia] brought gifts of voice and thought for all Hellas; he fashioned tools to echo the sounds of the tongue, he mingled sonant and consonant in one order of connected harmony. So he rounded off a graven model of speaking silence; for he had learnt the secrets of his country's sublime art.*
>
> Translation W. H. D. Rouse. Loeb Classical Library.

3.1 Encoding Texts

At the most basic level, computers only understand binary digits and numbers. Corpora as well as any computerized texts have to be converted into a digital format to be read by machines. From their American early history, computers inherited encoding formats designed for the English language. The most famous one is the **American Standard Code for Information Interchange** (ASCII). Although well established for English, the adaptation of ASCII to other languages led to clunky evolutions and many variants. It ended (temporarily?) with Unicode, a universal scheme compatible with ASCII and intended to cover all the scripts of the world.

We saw in Chap. 2 that some corpora include linguistic information to complement raw texts. This information is conveyed through annotations that

P.M. Nugues, *Language Processing with Perl and Prolog*, Cognitive Technologies,
DOI 10.1007/978-3-642-41464-0_3, © Springer-Verlag Berlin Heidelberg 2014

describe quantities of structures. They range from text organization, such as titles, paragraphs, and sentences, to semantic information including grammatical data, part-of-speech labels, or syntactic structures, etc. In contrast to character encoding, no annotation scheme has yet reached a level where it can claim to be a standard. However, the **Extensible Markup Language** (XML), a language to define annotations, is well underway to unify them under a shared markup syntax. XML in itself is not an annotation language. It is a scheme that enables users to define annotations within a specific framework.

In this chapter, we will introduce the most useful character encoding schemes and review the basics of XML. We will examine related topics of standardized presentation of time and date, and how to sort words in different languages. We will then outline two significant theoretical concepts behind codes – entropy and perplexity – and how they can help design efficient codes. Entropy is a very versatile measure with many applications. We will use it to learn automatically decision trees from data and build classifiers. This will enable us to review our first machine-learning algorithm. Machine learning is now instrumental in most areas of natural language processing and we will conclude this chapter with the description of three other linear classifiers that are among the most widely used.

3.2 Character Sets

3.2.1 Representing Characters

Words, at least in European languages, consist of characters. Prior to any further digital processing, it is necessary to build an encoding scheme that maps the character or symbol repertoire of a language to numeric values – integers. The Baudot code is one of the oldest electric codes. It uses 5 bits and hence has the capacity to represent $2^5 = 32$ characters: the Latin alphabet and some control commands like the carriage return and the bell. The ASCII code uses 7 bits. It can represent $2^7 = 128$ symbols with positive integer values ranging from 0 to 127. The characters use the contiguous positions from 32 to 126. The values in the range [0..31] and 127 correspond to controls used, for instance, in data transmission (Table 3.1).

ASCII was created originally for English. It cannot handle other European languages that have accented letters, such as *é*, *à*, or other diacritics like *ø* and *ä*, not to mention languages that do not use the Latin alphabet. Table 3.2 shows characters used in French and German that are ignored by ASCII. Most computers used to represent characters on octets – words of 8 bits – and ASCII was extended with the eighth unoccupied bit to the values in the range [128..255] ($2^8 = 256$). Unfortunately, these extensions were not standardized and depended on the operating system. The same character, for instance, *ê*, could have a different encoding in the Windows, Macintosh, and Unix operating systems.

Table 3.1 The ASCII character set arranged in a table consisting of 6 rows and 16 columns. We obtain the ASCII code of a character by adding the first number of its row and the number of the column. For instance, *A* has the decimal code 64 + 1 = 65, and *e* has the code 96 + 5 = 101

	0	1	2	4	3	5	6	7	8	9	10	11	12	13	14	15
32		!	"	#	$	%	&	’	(	)	*	+	,	-	.	/
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
96	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	

Table 3.2 Characters specific to French and German

	French															**German**			
Lowercase	à	â	æ	ç	é	è	ê	ë	î	ï	ô	œ	ù	û	ü	ÿ	ä	ö	ü	ß
Uppercase	À	Â	Æ	Ç	É	È	Ê	Ë	Î	Ï	Ô	Œ	Ù	Û	Ü	Ÿ	Ä	Ö	Ü	

Table 3.3 The ISO Latin 1 character set (ISO-8859-1) covering most characters from Western European languages

	0	1	2	4	3	5	6	7	8	9	10	11	12	13	14	15
160		¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	-	®	¯
176	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
192	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
208	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
224	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
240	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

Table 3.4 The ISO Latin 9 character set (ISO-8859-15) that replaces rare symbols from Latin 1 with the characters œ, Œ, š, Š, ž, Ž, Ÿ, and €. The table only shows rows that differ from Latin 1

	0	1	2	4	3	5	6	7	8	9	10	11	12	13	14	15
160		¡	¢	£	€	¥	Š	§	š	©	ª	«	¬	-	®	¯
176	°	±	²	³	Ž	µ	¶	·	ž	¹	º	»	Œ	œ	Ÿ	¿

The ISO Latin 1 character set (ISO-8859-1) is a standard that tried to reconcile Western European character encodings (Table 3.3). Unfortunately, Latin 1 was ill-designed and forgot characters such as the French *Œ*, *œ*, the German quote „ or the Dutch *ij*, *IJ*. Operating systems such as Windows and Mac OS used a variation of it that they had to complement with the missing characters. Later, ISO Latin 9 (ISO-8859-15) updated Latin 1 (Table 3.4). It restored forgotten French and Finnish characters and added the euro currency sign, €.

3.2.2 Unicode

While ASCII has been very popular, its 128 positions could not support the characters of many languages in the world. Therefore a group of companies formed a consortium to create a new, universal coding scheme: Unicode. Unicode has quickly replaced older encoding schemes, and Windows, Mac OS, and Java platforms have now adopted it while sometimes ensuring backward compatibility.

The initial goal of Unicode was to define a superset of all other character sets, ASCII, Latin 1, and others, to represent all the languages of the world. The Unicode consortium has produced character tables of most alphabets and scripts of European, Asian, African, and Near Eastern languages, and assigned numeric values to the characters. Unicode started with a 16-bit code that could represent up to 65,000 characters. The code was subsequently extended to 32 bits with values ranging from 0 to 10FFFF in hexadecimal. This Unicode code space has then a capacity of 1,114,112 characters.

The standardized set of Unicode characters is called the universal character set (UCS). It is divided into several planes, where the basic multilingual plane (BMP) contains all the common characters, with the exception of some Chinese ideograms. Characters in the BMP fit on a 2-octet code (UCS-2). The 4-octet code (UCS-4) can represent, as we saw, more than a million characters. It covers all the UCS-2 characters and rare characters: historic scripts, some mathematical symbols, private characters, etc.

Unicode groups characters or symbols by script – Latin, Greek, Cyrillic, Hebrew, Arabic, Indic, Japanese, Chinese – and identifies each character by a single hexadecimal number, called a code point, and a name as

```
U+0041 LATIN CAPITAL LETTER A
U+0042 LATIN CAPITAL LETTER B
U+0043 LATIN CAPITAL LETTER C

...

U+0391 GREEK CAPITAL LETTER ALPHA
U+0392 GREEK CAPITAL LETTER BETA
U+0393 GREEK CAPITAL LETTER GAMMA
```

The `U+` symbol means that the number after it corresponds to a Unicode position.

Unicode allows the composition of accented characters from a base character and one or more diacritics. That is the case for the French $\hat{E}$ or the Scandinavian $\mathring{A}$. Both characters have a single code point:

```
U+00CA LATIN CAPITAL LETTER E WITH CIRCUMFLEX
U+00C5 LATIN CAPITAL LETTER A WITH RING ABOVE
```

They can also be defined as a sequence of two keys: E + ˆ and A + °, corresponding to respectively to

```
U+0045 LATIN CAPITAL LETTER E
```

```
U+0302 COMBINING CIRCUMFLEX ACCENT
```

and

```
U+0041 LATIN CAPITAL LETTER A
U+030A COMBINING RING ABOVE
```

The resulting graphical symbol is called a grapheme. A grapheme is a "natural" character or a symbol. It may correspond to a single code point as *E* or *A*, or result from a composition as $\hat{E}$ or $\mathring{A}$.

Unicode allocates contiguous blocks of code to scripts from `U+0000`. They start with alphabetic scripts: Latin, Greek, Cyrillic, Hebrew, Arabic, etc., then the symbols area, and Asian ideograms or alphabets. Ideograms used by the Chinese, Japanese, and Korean (CJK) languages are unified to avoid duplication. Table 3.5 shows the script allocation. The space devoted to Asian scripts occupies most of the table.

3.2.3 Unicode Character Properties

Unicode associates a list of properties to each code point. This list is defined in the Unicode character database and includes the name of the code point (character name), its so-called general category – whether it is a letter, digit, punctuation, symbol, mark, or other – the name of its script, for instance Latin or Arabic, and its code block (The Unicode Consortium 2012).

Each property has a set of possible values. Table 3.6 shows this set for the general category, where each value consists of one or two letters. The first letter is a major class and the second one, a subclass of it. For instance, `L` corresponds to a letter, `Lu` to an uppercase letter; `Ll`, to a lowercase letter, while `N` corresponds to a number and `Nd`, to a number, decimal digit.

We can use these Unicode properties in Perl regular expressions to search characters, categories, blocks, and scripts by their names. We match a specific code point with the `\N{name}` construct, where `name` is the name of the code point, or with its hexadecimal `\x{hexcode}` code as:

- `\N{LATIN CAPITAL LETTER E WITH CIRCUMFLEX}` and `\x{CA}` that match Ê and
- `\N{GREEK CAPITAL LETTER GAMMA}` and `\x{393}` that match γ.

We match code points in blocks, categories, and scripts with the `\p{property}` construct introduced in Sect. 2.3.5, or its complement `\P{property}` to match code points without the property:

For a block, we build a Perl regex by replacing `property` with the block name in Table 3.5. Perl also requires an `In` prefix and that white spaces are replaced with underscores as `InBasic_Latin` or `InLatin_Extended-A`.

Table 3.5 Unicode subrange allocation of the universal character set (simplified)

Code	Name	Code	Name
0000	Basic Latin	1400	Unified Canadian Aboriginal Syllabics
0080	Latin-1 Supplement	1680	Ogham
0100	Latin Extended-A	16A0	Runic
0180	Latin Extended-B	1780	Khmer
0250	IPA Extensions	1800	Mongolian
02B0	Spacing Modifier Letters	1E00	Latin Extended Additional
0300	Combining Diacritical Marks	1F00	Greek Extended
0370	Greek and Coptic	2000	General Punctuation
0400	Cyrillic	2800	Braille Patterns
0500	Cyrillic Supplement	2E80	CJK Radicals Supplement
0530	Armenian	2F00	Kangxi Radicals
0590	Hebrew	3000	CJK Symbols and Punctuation
0600	Arabic	3040	Hiragana
0700	Syriac	30A0	Katakana
0750	Arabic Supplement	3100	Bopomofo
0780	Thaana	3130	Hangul Compatibility Jamo
07C0	NKo	3190	Kanbun
0800	Samaritan	31A0	Bopomofo Extended
0900	Devanagari	3200	Enclosed CJK Letters and Months
0980	Bengali	3300	CJK Compatibility
0A00	Gurmukhi	3400	CJK Unified Ideographs Extension A
0A80	Gujarati	4E00	CJK Unified Ideographs
0B00	Oriya	A000	Yi Syllables
0B80	Tamil	A490	Yi Radicals
0C00	Telugu	AC00	Hangul Syllables
0C80	Kannada	D800	High Surrogates
0D00	Malayalam	E000	Private Use Area
0D80	Sinhala	F900	CJK Compatibility Ideographs
0E00	Thai	10000	Linear B Syllabary
0E80	Lao	10140	Ancient Greek Numbers
0F00	Tibetan	10190	Ancient Symbols
1000	Myanmar	10300	Old Italic
10A0	Georgian	10900	Phoenician
1100	Hangul Jamo	10920	Lydian
1200	Ethiopic	12000	Cuneiform
13A0	Cherokee	100000	Supplementary Private Use Area-B

For example, `\p{InGreek_and_Coptic}` matches code points in the Greek and Coptic block whose Unicode range is `[0370..03FF]`. This roughly corresponds to the Greek characters. However, some of the code points in this block are not assigned and some others are Coptic characters.

For a general category, we use either the short or long names in Table 3.6 as `Letter` or `Lu`. For example, `\p{Currency_Symbol}` matches currency symbols and `\P{L}` all nonletters.

Table 3.6 Values of the general category with their short and long names. The *left column* lists to the major classes, and the *right* one the subclasses (After The Unicode Consortium (2012))

Major classes		**Subclasses**	
Short	**Long**	**Short**	**Long**
L	Letter		
		Lu	Uppercase_Letter
		Ll	Lowercase_Letter
		Lt	Titlecase_Letter
		Lm	Modifier_Letter
		Lo	Other_Letter
M	Mark		
		Mn	Nonspacing_Mark
		Mc	Spacing_Mark
		Me	Enclosing_Mark
N	Number		
		Nd	Decimal_Number
		Nl	Letter_Number
		No	Other_Number
P	Punctuation		
		Pc	Connector_Punctuation
		Pd	Dash_Punctuation
		Ps	Open_Punctuation
		Pe	Close_Punctuation
		Pi	Initial_Punctuation
		Pf	Final_Punctuation
		Po	Other_Punctuation
S	Symbol		
		Sm	Math_Symbol
		Sc	Currency_Symbol
		Sk	Modifier_Symbol
		So	Other_Symbol
Z	Separator		
		Zs	Space_Separator
		Zl	Line_Separator
		Zp	Paragraph_Separator
C	Control		
		Cc	Control
		Cf	Format
		Cs	Surrogate
		Co	Private_Use
		Cn	Unassigned

For a script, we use its name in Table 3.7. The regex will match all the code points belonging to this script, even if they are scattered in different blocks. For example, the regex `\p{Greek}` matches the Greek characters in the Greek and Coptic, Greek Extended, and Ancient Greek Numbers blocks, respectively

Table 3.7 Unicode script names

Arabic Armenian Avestan Balinese Bamum Bengali Bopomofo Braille Buginese Buhid Canadian_Aboriginal Carian Cham Cherokee Common Coptic Cuneiform Cypriot Cyrillic Deseret Devanagari Egyptian_Hieroglyphs Ethiopic Georgian Glagolitic Gothic Greek Gujarati Gurmukhi Han Hangul Hanunoo Hebrew Hiragana Imperial_Aramaic Inherited Inscriptional_Pahlavi Inscriptional_Parthian Javanese Kaithi Kannada Katakana Kayah_Li Kharoshthi Khmer Lao Latin Lepcha Limbu Linear_B Lisu Lycian Lydian Malayalam Meetei_Mayek Mongolian Myanmar New_Tai_Lue Nko Ogham Ol_Chiki Old_Italic Old_Persian Old_South_Arabian Old_Turkic Oriya Osmanya Phags_Pa Phoenician Rejang Runic Samaritan Saurashtra Shavian Sinhala Sundanese Syloti_Nagri Syriac Tagalog Tagbanwa Tai_Le Tai_Tham Tai_Viet Tamil Telugu Thaana Thai Tibetan Tifinagh Ugaritic Vai Yi

`[0370..03FF]`, `[1F00..1FFF]`, and `[10140..1018F]`, ignoring the unassigned code points of these blocks and characters that may belong to another script, here Coptic.

Practically, the three instructions below match lines consisting respectively of ASCII characters, of characters in the Greek and Coptic block, and of Greek characters:

```
$line =~ m/^\p{IsASCII}+$/;
$line =~ m/^\p{InGreek_and_Coptic}+$/;
$line =~ m/^\p{Greek}+$/;
```

The Perl program must include the pragma:

```
use charnames ':full';
```

to use Unicode names such as:

```
$line =~ m/\N{GREEK SMALL LETTER ALPHA}/);
```

It must also include `use utf8;` to use UTF-8 characters in the program such as in the instruction:

```
$line =~ m/α/;
```

Finally, to tell Perl of UTF-8 input and output, the command must be run with the option `-CS` as:

```
perl -CS command.pl <input_file
```

Moreover, Perl maintains a list of classes that corresponds to synonyms of Unicode properties or to composite properties. Table 2.11 in Chap. 2 showed some of these classes using the `\p{\ldots}` syntax. For example,

- `\p{IsASCII}` is equivalent to `\p{InBasic_Latin}` and to the range `[\x00-\x7f]`;
- `\p{IsDigit}` is equivalent to `[\p{Nd}]`;
- `\p{IsAlpha}` is equivalent to `[\p{Ll}\p{Lu}\p{Lt}\p{Lo}]`;

Table 3.8 Mapping of 32-bit character code points to 8-bit units according to UTF-8. The xxx corresponds to the rightmost bit values used in the character code points

Range	Encoding
U-0000 – U-007F	0xxxxxxx
U-0080 – U-07FF	110xxxxx 10xxxxxx
U-0800 – U-FFFF	1110xxxx 10xxxxxx 10xxxxxx
U-010000 – U-10FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

- `\p{IsAlnum}` is equivalent to `[\p{Ll}\p{Lu}\p{Lt}\p{Lo}\p{Nd}]`.

Perl created such composite properties to provide equivalent classes available in other regular expression languages, like the POSIX regular expressions.

3.2.4 The Unicode Encoding Schemes

Unicode offers three major different encoding schemes: UTF-8, UTF-16, and UTF-32. The UTF schemes – Unicode transformation format – encode the same data by units of 8, 16, or 32 bits and can be converted from one to another without loss.

UTF-16 was the original encoding scheme when Unicode started with 16 bits. It uses fixed units of 16 bits – 2 bytes – to encode directly most characters. The code units correspond to the sequence of their code points using precomposed characters, such as *Ê* in *FÊTE*

```
0046 00CA 0054 0045
```

or composing it as with E+ˆ in FEˆTE

```
0046 0045 0302 0054 0045
```

Depending on the operating system, 16-bit codes like U+00CA can be stored with highest byte first – `00CA` – or last – `CA00`. To identify how an operating system orders the bytes of a file, it is a possible to insert a *byte order mark* (BOM), a dummy character tag, at the start of the file. UTF-16 uses the code point U+FEFF to tell whether the storage uses the big-endian convention, where the "big" part of the code is stored first, (`FEFF`) or the little-endian one: (`FFFE`).

UTF-8 is a variable-length encoding. It maps the ASCII code characters U+0000 to U+007F to their byte values 00 to 7F. It then takes on the legacy of ASCII. All the other characters in the range U+007F to U+FFFF are encoded as a sequence of two or more bytes. Table 3.8 shows the mapping principles of the 32-bit character code points to 8-bit units.

Let us encode *FÊTE* in UTF-8. The letters *F*, *T*, and *E* are in the range U-00000000 – U-0000007F. Their numeric code values are exactly the same in ASCII and UTF-8. The code point of *Ê* is `U+00CA` and is in the range U-00000080 – U-000007FF. Its binary representation is 0000 0<u>000 11</u>**00 1010**. UTF-8 uses the 11 rightmost bits of `00CA`. The first five underlined bits together with the prefix 110

form the octet 1100 0011 that corresponds to C3 in hexadecimal. The seven next boldface bits with the prefix 10 form the octet 10**00 1010** or 8A in hexadecimal. The letter $\hat{E}$ is then encoded as 1100 0011 1000 1010 or C3 8A in UTF-8. Hence, the word FÊTE and the code points U+0046 U+00CA U+0054 U+0045 are encoded as

```
46 C3 8A 54 45
```

UTF-32 represents exactly the codes points by their code values. One question remains: how does UTF-16 represent the code points above U+FFFF? The answer is: it uses two surrogate positions consisting of a high surrogate in the range U+DC00 .. U+DFFF and a low surrogate in the range U+D800 .. U+DBFF. This is made possible because the Unicode consortium does not expect to assign characters beyond the code point U+10FFFF. Using the two surrogates, characters between U+10000 and U+10FFFF can be converted from UTF-32 to UTF-16, and vice versa.

Finally, the storage requirements of the Unicode encoding schemes are, of course, different and depend on the language. A text in English will have approximately the same size in ASCII and in UTF-8. The size of the text will be doubled in UTF-16 and four times its original size in UTF-32, because all characters take four bytes.

A text in a Western European language will be larger in UTF-8 than in ASCII because of the accented characters: a nonaccented character takes one octet, and an accented one takes two. The exact size will thus depend on the proportion of accented characters. The text size will be twice its ASCII size in UTF-16. Characters in the surrogate space take 4 bytes, but they are very rare and should not increase the storage requirements. UTF-8 is then more compact for most European languages. This is not the case with other languages. A Chinese or Indic character takes, on average, three bytes in UTF-8 and only two in UTF-16.

3.3 Locales and Word Order

3.3.1 Presenting Time, Numerical Information, and Ordered Words

In addition to using different sets of characters, languages often have specific presentations for times, dates, numbers, or telephone numbers, even when they are restricted to digits. Most European languages outside English would write $\pi = 3,14159$ instead of $\pi = 3.14159$. Inside a same language, different communities may have different presentation conventions. The US English date February 24, 2003, would be written 24 February 2003 or February 24th, 2003, in England. It would be abridged 2/24/03 in the United States, 24/02/2003 in Britain, and 2003/02/24 in Sweden. Some communities may be restricted to an administration

Table 3.9 Examples of locales

Locale	Language	Region	Variant
English (United States)	en	US	
English (United Kingdom)	en	GB	
French (France)	fr	FR	
French (Canada)	fr	CA	
German (Germany)	de	DE	
German (Austria)	de	AT	

Table 3.10 Sorting with the ASCII code comparison and the dictionary order

ASCII order	Dictionary order
ABC	abc
Abc	Abc
Def	ABC
aBf	aBf
abc	def
def	Def

or a company, for instance, the military in the US, which writes times and dates differently than the rest of society.

The International Organization for Standardization (ISO) has standardized the identification of languages and communities under the name of **locales**. Each locale uses a set of rules that defines the format of dates, times, numbers, currency, and how to **collate** – sort – strings of characters. A locale is defined by three parameters: the language, the region, and the variant that corresponds to more specific conventions used by a restricted community. Table 3.9 shows some locales for English, French, and German.

One of the most significant features of a locale is the collation component that defines how to compare and order strings of characters. In effect, elementary sorting algorithms consider the ASCII or Unicode values with a predefined comparison operator such as the inequality predicate `@</2` in Prolog. They determine the lexical order using the numerical ranking of the characters.

These basic sorting procedures do not arrange the words in the classical dictionary order. In ASCII as well as in Unicode, lowercase letters have a greater code value than uppercase ones. A basic algorithm would then sort *above* after *Zambia*, which would be quite misleading for most users.

Current dictionaries in English, French, and German use a different convention. The lowercase letters precede their uppercase equivalents when the strings are equal except for the case. Table 3.10 shows the collation results for some strings.

A basic sorting algorithm may suffice for some applications. However, most of the time it would be unacceptable when the ordered words are presented to a user. The result would be even more confusing with accented characters, since their location is completely random in the extended ASCII tables.

In addition, the lexicographic ordering of words varies from language to language. French and English dictionaries sort accented letters as nonaccented ones,

except when two strings are equal except for the accents. Swedish dictionaries treat the letters *Å*, *Ä*, and *Ö* as distinct symbols of the alphabet and sort them after *Z*. German dictionaries have two sorting standards. They process accented letters either as single characters or as couples of nonaccented letters. In the latter case, *Ä*, *Ö*, *Ü*, and *ß* are considered respectively as *AE*, *OE*, *UE*, and *ss*.

3.3.2 The Unicode Collation Algorithm

The Unicode consortium has defined a collation algorithm (Davis and Whistler 2009) that takes into account the different practices and cultures in lexical ordering. It can be parameterized to cover most languages and conventions. It uses three levels of difference to compare strings. We outline their features for European languages and Latin scripts:

- The primary level considers differences between base characters, for instance, between *A* and *B*.
- If there are no differences at the first level, the secondary level considers the accents on the characters.
- And finally, the third level considers the case differences between the characters.

These level features are general, but not universal. Accents are a secondary difference in many languages, but we saw that Swedish sorts accented letters as individual ones and hence sets a primary difference between *A* and *Å*, or *o* and *Ö*. Depending on the language, the levels may have other features.

To deal with the first level, the Unicode collation algorithm defines classes of letters that gather upper- and lowercase variants, accented and unaccented forms. Hence, we have the ordered sets: {a, A, á, Á, à, À, etc.} < {b, B} < {c, C, ć, Ć, ĉ, Ĉ, ç, Ç, etc.} < {e, E, é, É, è, È, ê, Ê, ë, Ë, etc.} < ....

The second level considers the accented letters if two strings are equal at the first level. Accented letters are ranked after their nonaccented counterparts. The first accent is the acute one (´), then come the grave accent (`), the circumflex (ˆ), and the umlaut (¨). So, instances of letter *E* with accents, in lower- and uppercase have the order: {e, E} << {é, É} << {è, È} << {ê, Ê} << {ë, Ë}, where << denotes a difference at the second level. The comparison at the second level is done from the left to the right of a word in English and most languages. It is carried out from the right to the left in French, i.e., from the end of a word to its beginning.

Similarly, the third level considers the case of letters when there are no differences at the first and second levels. Lowercase letters are before uppercase ones, that is, {a} <<< {A}, where <<< denotes a difference at the third level.

Table 3.11 shows the lexical order of *pêcher* 'peach tree' and *Péché* 'sin', together with various conjugated forms of the verbs *pécher* 'to sin' and *pêcher* 'to fish' in French and English. The order takes the three levels into account and the reversed direction of comparison in French for the second level. German adopts the English sorting rules for these accents.

Table 3.11 Lexical order of words with accents. Note the reversed order of the second level comparison in French

English	French
Péché	*pèche*
PÉCHÉ	*pêche*
pèche	*Pêche*
pêche	*Péché*
Pêche	*PÉCHÉ*
pêché	*pêché*
Pêché	*Pêché*
pécher	*pécher*
pêcher	*pêcher*

Some characters are expanded or contracted before the comparison. In French, the letters *Œ* and *Æ* are considered as pairs of two distinct letters: *OE* and *AE*. In traditional German used in telephone directories, *Ä*, *Ö*, *Ü*, and *ß* are expanded into *AE*, *OE*, *UE*, and *ss* and are then sorted as an accent difference with the corresponding letter pairs. In traditional Spanish, *Ch* is contracted into a single letter that sorts between *Cz* and *D*.

The implementation of the collation algorithm (Davis and Whistler 2009, Sect. 4) first maps the characters onto collation elements that have three numerical fields to express the three different levels of comparison. Each character has constant numerical fields that are defined in a collation element table. The mapping may require a preliminary expansion, as for *æ* and *œ* into *ae* and *oe* or a contraction. The algorithm then forms for each string the sequence of the collation elements of its characters. It creates a sort key by rearranging the elements of the string and concatenating the fields according to the levels: the first fields of the string, then second fields, and third ones together. Finally, the algorithm compares two sort keys using a binary comparison that applies to the first level, to the second level in case of equality, and finally to the third level if levels 1 and 2 show no differences.

3.4 Markup Languages

3.4.1 A Brief Background

Corpus annotation uses sets of labels, also called markup languages. Corpus markup languages are comparable to those of standard word processors such as Microsoft Word or LaTeX. They consist of tags inserted in the text that request, for instance, to start a new paragraph, or to set a phrase in italics or in bold characters. The Rich Text Format (RTF) from Microsoft (2004) and the (La)TeX format designed by Knuth (1986) are widely used markup languages (Table 3.12).

While RTF and LaTeX are used by communities of million of persons, they are not acknowledged as standards. The standard generalized markup language (SGML) takes this place. SGML could have failed and remained a forgotten international

Table 3.12 Some formatting tags in RTF, LaTeX, and HTML

Language	Text in italics	New paragraph	Accented letter *é*
RTF	`{\i text in italics}`	`\par`	`\'e9`
LaTeX	`{\it text in italics}`	`\cr`	`\'{e}`
HTML	`<i>text in italics</i>`	` `	`&eacute;`

initiative. But the Internet and the World Wide Web, which use hypertext markup language (HTML), a specific implementation of SGML, have ensured its posterity. In the next sections, we introduce the **extensible markup language** (XML), which builds on the simplicity of HTML that has secured its success, and extends it to handle any kind of data.

3.4.2 An Outline of XML

XML is a coding framework: a language to define ways of structuring documents. XML can incorporate logical and presentation markups. Logical markups describe the document structure and organization such as, for instance, the title, the sections, and inside the sections, the paragraphs. Presentation markups describe the text appearance and enable users to set a sentence in italic or bold type, or to insert a page break. Contrary to other markup languages, like HTML, XML does not have a predefined set of tags. The programmer defines them together with their meaning.

XML separates the definition of structure instructions from the content – the data. Structure instructions are described in a document type definition (DTD) that models a class of XML documents. DTDs correspond to specific tagsets that enable users to mark up texts. A DTD lists the legal tags and their relationships with other tags, for instance, to define what is a chapter and to verify that it contains a title. Among coding schemes defined by DTDs, there are:

- The extensible hypertext markup language (XHTML), a clean XML implementation of HTML that models the Internet Web pages;
- The Text Encoding Initiative (TEI), which is used by some academic projects to encode texts, in particular, literary works;
- DocBook, which is used by publishers and open-source projects to produce books and technical documents.

A DTD is composed of three kinds of components called elements, attributes, and entities. Comments of DTDs and XML documents are enclosed between the `<!--` and `-->` tags.

Elements

Elements are the logical units of an XML document. They are delimited by surrounding tags. A start tag enclosed between angle brackets precedes the element

content, and an end tag terminates it. End tags are the same as start tags with a / prefix. XML tags must be balanced, which means that an end tag must follow each start tag. Here is a simple example of an XML document inspired by the DocBook specification:

```
<!-- My first XML document -->
<book>
  <title>Language Processing Cookbook</title>
  <author>Pierre Cagné</author>
  <!-- Image to show on the cover -->
  <img></img>
  <text>Here comes the text!</text>
</book>
```

where `<book>` and `</book>` are legal tags indicating, respectively, the start and the end of the book, and `<title>` and `</title>` the beginning and the end of the title. **Empty elements**, such as the image `<img></img>`, can be abridged as `<img/>`. Unlike HTML, XML tags are case sensitive: `<TITLE>` and `<title>` define different elements.

Attributes

An element can have attributes, i.e., a set of properties attached to the element. Let us complement our `book` example so that the `<title>` element has an alignment whose possible values are flush left, right, or center, and a character style taken from underlined, bold, or italics. Let us also indicate where `<img>` finds the image file. The DTD specifies the possible attributes of these elements and the value list among which the actual attribute value will be selected. The actual attributes of an element are supplied as name–value pairs in the element start tag.

Let us name the alignment and style attributes `align` and `style` and set them in boldface characters and centered, and let us store the image file name of the `img` element in the `src` attribute. The markup in the XML document will look like:

```
<title align="center" style="bold">
  Language Processing Cookbook
</title>
<author>Pierre Cagné</author>
<img src="pierre.jpg"/>
```

Entities

Finally, entities correspond to data stored somewhere in a computer. They can be accented characters, symbols, strings as well as text or image files. The programmer declares or defines variables referring to entities in a DTD and uses

Table 3.13 The predefined entities of XML

Symbol	Entity encoding	Meaning
<	`<`	Less than
>	`>`	Greater than
&	`&`	Ampersand
"	`&quot;`	Quotation mark
'	`&apos;`	Apostrophe

them subsequently in XML documents. There are two types of entities: general and parameter. General entities, or simply entities, are declared in a DTD and used in XML document contents. Parameter entities are only used in DTDs. The two types of entities correspond to two different contexts. They are declared and referred to differently. We set aside the parameter entities here; we will examine them in the next section.

An entity is referred to within an XML document by enclosing its name between the start delimiter "`&`" and the end delimiter "`;`", such as `&EntityName;`. The XML parser will substitute the reference with the content of `EntityName` when it is encountered.

XML recognizes a set of predefined or implicitly defined entities that do not need to be declared in a DTD. These entities are used to encode special or accented characters. They can be divided into two groups. The first group consists of five predefined entities (Table 3.13). They correspond to characters used by the XML standard, which cannot be used as is in a document. The second group, called numeric character entities, is used to insert non-ASCII symbols or characters. Character references consist of a Unicode hexadecimal number delimited by "`&#x`" and "`;`", such as `&#xc9;` for *É* and `&#xA9;` for ©.

3.4.3 Writing a DTD

The DTD specifies the formal structure of a document type. It enables an XML parser to determine whether a document is valid. The DTD file contains the description of all the legal elements, attributes, and entities.

Elements

The description of the elements is enclosed between the start and end delimiters `<!ELEMENT` and `>`. It contains the element name and the content model in terms of other elements or reserved keywords (Table 3.14). The content model specifies how the elements appear, their order, and their number of occurrences (Table 3.15). For example:

```
<!ELEMENT book (title, (author | editor)?, img, chapter+)>
<!ELEMENT title (#PCDATA)>
```

Table 3.14 Character types

Character type	Description
PCDATA	Parsed character data. This data will be parsed and must only be text, punctuation, and special characters; no embedded elements
ANY	PCDATA or any DTD element
EMPTY	No content – just a placeholder

Table 3.15 List separators and occurrence indicators

List notation	Description
,	Elements must all appear and be ordered as listed
|	Only one element must appear (exclusive or)
+	Compulsory element (one or more)
?	Optional element (zero or one)
*	Optional element (zero or more)

Table 3.16 Some XML attribute types

Attribute types	Description
CDATA	The string type: any character except <, >, &, ', and "
ID	An identifier of the element unique in the document; ID must begin with a letter, an underscore, or a colon
IDREF	A reference to an identifier
NMTOKEN	String of letters, digits, periods, underscores, hyphens, and colons. It is more restrictive than CDATA; for instance, spaces are not allowed

states that a book consists of a title, a possible author or editor, an image img, and one or more chapters. The title consists of PCDATA, that is, only text with no other embedded elements.

Attributes

Attributes are the possible properties of the elements. Attribute lists are usually defined after the element they refer to. Their description is enclosed between the delimiters <!ATTLIST and >. An attribute list contains:

- The element the attribute refers to
- The attribute name
- The kind of value the attribute may take: a predefined type (Table 3.16) or an enumerated list of values between brackets and separated by vertical bars
- The default value between quotes or a predefined keyword (Table 3.17)

For example:

```
<!ATTLIST title
  style (underlined | bold | italics) "bold"
  align (left | center | right) "left">
```

Table 3.17 Some default value keywords

Predefined default values	Description
`#REQUIRED`	A value must be supplied
`#FIXED`	The attribute value is constant and must be equal to the default value
`#IMPLIED`	If no value is supplied, the processing system will define the value

```
<!ATTLIST author
  style (underlined | bold | italics) #REQUIRED>
```

says that `title` has two attributes, `style` and `align`. The `style` attribute can have three possible values and, if not specified in the XML document, the default value will be `bold`; `author` has one `style` attribute that must be specified in the document.

Entities

Entities enable users to define variables in a DTD. Their declaration is enclosed between the delimiters `<!ENTITY` and `>`. It contains the entity name and the entity content (possibly a sequence):

```
<!ENTITY myEntity "Introduction">
```

This entity can then be used in an XML document with the reference `&myEntity;`. The XML parser will replace all the references it encounters with the value `Introduction`.

Parameter entities are only used in DTDs. They have a "`%`" sign before the entity name, as in

```
<!ENTITY % myParEntity "<!ELEMENT textbody (para)+>">
```

Further references to parameter entities in a DTD use "`%`" and "`;`" as delimiters, such as `%myParEntity;`.

A DTD Example

Let us now suppose that we want to publish cookbooks. We define a document type, and we declare the rules that will form its DTD: a book will consist of a title, a possible author or editor, an image, one or more chapters, and one or more paragraphs in these chapters. Let us then suppose that the main title and the chapter titles can be in bold, in italics, or underlined. Let us finally suppose that the chapter titles can be numbered in Roman or Arabic notation. The DTD elements and attributes are

```
<!ELEMENT book (title, (author | editor)?, img, chapter+)>
<!ELEMENT title (#PCDATA)>
```

```
<!ATTLIST title style (u | b | i) "b">
<!ELEMENT author (#PCDATA)>
<!ATTLIST author style (u | b | i) "i">
<!ELEMENT editor (#PCDATA)>
<!ATTLIST editor style (u | b | i) "i">
<!ELEMENT img EMPTY>
<!ATTLIST img src CDATA #REQUIRED>
<!ELEMENT chapter (subtitle, para+)>
<!ATTLIST chapter number ID #REQUIRED>
<!ATTLIST chapter numberStyle (Arabic | Roman) "Roman">
<!ELEMENT subtitle (#PCDATA)>
<!ELEMENT para (#PCDATA)>
```

The name of the document type corresponds to the **root element**, here `book`, which must be unique.

XML Schema

You probably noticed that the DTD syntax does not fit very well with that of XML. This bothered some people, who tried to make it more compliant. This gave birth to XML Schema, a document definition standard using the XML style. As of today, DTD is still "king," however, XML Schema is gaining popularity. Specifications are available from the Web consortium at http://www.w3.org/XML/Schema.

3.4.4 Writing an XML Document

We shall now write a document conforming to the `book` document type. A complete XML document begins with a prologue, a declaration like this one:

```
<?xml version="1.1" encoding="UTF-8" standalone="no"?>
```

describing the XML version, the encoding used, and whether the document is self-contained or not (standalone). In our example, if we have an external DTD, we must set `standalone` to `no`. This prologue is mandatory from version 1.1 of XML. If not specified, the default encoding is UTF-8.

The document can contain any Unicode character. The encoding refers to how the characters are stored in the file. This has no significance if you only use unaccented characters in the basic Latin set from position 0 to 127. If you type accented characters, the editor will have to save them as UTF-8 codes. In the document above, *Cagné* must be stored as 43 61 67 6E C3 A9, where *é* is corresponds to C3 A9.

If your text editor does not support UTF-8, you will have to enter the accented characters as entities with their Unicode code point, for instance, `&#xC9;` for *É*, or `&#xE9;` for *é*. You may also type the characters *É* or *é* and use your machine's default encoding, such as Latin 1 (ISO-8859-1), Windows-1252, or MacRoman, to

save the XML file. You will have then to declare the corresponding encoding, for instance, `encoding="ISO-8859-1"`.

Then, the document declares the DTD it uses. The DTD can be inside the XML document and enclosed between the delimiters `<!DOCTYPE [` and `]>`, for instance:

```
<!DOCTYPE book [
<!ELEMENT book (title, (author | editor)?, img, chapter+)>
<!ELEMENT title (#PCDATA)>
...
]>
```

Or the DTD can be external to the document, for instance, in a file called `book_definition.dtd`. In this case, `DOCTYPE` indicates its location on the computer using the keyword `SYSTEM`:

```
<!DOCTYPE book SYSTEM "/home/pierre/xml/book_definition.dtd">
```

Finally, we can write the document content. Let us use the XML tags to sketch a very short book. It could look like this:

```
<book>
  <title style="i">Language Processing Cookbook</title>
  <author style="b">Pierre Cagné</author>
  <img src="pierre.jpg"/>
  <chapter number="c1">
    <subtitle>Introduction</subtitle>
    <para>Let&apos;s start doing simple things:
      Collect texts.
    </para>
    <para>First, choose an author you like.</para>
  </chapter>
</book>
```

Once, we have written an XML document, we must check that is **well formed**, which means that it has no syntax errors: the brackets are balanced, the encoding is correct, etc. We must also **validate** it, i.e., check that it conforms to the DTD. This can be done with a variety of parsers available from the Internet, for instance the W3C markup validation service (http://validator.w3.org/). Another easy way to do it is to use the embedded XML parser of any a modern web browser.

3.4.5 Namespaces

In our examples, we used element names that can be part of other DTDs. The string `title`, for instance, is used by XHTML. The XML namespaces is a device to avoid collisions. It is a naming scheme that enables us to define groups of elements and attributes in the same document and prevent name conflicts.

We declare a namespace using the predefined `xmlns` attribute as

```
<my-element xmlns:prefix="URI">
```

It starts a namespace inside `my-element` and its descendants, where `prefix` defines a group of names. Names members of this namespace are preceded by the prefix, as in `prefix:title`. `URI` has the syntax of a web address. However, it is just a unique name; it is never accessed.

Declaring two namespaces in `book`, we can reuse `title` for different purposes:

```
<book
  xmlns:pierre="http://www.cs.lth.se/~pierre"
  xmlns:raymond="http://www.grandecuisine.com">

  <pierre:title style="i">Language Processing Cookbook
  </pierre:title>

  <raymond:title style="i">A French Cookbook
  </raymond:title>
</book>
```

3.4.6 XML and Databases

Although we introduced XML to annotate corpora and narrative documents, many applications use it to store and exchange structured data like records, databases, or configuration files. In fact, creating tabular data in the form of collections of property names and values is easy with XML: we just need to define elements to mark the names (or keys) and the values. Such structures are called dictionaries, like this one:

```
<dict>
  <key>language</key>  <value>German</value>
  <key>currency</key>  <value>euro</value>
</dict>
```

As soon as it was created, XML gained a large popularity among program developers for this purpose. People found it easier to use XML rather than creating their own solution because of its simplicity, its portability, and the wide availability of parsers.

3.5 Further Reading

Many operating systems such as Windows, Mac OS X, and Unix, or programming languages such as Java have adopted Unicode and take the language parameter of a computer into account. Basic lexical methods such as date and currency formatting,

word ordering, and indexing are now supported at the operating system level. Operating systems or programming languages offer toolboxes and routines that you can use in applications.

The Unicode Consortium publishes books, specifications, and technical reports that describe the various aspects of the standard. *The Unicode Standard* (The Unicode Consortium 2012) is the most comprehensive document, while Davis and Whistler (2009) describe in detail the Unicode collation algorithm. Both documents are available in electronic format from the Unicode web site: http://www.unicode.org/. The Unicode Consortium also maintains a public and up-to-date version of the character database (http://www.unicode.org/ucd/). IBM implemented a large library of Unicode components in Java and C++, which are available as open-source software (http://site.icu-project.org/).

HTML and XML markup standards are continuously evolving. Their specifications are available from the World Wide Web consortium (http://www.w3.org/). Finally, a good reference on XML is *Learning XML* (Ray 2003).

Exercises

3.1. Implement UTF-8 that transforms a sequence of code points in a sequence of octets in Prolog.

3.2. Implement a word collation algorithm for English, French, German, or Swedish.

3.3. Modify the DTD in Sect. 3.4.4 so that the cookbook consists of meals instead of chapters, and each meal has an ingredient and a recipe section.

3.4. Modify the DTD in Sect. 3.4.4 to declare the general and parameter entities:

```
<!ENTITY myEntity "Introduction">
<!ENTITY %myEntity "<!ELEMENT textbody (para)+>">
```

Use these entities in the DTD and the document.

3.5. Write a Prolog program that removes the tags from a text encoded in HTML.

3.6. Write a Prolog program that processes a text encoded in HTML: it retains headers (`Hn` tags) and discards the rest.

Chapter 4
Topics in Information Theory and Machine Learning

4.1 Introduction

Information theory underlies the design of codes. Claude Shannon probably started the field with a seminal article (1948), in which he defined a measure of information: the **entropy**. In this chapter, we introduce essential concepts in information theory: entropy, optimal coding, cross entropy, and **perplexity**. Entropy is a very versatile measure of the average information content of symbol sequences and we will explore how it can help us design efficient encodings.

In natural language processing, we often need to determine the category of an object or an observation, such as the part of speech of a word. We will show how we can use entropy to learn decision trees from data sets. This will enable us to build a simple and essential machine-learning algorithm: ID3. We will apply the decision trees we derive from the data sets as classifiers, i.e., devices to classify new data or new objects.

Machine-learning techniques are now instrumental in most areas of natural language processing, and we will use them throughout this book. We will conclude this chapter with the description of three other linear classifiers from among the most popular ones.

4.2 Codes and Information Theory

4.2.1 Entropy

Information theory models a text as a sequence of symbols. Let $x_1, x_2, \ldots, x_N$ be a discrete set of N symbols representing the characters. The **information content** of a symbol is defined as

P.M. Nugues, *Language Processing with Perl and Prolog*, Cognitive Technologies,
DOI 10.1007/978-3-642-41464-0_4, © Springer-Verlag Berlin Heidelberg 2014

$$I(x_i) = -\log_2 p(x_i) = \log_2 \frac{1}{p(x_i)},$$

and it is measured in bits. When the symbols have equal probabilities, they are said to be equiprobable and

$$p(x_1) = p(x_2) = \ldots = p(x_N) = \frac{1}{N}.$$

The information content of x_i is then $I(x_i) = \log_2 N$.

The information content corresponds to the number of bits that are necessary to encode the set of symbols. The information content of the alphabet, assuming that it consists of 26 unaccented equiprobable characters and the space, is $\log_2(26 + 1) = 4.75$, which means that 5 bits are necessary to encode it. If we add 16 accented characters, the uppercase letters, 11 punctuation signs, [, . ; : ? ! " – ( ) '], and the space, we need $(26 + 16) \times 2 + 12 = 96$ symbols. Their information content is $\log_2 96 = 6.58$, and they can be encoded on 7 bits.

The information content assumes that the symbols have an equal probability. This is rarely the case in reality. Therefore this measure can be improved using the concept of entropy, the average information content, which is defined as:

$$H(X) = -\sum_{x \in X} p(x) \log_2 p(x),$$

where X is a random variable over a discrete set of variables, $p(x) = P(X = x), x \in X$, with the convention $0 \log_2 0 = 0$. When the symbols are equiprobable, $H(X) = \log_2 N$. This also corresponds to the upper bound on the entropy value, and for any random variable, we have the inequality $H(X) \leq \log_2 N$.

To evaluate the entropy of printed French, we computed the frequency of the printable French characters in Gustave Flaubert's novel *Salammbô*. Table 4.1 shows the frequency of 26 unaccented letters, the 16 accented or specific letters, and the blanks (spaces).

The entropy of the text restricted to the characters in Table 4.1 is defined as:

$$\begin{aligned} H(X) &= -\sum_{x \in X} p(x) \log_2 p(x). \\ &= -p(A) \log_2 p(A) - p(B) \log_2 p(B) - p(C) \log_2 p(C) - \ldots \\ &\quad -p(Z) \log_2 p(Z) - p(\grave{A}) \log_2 p(\grave{A}) - p(\hat{A}) \log_2 p(\hat{A}) - \ldots \\ &\quad -p(\ddot{U}) \log_2 p(\ddot{U}) - p(\ddot{Y}) \log_2 p(\ddot{Y}) - p(\textit{blanks}) \log_2 p(\textit{blanks}). \end{aligned}$$

If we distinguish between upper- and lowercase letters and if we include the punctuation signs, the digits, and all the other printable characters – ASCII ≥ 32 – the entropy of Gustave Flaubert's *Salammbô* in French is $H(X) = 4.376$.

Table 4.1 Letter frequencies in the French novel *Salammbô* by Gustave Flaubert. The text has been normalized in uppercase letters. The table does not show the frequencies of the punctuation signs or digits

Letter	Frequency	Letter	Frequency	Letter	Frequency	Letter	Frequency
A	42,439	*L*	30,960	*W*	1	*Ë*	6
B	5,757	*M*	13,090	*X*	2,206	*Î*	277
C	14,202	*N*	32,911	*Y*	1,232	*Ï*	66
D	18,907	*O*	22,647	*Z*	413	*Ô*	397
E	71,186	*P*	13,161	*À*	1,884	*Œ*	96
F	4,993	*Q*	3,964	*Â*	605	*Ù*	179
G	5,148	*R*	33,555	*Æ*	9	*Û*	213
H	5,293	*S*	46,753	*Ç*	452	*Ü*	0
I	33,627	*T*	35,084	*É*	7,709	*Ÿ*	0
J	1,220	*U*	29,268	*È*	2,002	Blanks	103,481
K	92	*V*	6,916	*Ê*	898	**Total:**	593,299

Table 4.2 Frequency counts of the symbols

	A	B	C	D	E	F	G	H
Freq	42,439	5,757	14,202	18,907	71,186	4,993	5,148	5,293
Prob	0.25	0.03	0.08	0.11	0.42	0.03	0.03	0.03

Table 4.3 A possible encoding of the symbols on 3 bits

A	B	C	D	E	F	G	H
000	001	010	011	100	101	110	111

4.2.2 Huffman Coding

The information content of the French character set is less than the 7 bits required by equiprobable symbols. Although it gives no clue about an encoding algorithm, it indicates that a more efficient code is theoretically possible. This is what we examine now with Huffman coding, which is a general and simple method to build such a code.

Huffman coding uses variable-length code units. Let us simplify the problem and use only the eight symbols A, B, C, D, E, F, G, and H with the count frequencies in Table 4.2.

The information content of equiprobable symbols is $\log_2 8 = 3$ bits. Table 4.3 shows a possible code with constant-length units.

The idea of Huffman coding is to encode frequent symbols using short code values and rare ones using longer units. This was also the idea of the Morse code, which assigns a single signal to letter E: ., and four signals to letter X: -..-.

This first step builds a Huffman tree using the frequency counts. The symbols and their frequencies are the leaves of the tree. We grow the tree recursively from the leaves to the root. We merge the two symbols with the lowest frequencies into a new node that we annotate with the sum of their frequencies. In Fig. 4.1, this new node

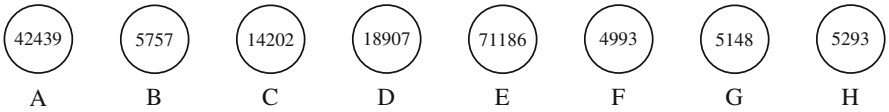


Fig. 4.1 The symbols and their frequencies

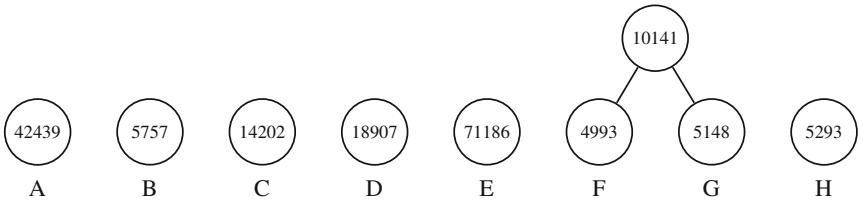


Fig. 4.2 Merging the symbols with the lowest frequencies

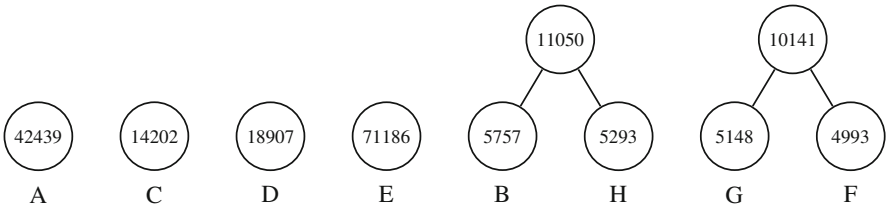


Fig. 4.3 The second iteration

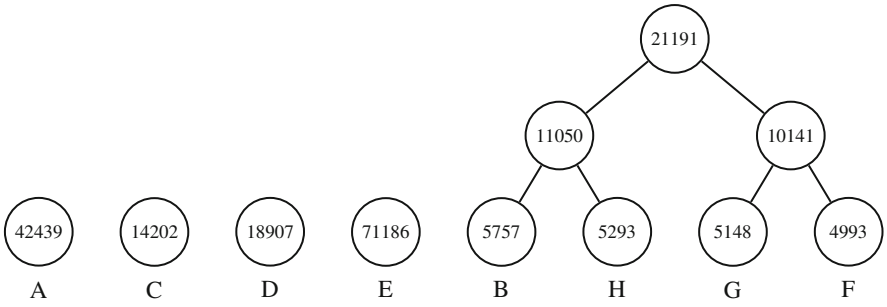


Fig. 4.4 The third iteration

corresponds to the letters F and G with a combined frequency of $4{,}993 + 5{,}148 = 10{,}141$ (Fig. 4.2). The second iteration merges B and H (Fig. 4.3); the third one, (F, G) and (B, H) (Fig. 4.4), and so on (Figs. 4.5–4.8).

The second step of the algorithm generates the Huffman code by assigning a 0 to the left branches and a 1 to the right branches (Table 4.4).

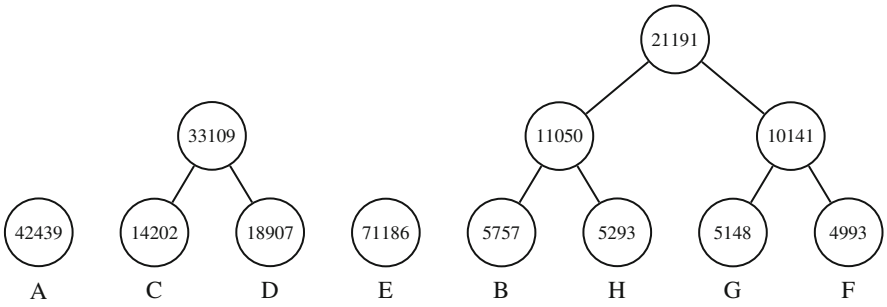


Fig. 4.5 The fourth iteration

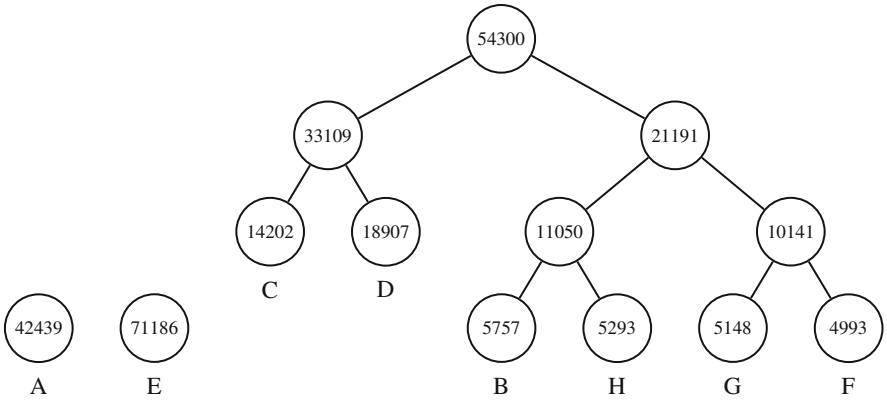


Fig. 4.6 The fifth iteration

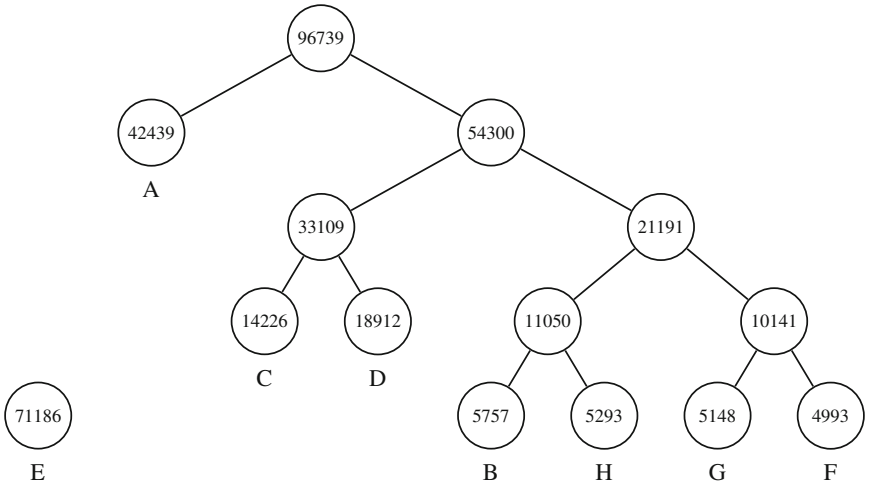


Fig. 4.7 The sixth iteration

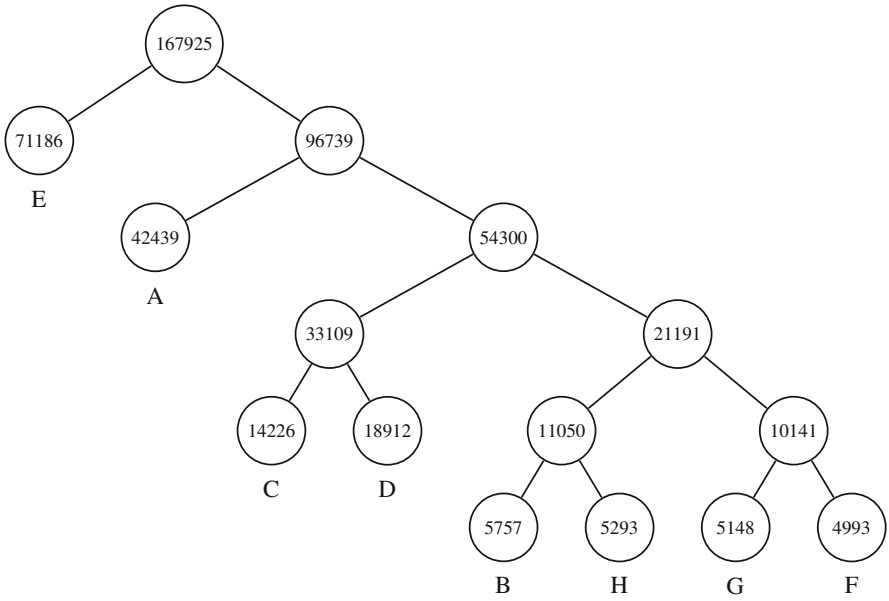


Fig. 4.8 The final Huffman tree

Table 4.4 The Huffman code

A	B	C	D	E	F	G	H
10	11100	1100	1101	0	11111	11110	11101

The average number of bits is the weighted length of a symbol. If we compute it for the data in Table 4.2, it corresponds to:

$$0.25 \times 2 \text{ bit} + 0.03 \times 5 \text{ bit} + 0.08 \times 4 \text{ bit} + 0.11 \times 4 \text{ bit} + 0.42 \times 1 \text{ bit}$$
$$+0.03 \times 5 \text{ bit} + 0.03 \times 5 \text{ bit} + 0.03 \times 5 \text{ bit} = 2.35$$

We can compute the entropy from the counts in Table 4.2. It is defined by the expression:

$$-\Bigg(\frac{42{,}439}{167{,}925}\log_2\frac{42{,}439}{167{,}925} + \frac{5{,}757}{167{,}925}\log_2\frac{5{,}757}{167{,}925} + \frac{14{,}202}{167{,}925}\log_2\frac{14{,}202}{167{,}925} +$$
$$\frac{18{,}907}{167{,}925}\log_2\frac{18{,}907}{167{,}925} + \frac{71{,}186}{167{,}925}\log_2\frac{71{,}186}{167{,}925} + \frac{4{,}993}{167{,}925}\log_2\frac{4{,}993}{167{,}925} +$$
$$\frac{5{,}148}{167{,}925}\log_2\frac{5{,}148}{167{,}925} + \frac{5{,}293}{167{,}925}\log_2\frac{5{,}293}{167{,}925}\Bigg) = 2.31$$

Table 4.5 The entropy is measured on the file itself and the cross entropy is measured with Chapters 1–14 of Gustave Flaubert's *Salammbô* taken as the model

	Entropy	**Cross entropy**	**Difference**
Salammbô, chapters 1–14, training set	4.37745	4.37745	0.0
Salammbô, chapter 15, test set	4.31793	4.33003	0.01210
Notre Dame de Paris, test set	4.43696	4.45590	0.01894
Nineteen Eighty-Four, test set	4.35922	4.80767	0.44845

We can see that although the Huffman code reduces the average number of bits from 3 to 2.35, it does not reach the limit defined by entropy, which is, in our example, 2.31.

4.2.3 Cross Entropy

Let us now compare the letter frequencies between two parts of *Salammbô*, then between *Salammbô* and another text in French or in English. The symbol probabilities will certainly be different. Intuitively, the distributions of two parts of the same novel are likely to be close, further apart between *Salammbô* and another French text from the twenty-first century, and even further apart with a text in English. This is the idea of cross entropy, which compares two probability distributions.

In the cross entropy formula, one distribution is referred to as the model. It corresponds to data on which the probabilities have been trained. Let us name it m with the distribution $m(x_1), m(x_2), \ldots, m(x_N)$. The other distribution, p, corresponds to the test data: $p(x_1)$, $p(x_2), \ldots, p(x_N)$. The cross entropy of m on p is defined as:

$$H(p, m) = -\sum_{x \in X} p(x) \log_2 m(x).$$

Cross entropy quantifies the average surprise of the distribution when exposed to the model. We have the inequality $H(p) \leq H(p, m)$ for any other distribution m with equality if and only if $m(x_i) = p(x_i)$ for all i. The difference $H(p, m) - H(p)$ is a measure of the relevance of the model: the closer the cross entropy, the better the model.

To see how the probability distribution of Flaubert's novel could fare on other texts, we trained a model on the first 14 chapters of *Salammbô*, and we applied it to the last chapter of *Salammbô* (Chap. 15), to Victor Hugo's *Notre Dame de Paris*, both in French, and to *Nineteen Eighty-Four* by George Orwell in English. The data in Table 4.5 conform to our intuition. They show that the first chapters of *Salammbô* are a better model of the last chapter of *Salammbô* than of *Notre Dame de Paris*, and even better than of *Nineteen Eighty-Four*.

Table 4.6 The perplexity and cross perplexity of texts measured with Chapters 1–14 of Gustave Flaubert's *Salammbô* taken as the model

	Perplexity	**Cross perplexity**
Salammbô, chapters 1–14, training set	20.78	20.78
Salammbô, chapter 15, test set	19.94	20.11
Notre Dame de Paris, test set	21.66	21.95
Nineteen Eighty-Four, test set	20.52	28.01

4.2.4 Perplexity and Cross Perplexity

Perplexity is an alternate measure of information that is mainly used by the speech processing community. Perplexity is simply defined as $2^{H(X)}$. The cross perplexity is defined similarly as $2^{H(p,m)}$.

Although perplexity does not bring anything new to entropy, it presents the information differently. Perplexity reflects the averaged number of choices of a random variable. It is equivalent to the size of an imaginary set of equiprobable symbols, which is probably easier to understand.

Table 4.6 shows the perplexity and cross perplexity of the same texts measured with Chaps. 1–14 of Gustave Flaubert's *Salammbô* taken as the model.

4.3 Entropy and Decision Trees

Decision trees are useful devices to classify objects into a set of classes. In this section, we describe what they are and see how entropy can help us learn – or induce – automatically decision trees from a set of data. The algorithm, which resembles a reverse Huffman encoding, is one of the simplest machine-learning techniques.

4.3.1 Machine Learning

Machine learning considers collections of objects or observations, where each object is defined by a set of attributes, SA. Each attribute has a set of possible values called the attribute domain. Table 4.7 from Quinlan (1986) shows a collection of objects, where:

$$SA = \{Outlook, Temperature, Humidity, Windy\},$$

with the following respective domains:

- $\text{dom}(Outlook) = \{sunny, overcast, rain\}$,
- $\text{dom}(Temperature) = \{hot, mild, cool\}$,

Table 4.7 A set of object members of two classes: N and P. Here the objects are weather observations (After Quinlan (1986))

	Attributes				
Object	**Outlook**	**Temperature**	**Humidity**	**Windy**	**Class**
1	Sunny	Hot	High	False	N
2	Sunny	Hot	High	True	N
3	Overcast	Hot	High	False	P
4	Rain	Mild	High	False	P
5	Rain	Cool	Normal	False	P
6	Rain	Cool	Normal	True	N
7	Overcast	Cool	Normal	True	P
8	Sunny	Mild	High	False	N
9	Sunny	Cool	Normal	False	P
10	Rain	Mild	Normal	False	P
11	Sunny	Mild	Normal	True	P
12	Overcast	Mild	High	True	P
13	Overcast	Hot	Normal	False	P
14	Rain	Mild	High	True	N

- $\text{dom}(Humidity) = \{normal, high\}$,
- $\text{dom}(Windy) = \{true, false\}$.

Each object is member of a class, N or P in this data set.

Machine-learning algorithms can be categorized along two main lines: supervised and unsupervised classification. In supervised machine-learning, each object belongs to a predefined class, here P and N. This is the technique we will use in the induction of decision trees, where we will automatically create a tree from a training set, here the examples in Fig. 4.7. Once the tree is induced, it will be able to predict the class of examples taken outside the training set.

Machine-learning techniques make it possible to build programs that organize and classify data, like annotated corpora, without the chore of manually explicating the rules behind this organization or classification. Because of the availability of massive volumes of data, they have become extremely popular in all the fields of language processing. They are now instrumental in many NLP applications and tasks, including part-of-speech tagging, parsing, semantic role labeling, or coreference solving, that we will describe in the next chapters of this book.

4.3.2 Decision Trees

A decision tree is a tool to classify objects such as those in Table 4.7. The nodes of a tree represent conditions on the attributes of an object, and a node has as many branches as its corresponding attribute has values. An object is presented at the root of the tree, and the values of its attributes are tested by the tree nodes from the root

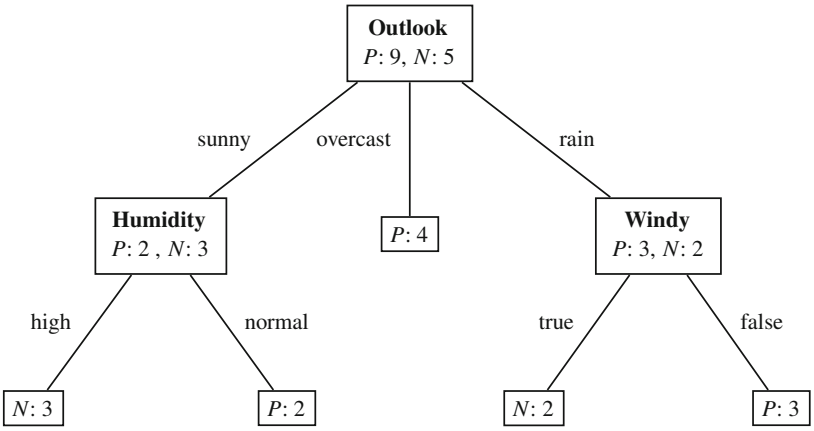


Fig. 4.9 A decision tree classifying the objects in Table 4.7. Each node represents an attribute with the number of objects in the classes P and N. At the start of the process, the collection has nine objects in class P and five in class N. The classification is done by testing the attribute values of each object in the nodes until a leaf is reached, where all the objects belong to one class, P or N (After Quinlan (1986))

down to a leaf. The leaves return a decision, which is the object class or probabilities to be the member of a class.

Figure 4.9 shows a decision tree that correctly classifies all the objects in the set shown in Table 4.7 (Quinlan 1986).

4.3.3 Inducing Decision Trees Automatically

It is possible to design many trees that classify successfully the objects in Table 4.7. The tree in Fig. 4.9 is interesting because it is efficient: a decision can be made with a minimal number of tests.

An efficient decision tree can be induced from a set of examples, members of mutually exclusive classes, using an entropy measure. We will describe the induction algorithm using two classes of p positive and n negative examples, although it can be generalized to any number of classes. As we saw earlier, each example is defined by a finite set of attributes, SA.

At the root of the tree, the condition, and hence the attribute, must be the most discriminating, that is, have branches gathering most positive examples while others gather negative examples. A perfect attribute for the root would create a partition with subsets containing only positive or negative examples. The decision would then be made with one single test. The ID3 (Quinlan 1986) algorithm uses this idea and the entropy to select the best attribute to be this root. Once we have the root, the initial set is split into subsets according to the branching conditions that correspond

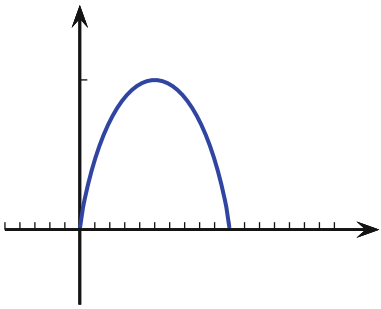


Fig. 4.10 The binary entropy function: $-x \log_2 x - (1-x) \log_2 (1-x)$

to the values of the root attribute. Then, the algorithm determines recursively the next attributes of the resulting nodes.

ID3 defines the information gain of an attribute as the difference of entropy before and after the decision. It measures its separating power: the more the gain, the better the attribute. At the root, the entropy of the collection is constant. As defined previously, for a two-class set of p positive and n negative examples, it is:

$$I(p,n) = -\frac{p}{p+n} \log_2 \frac{p}{p+n} - \frac{n}{p+n} \log_2 \frac{n}{p+n}.$$

Figure 4.10 shows this binary entropy function with $x = \frac{p}{p+n}$, for x ranging from 0 to 1. The function attains its maximum of 1 at $x = 0.5$, when $p = n$ and there are as many positive as negative examples in the set, and its minimum of 0 at $x = 0$ and $x = 1$, when $p = 0$ or $n = 0$ and the examples in the set are either all positive or all negative.

An attribute A with v possible values $\{A_1, A_2, \ldots, A_v\}$ creates a partition of the collection into v subsets, where each subset corresponds to one value of A and contains p_i positive and n_i negative examples. The entropy of a subset is $I(p_i, n_i)$ and the weighted average of entropies of the partition created by A is:

$$E(A) = \sum_{i=1}^{v} \frac{p_i + n_i}{p+n} I\left(\frac{p_i}{p_i + n_i}, \frac{n_i}{p_i + n_i}\right).$$

The information gain is defined as $Gain(A) = I(p,n) - E(A)$ (or $I_{\text{before}} - I_{\text{after}}$). We would reach the maximum possible gain with an attribute that creates subsets containing examples that are either all positive or all negative. In this case, the entropy of the nodes below the root would be 0.

For the tree in Fig. 4.9, let us compute the information gain of attribute *Outlook*. The entropy of the complete data set is:

$$I(p,n) = -\frac{9}{14} \log_2 \frac{9}{14} - \frac{5}{14} \log_2 \frac{5}{14} = 0.940.$$

Outlook has three values: *sunny*, *overcast*, and *rain*. The entropies of the respective subsets created by these values are:

$$\begin{aligned} &\textit{sunny}: && I(p_1, n_1) = -\frac{2}{5}\log_2\frac{2}{5} - \frac{3}{5}\log_2\frac{3}{5} = 0.971. \\ &\textit{overcast}: && I(p_2, n_2) = 0. \\ &\textit{rain}: && I(p_3, n_3) = -\frac{3}{5}\log_2\frac{3}{5} - \frac{2}{5}\log_2\frac{2}{5} = 0.971. \end{aligned}$$

Thus

$$E(\textit{Outlook}) = \frac{5}{14}I(p_1, n_1) + \frac{4}{14}I(p_2, n_2) + \frac{5}{14}I(p_3, n_3) = 0.694.$$

Gain(*Outlook*) is then 0.940 – 0.694 = 0.246, which is the highest for the four attributes. *Gain*(*Temperature*), *Gain*(*Humidity*), and *Gain*(*Windy*) are computed similarly.

The algorithm to build the decision tree is simple. The information gain is computed on the data set for all attributes, and the attribute with the highest gain:

$$A = \arg\max_{a \in SA} I(n, p) - E(a).$$

is selected to be the root of the tree. The data set is then split into v subsets $\{N_1, \ldots, N_v\}$, where the value of A for the objects in N_i is A_i, and for each subset, a corresponding node is created below the root. This process is repeated recursively for each node of the tree with the subset it contains until all the objects of the node are either positive or negative. For a training set of N instances each having M attributes, Quinlan (1986) showed that ID3's complexity to generate a decision tree is $O(NM)$.

4.4 Classification Using Linear Methods

4.4.1 Linear Classifiers

Decision trees are simple and efficient devices to design classifiers. Together with the information gain, they enabled us to induce optimal trees from a set of examples and to deal with symbolic values such as *sunny*, *hot*, and *high*.

Linear classifiers are another set of techniques that have the same purpose. As with decision trees, they produce a function splitting a set of objects into two or more classes. This time, however, the objects will be represented by a vector of numerical parameters. Such parameters are often called **features**. In the next sections, we examine linear classification methods in an n-dimensional space, where the dimension of the vector space is equal to the number of features used to characterize the objects.

Table 4.8 The frequency of *A* in the chapters of *Salammbô* in English and French. Letters have been normalized in uppercase and duplicate spaces removed

	French		**English**	
Chapter	# Characters	# *A*	# Characters	# *A*
Chapter 1	36,961	2,503	35,680	2,217
Chapter 2	43,621	2,992	42,514	2,761
Chapter 3	15,694	1,042	15,162	990
Chapter 4	36,231	2,487	35,298	2,274
Chapter 5	29,945	2,014	29,800	1,865
Chapter 6	40,588	2,805	40,255	2,606
Chapter 7	75,255	5,062	74,532	4,805
Chapter 8	37,709	2,643	37,464	2,396
Chapter 9	30,899	2,126	31,030	1,993
Chapter 10	25,486	1,784	24,843	1,627
Chapter 11	37,497	2,641	36,172	2,375
Chapter 12	40,398	2,766	39,552	2,560
Chapter 13	74,105	5,047	72,545	4,597
Chapter 14	76,725	5,312	75,352	4,871
Chapter 15	18,317	1,215	18,031	1,119
Total	619,431	42,439	608,230	39,056

4.4.2 Choosing a Data Set

To illustrate linear classification in a two-dimensional space, we will use *Salammbô* again in its original French version and in an English translation, and we will try to predict automatically the language of the version. As features, we will use the letter counts in each chapter: how many *A*, *B*, *C*, etc. The distribution of letters is different across both languages, for instance, *W* is quite frequent in English and rare in French. This makes it possible to use distribution models as an elementary method to identify the language of a text.

Although a more realistic language guesser would use all the letters of the alphabet, we will restrict it to *A*. We will count the total number of characters and the frequency of *A*s in each of the 15 chapters and try to derive a model from the data. Table 4.8 shows these counts in French and in English.

4.5 Linear Regression

Before we try to discriminate between French and English, let us examine how we can model the distribution of the letters in one language.

Figure 4.11 shows the plot of data in Table 4.8, where each point represents the letter counts in one of the 15 chapters. The x-axis corresponds to the total count of letters in the chapter, and the y-axis, the count of *A*s. We can see from the figure that

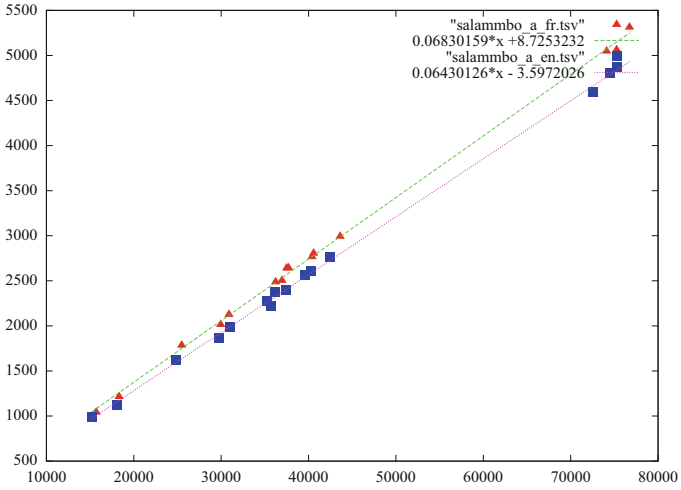


Fig. 4.11 Plot of the frequencies of A, y, versus the total character counts, x, in the 15 chapters of *Salammbô*. *Squares* correspond to the English version and *triangles* to the French original

the points in both languages can be fitted quite precisely to two straight lines. This fitting process is called a linear regression, where a line equation is given by:

$$y = mx + b.$$

To determine the m and b coefficients, we will minimize a fitting error between the point distribution given by the set of q observations: $\{(x_i, y_i)\}_{i=1}^{q}$ and a perfect linear alignment given by the set $\{(x_i, f(x_i))\}_{i=1}^{q}$, where $f(x_i) = mx_i + b$. In our data set, we have 15 observations from each chapter in *Salammbô*, and hence $q = 15$.

4.5.1 Least Squares

The least squares method is probably the most common technique used to model the fitting error and estimate m and b. This error is defined as the sum of the squared errors (SSE) over all the q points (Legendre 1805):

$$\begin{aligned} SSE(m, b) &= \sum_{i=1}^{q} (y_i - f(x_i))^2, \\ &= \sum_{i=1}^{q} (y_i - (mx_i + b))^2. \end{aligned}$$

Ideally, all the points would be aligned and this sum would be zero. This is rarely the case in practice, and we fall back to an approximation that minimizes it.

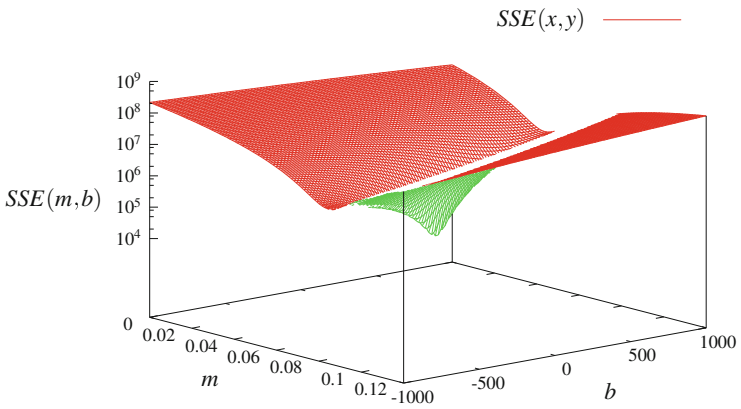


Fig. 4.12 Plot of $SSE(m,b)$ applied to the 15 chapters of the English version of *Salammbô*

Figure 4.12 shows the plot of $SSE(m,b)$ applied to the 15 chapters of the English version of *Salammbô*. Using a logarithmic scale, the surface shows a visible minimum somewhere between 0.6 and 0.8 for m and close to 0 for b. Let us now compute precisely these values.

We know from differential calculus that we reach the minimum of $SSE(m,b)$ when its partial derivatives over m and b are zero:

$$\begin{aligned}
\frac{\partial SSE(m,b)}{\partial m} &= \sum_{i=1}^{q} \frac{\partial}{\partial m}(y_i - (mx_i + b))^2 = -2\sum_{i=1}^{q} x_i(y_i - (mx_i + b)) = 0.\\
\frac{\partial SSE(m,b)}{\partial b} &= \sum_{i=1}^{q} \frac{\partial}{\partial b}(y_i - (mx_i + b))^2 = -2\sum_{i=1}^{q} (y_i - (mx_i + b)) = 0.
\end{aligned}$$

We obtain then:

$$m = \frac{\sum_{i=1}^{q} x_i y_i - q\overline{x}\,\overline{y}}{\sum_{i=1}^{q} x_i^2 - q\overline{x}^2} \quad \text{and} \quad b = \overline{y} - m\overline{x},$$

with

$$\overline{x} = \frac{1}{q}\sum_{i=1}^{q} x_i \quad \text{and} \quad \overline{y} = \frac{1}{q}\sum_{i=1}^{q} y_i.$$

Using these formulas, we find the two regression lines for French and English:

$$\begin{aligned} \text{French: } & y = 0.0683x + 8.7253 \\ \text{English: } & y = 0.0643x - 3.5972 \end{aligned}$$

Least Absolute Deviation

An alternative to the least squares is to minimize the sum of the absolute errors (SAE) (Boscovich 1770, Livre V, note):

$$SAE(m, b) = \sum_{i=1}^{q} |y_i - f(x_i)|.$$

The corresponding minimum value is called the least absolute deviation (LAD). Solving methods to find this minimum use linear programming. Their description falls outside the scope of this book.

Notations in an n-Dimensional Space

Up to now, we have formulated the regression problem with two parameters: the letter count and the count of *A*s. In most practical cases, we will have a much larger set. To describe algorithms applicable to any number of parameters, we need to extend our notation to a general n-dimensional space. Let us introduce it now.

In an n-dimensional space, it is probably easier to describe linear regression as a prediction technique: given input parameters in the form of a feature vector, predict the output value. In the *Salammbô* example, the input would be the number of letters in a chapter, and the output, the number of *A*s.

In a typical data set such as the one shown in Table 4.8, we have:

The input parameters: These parameters describe the observations we will use to predict an output. They are also called **feature vectors**, and we denote them $(1, x_1, x_2, \ldots, x_{n-1})$ or $\mathbf{x}$. The first parameter is set to 1 to make the computation easier. In the *Salammbô* example, this corresponds to the letter count in a chapter, for example: (1,36,961) in Chapter 1 in French.

The output value: Each output represents the answer to a feature vector, and we denote it y, when we observe it, or $\hat{y}$, when we predict it using the regression line. In *Salammbô*, in French, the count of *A*s is $y = 2,503$ in Chapter 1, and the predicted value using the regression line is $\hat{y} = 0.0683 \times 36,961 + 8.7253 = 2,533.22$.

The squared error: The squared error is the squared difference between the observed value and the prediction, $(y - \hat{y})^2$. In *Salammbô*, the squared error for Chapter 1 is $(2,503 - 2,533.22)^2 = 30.22^2 = 913.26$.

As we have seen, to compute the regression line, the least-squares method minimizes the sum of the squared errors for all the observations (here all the chapters). It is defined by its coefficients m and b in a two-dimensional space. In an n-dimensional space, we have:

The weight vector: The equivalent of a regression line when $n > 2$ is a hyperplane with a coefficient vector denoted $(w_0, w_1, w_2, \ldots, w_{n-1})$ or $\mathbf{w}$. These coefficients are usually called the weights. They correspond to (b, m) when $n = 2$. In *Salammbô*, the weight vector would be $(8.7253, 0.0683)$ for French and $(-3.5972, 0.0643)$ for English.

The intercept: This is the first weight w_0 of the weight vector. It corresponds to b when $n = 2$.

The hyperplane equation is given by the dot product of the weights by the feature variables. It is defined as:

$$y = \mathbf{w} \cdot \mathbf{x} = \sum_{i=0}^{n-1} w_i x_i,$$

where $\mathbf{w} = (w_0, w_1, w_2, \ldots, w_{n-1})$, $\mathbf{x} = (x_0, x_1, x_2, \ldots, x_{n-1})$, and $x_0 = 1$.

4.5.2 The Gradient Descent

Using partial derivatives, we have been able to find an analytical solution to the regression line. We will now introduce the gradient descent, a generic optimization method that uses a series of successive approximations instead. We will apply this technique to solve the least squares as well as the classification problems we will see in the next section.

The gradient descent (Cauchy 1847) is a numerical method to find a global or local minimum of a function:

$$\begin{aligned} y &= f(x_0, x_1, x_2, \ldots, x_n), \\ &= f(\mathbf{x}), \end{aligned}$$

even when there is no analytical solution.

As we can see on Fig. 4.12, the sum of squared errors has a minimum. This a general property of the least squares, and the idea of the gradient descent is to derive successive approximations in the form of a sequence of points $(\mathbf{x}_p)$ to find it. At each iteration, the current point will move one step down to the minimum. For a function f, this means that we will have the inequalities:

$$f(\mathbf{x}_1) > f(\mathbf{x}_2) > \ldots > f(\mathbf{x}_k) > f(\mathbf{x}_{k+1}) > \ldots > min.$$

Now given a point $\mathbf{x}$, how can we find the next point of the iteration? The steps in the gradient descent are usually small and we can define the points in the neighborhood of $\mathbf{x}$ by $\mathbf{x} + \mathbf{v}$, where $\mathbf{v}$ is a vector of $\mathbb{R}^n$ and $||\mathbf{v}||$ is small. So the problem of gradient descent can be reformulated as: given $\mathbf{x}$, find $\mathbf{v}$ subject to $f(\mathbf{x}) > f(\mathbf{x} + \mathbf{v})$.

As $||\mathbf{v}||$ is small, we can approximate $f(\mathbf{x} + \mathbf{v})$ using a Taylor expansion limited to the first derivatives:

$$f(\mathbf{x} + \mathbf{v}) \approx f(\mathbf{x}) + \mathbf{v} \cdot \nabla f(\mathbf{x}),$$

where the gradient defined as:

$$\nabla f(x_0, x_1, x_2, \ldots, x_n) = (\frac{\partial f}{\partial x_0}, \frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n})$$

is a direction vector corresponding to the steepest slope.

We obtain the steepest descent (respectively, ascent) when we choose $\mathbf{v}$ collinear to $\nabla f(\mathbf{x})$: $\mathbf{v} = -\alpha \nabla f(\mathbf{x})$ (respectively, $\mathbf{v} = \alpha \nabla f(\mathbf{x})$) with $\alpha > 0$. We have then:

$$f(\mathbf{x} - \alpha \nabla f(\mathbf{x})) \approx f(\mathbf{x}) - \alpha ||\nabla f(\mathbf{x})||^2,$$

and thus the inequality:

$$f(\mathbf{x}) > f(\mathbf{x} - \alpha \nabla f(\mathbf{x})).$$

This inequality enables us to write a recurrence relation between the steps:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \nabla f(\mathbf{x}_k)$$

and find a step sequence to the minimum, where α_k is a small positive number called the step size or learning rate. It can be constant over all the descent or change at each step. The convergence stops when $||\nabla f(\mathbf{x})||$ is less than a predefined threshold. This convergence is generally faster if the learning rate decreases over the iterations.

4.5.3 The Gradient Descent and Linear Regression

For a data set, *DS*, we find the minimum of the sum of squared errors and the coefficients of the regression equation through a walk down the surface using the recurrence relation above. Let us compute the gradient in a two-dimensional space first and then generalize it to multidimensional space.

In a Two-Dimensional Space

To make the generalization easier, let us rename the straight line coefficients (b, m) in $y = mx + b$ as (w_0, w_1). We want then to find the regression line:

$$\hat{y} = w_0 + w_1 x_1$$

given a data set DS of q examples: $DS = \{(1, x_1^j, y^j) | j : 1..q\}$, where the error is defined as:

$$\begin{aligned} SSE(w_0, w_1) &= \sum_{j=1}^{q} (y^j - \hat{y}^j)^2 \\ &= \sum_{j=1}^{q} (y^j - (w_0 + w_1 x_1^j))^2. \end{aligned}$$

The gradient of this two-dimensional equation $\nabla SSE(\mathbf{w})$ is:

$$\begin{aligned} \frac{\partial SSE(w_0, w_1)}{\partial w_0} &= -2 \sum_{j=1}^{q} (y^j - (w_0 + w_1 x_1^j)) \\ \frac{\partial SSE(w_0, w_1)}{\partial w_1} &= -2 \sum_{j=1}^{q} x_1^j \times (y^j - (w_0 + w_1 x_1^j)). \end{aligned}$$

From this gradient, we can now compute the iteration step. With q examples and a learning rate of $\frac{\alpha}{2q}$, inversely proportional to the number of examples, we have:

$$\begin{aligned} w_0 &\leftarrow w_0 + \frac{\alpha}{q} \cdot \sum_{j=1}^{q} (y^j - (w_0 + w_1 x_1^j)) \\ w_1 &\leftarrow w_1 + \frac{\alpha}{q} \cdot \sum_{j=1}^{q} x_1^j \times (y^j - (w_0 + w_1 x_1^j)). \end{aligned}$$

In the iteration above, we compute the gradient as a sum over all the examples before we carry out one update of the weights. This technique is called the **batch gradient descent**. An alternate technique is to go through DS and compute an update with each example:

$$\begin{aligned} w_0 &\leftarrow w_0 + \alpha \cdot (y^j - (w_0 + w_1 x_1^j)) \\ w_1 &\leftarrow w_1 + \alpha \cdot x_1^j \cdot (y^j - (w_0 + w_1 x_1^j)). \end{aligned}$$

The examples are usually selected randomly from *DS*. This is called the **stochastic gradient descent** or **online learning**.

The duration of the descent is measured in **epochs**, where an epoch is the period corresponding to one iteration over the complete data set: the q examples. The stochastic variant often has a faster convergence.

N-Dimensional Space

In an n-dimensional space, we want to find the regression hyperplane:

$$\hat{y} = w_0 + w_1 x_1 + w_2 x_2 + \ldots + w_n x_n,$$

given a data set *DS* of q examples: $DS = \{(1, x_1^j, x_2^j, \ldots, x_n^j, y^j) | j : 1..q\}$, where the error is defined as:

$$\begin{aligned} SSE(w_0, w_1, \ldots, w_n) &= \sum_{j=1}^{q} (y^j - \hat{y}^j)^2 \\ &= \sum_{j=1}^{q} (y^j - (w_0 + w_1 x_1^j + w_2 x_2^j + \ldots + w_n x_n^j))^2. \end{aligned}$$

To simplify the computation of partial derivatives, we introduce the parameter $x_0^j = 1$ so that:

$$SSE(w_0, w_1, \ldots, w_n) = \sum_{j=1}^{q} (y^j - (w_0 x_0^j + w_1 x_1^j + w_2 x_2^j + \ldots + w_n x_n^j))^2.$$

The gradient of *SSE* is:

$$\frac{\partial SSE}{\partial w_i} = -2 \sum_{j=1}^{q} x_i^j \times (y^j - (w_0 x_0^j + w_1 x_1^j + w_2 x_2^j + \ldots + w_n x_n^j)).$$

In the batch version, the iteration step considers all the examples in *DS*:

$$w_i \leftarrow w_i + \frac{\alpha}{q} \cdot \sum_{j=1}^{q} x_i^j \cdot (y^j - (w_0 x_0^j + w_1 x_1^j + w_2 x_2^j + \ldots + w_n x_n^j)).$$

In the stochastic version, we carry out the updates using one example at a time.

4.6 Linear Classification

4.6.1 An Example

We will now use the data set in Table 4.8 to describe classification techniques that split the texts into French or English. If we examine it closely, Fig. 4.11 shows that we can draw a straight line between the two regression lines to separate the two classes. This is the idea of linear classification. From a data representation in a Euclidian space, classification will consist in finding a line:

$$w_0 + w_1 x + w_2 y = 0$$

separating the plane into two half-planes defined by the inequalities:

$$w_0 + w_1 x + w_2 y > 0$$

and

$$w_0 + w_1 x + w_2 y < 0.$$

These inequalities mean that the points belonging to one class of the data set are on one side of the separating line and the others are on the other side.

In Table 4.8 and Fig. 4.11, the chapters in French have a steeper slope that the corresponding ones in English. The points representing the French chapters will then be above the separating line. Let us write the inequalities that reflect this and set w_2 to 1 to normalize them. The line we are looking for will have the property:

$$\begin{aligned} y_i &> w_0 + w_1 x_i \quad \text{for the set of points: } \{(x_i, y_i) | (x_i, y_i) \in \text{French}\} \text{ and} \\ y_i &< w_0 + w_1 x_i \quad \text{for the set of points: } \{(x_i, y_i) | (x_i, y_i) \in \text{English}\}, \end{aligned}$$

where x is the total count of letters in a chapter and y, the count of *A*s. In total, we will have 30 inequalities, 15 for French and 15 for English shown in Table 4.9. Any weight vector $\mathbf{w} = (w_0, w_1)$ that satisfies all of them will define a classifier correctly separating the chapters into two classes: French or English.

Let us represent graphically the inequalities in Table 4.9 and solve the system in the two-dimensional space defined by w_0 and w_1. Figure 4.13 shows a plot with the two first chapters, where w_1 is the abscissa and w_0, the ordinate. Each inequality defines a half-plane that restricts the set of possible weight values. The four inequalities delimit the solution region in white, where the two upper lines are constraints applied by the two chapters in French and the two below by their English translations.

Table 4.9 Inequalities derived from Table 4.8 for the 15 chapters in *Salammbô* in French and English

Chapter	French	English
1	$2{,}503 > w_0 + 36{,}961w_1$	$2{,}217 < w_0 + 35{,}680w_1$
2	$2{,}992 > w_0 + 43{,}621w_1$	$2{,}761 < w_0 + 42{,}514w_1$
3	$1{,}042 > w_0 + 15{,}694w_1$	$990 < w_0 + 15{,}162w_1$
4	$2{,}487 > w_0 + 36{,}231w_1$	$2{,}274 < w_0 + 35{,}298w_1$
5	$2{,}014 > w_0 + 29{,}945w_1$	$1{,}865 < w_0 + 29{,}800w_1$
6	$2{,}805 > w_0 + 40{,}588w_1$	$2{,}606 < w_0 + 40{,}255w_1$
7	$5{,}062 > w_0 + 75{,}255w_1$	$4{,}805 < w_0 + 74{,}532w_1$
8	$2{,}643 > w_0 + 37{,}709w_1$	$2{,}396 < w_0 + 37{,}464w_1$
9	$2{,}126 > w_0 + 30{,}899w_1$	$1{,}993 < w_0 + 31{,}030w_1$
10	$1{,}784 > w_0 + 25{,}486w_1$	$1{,}627 < w_0 + 24{,}843w_1$
11	$2{,}641 > w_0 + 37{,}497w_1$	$2{,}375 < w_0 + 36{,}172w_1$
12	$2{,}766 > w_0 + 40{,}398w_1$	$2{,}560 < w_0 + 39{,}552w_1$
13	$5{,}047 > w_0 + 74{,}105w_1$	$4{,}597 < w_0 + 72{,}545w_1$
14	$5{,}312 > w_0 + 76{,}725w_1$	$4{,}871 < w_0 + 75{,}352w_1$
15	$1{,}215 > w_0 + 18{,}317w_1$	$1{,}119 < w_0 + 18{,}031w_1$

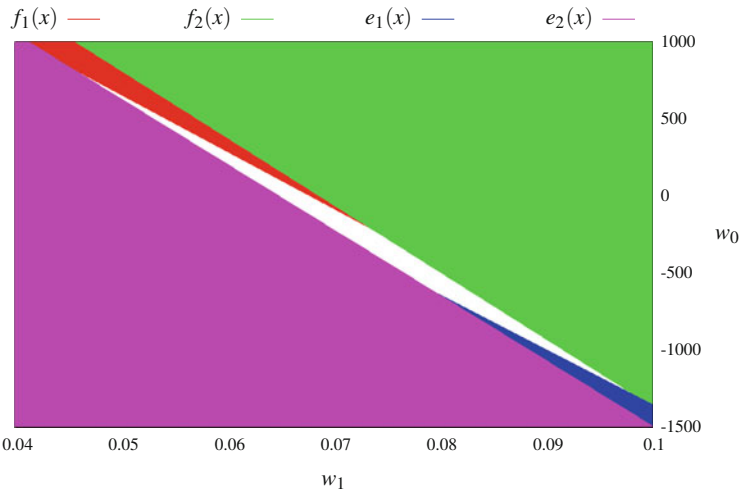


Fig. 4.13 A graphical representation of the inequality system restricted to the two first chapters in French, f_1 and f_2, and in English, e_1 and e_2. We can use any point coordinates in the *white region* as parameters of the line to separate these two chapters

Figure 4.14 shows the plot for all the chapters. The remaining inequalities shrink even more the polygonal region of possible values. The point coordinates (w_1, w_0) in this region, as, for example, $(0.066, 0)$ or $(0.067, -20)$, will satisfy all the inequalities and correctly separate the 30 observations into two classes: 15 chapters in French and 15 in English.

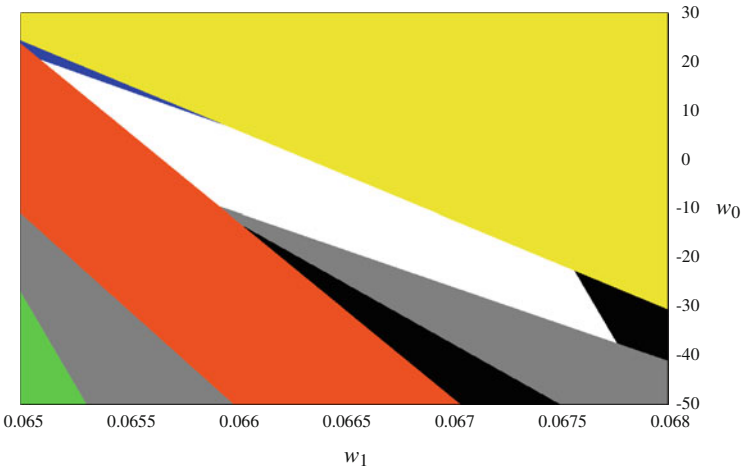


Fig. 4.14 A graphical representation of the inequality system with all the chapters. The point coordinates in the *white polygonal region* correspond to weights vectors (w_1, w_0) defining a separating line for all the chapters

4.6.2 Classification in an N-Dimensional Space

In the example above, we used a set of two-dimensional points, (x_i, y_i) to represent our observations. This process can be generalized to vectors in a space of dimension n. The separator will then be a hyperplane of dimension $n - 1$. In a space of dimension 2, a hyperplane is a line; in dimension 3, a hyperplane is a plane of dimension 2, etc. In an n-dimensional space, the inequalities defining the two classes will be:

$$w_0 + w_1 x_1 + w_2 x_2 + \ldots + w_n x_n > 0$$

and

$$w_0 + w_1 x_1 + w_2 x_2 + \ldots + w_n x_n < 0,$$

where each observation is described by a feature vector $\mathbf{x}$.

The sums in the inequalities correspond to the dot product of the weight vector, $\mathbf{w}$, by the the feature vector, $\mathbf{x}$, defined as

$$\mathbf{w} \cdot \mathbf{x} = \sum_{i=0}^{n} w_i x_i,$$

where $\mathbf{w} = (w_0, w_1, w_2, \ldots, w_n)$, $\mathbf{x} = (x_0, x_1, x_2, \ldots, x_n)$, and $x_0 = 1$.

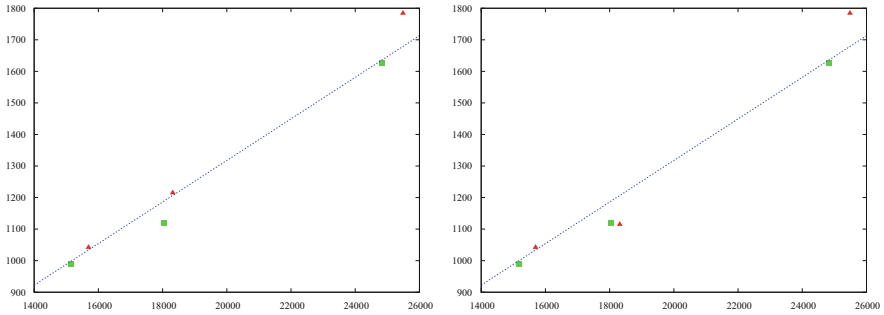


Fig. 4.15 *Left part*: A *thin line* can separate the three chapters into French and English text. The two classes are linearly separable. *Right part*: We cannot draw a line between the two classes. They are not linearly separable

The purpose of the classification algorithms is to find lines or hyperplanes separating most accurately a set of data represented by numerical vectors into two classes. As with the decision trees, these separators will be approximated from training sets and evaluated on distinct test sets. We will review the vocabulary used with machine learning methods in more detail in Sect. 5.6.2.

4.6.3 Linear Separability

It is not always the case that a line can perfectly separate the two classes of a data set. Let us return to our data set in Table 4.8 and restrict ourselves to the three shortest chapters: the 3rd, 10th, and 15th. Figure 4.15, left, shows the plot of these three chapters from the counts collected in the actual texts. A thin line can divide the chapters into two classes. Now let us imagine that in another data set, Chapter 10 in French has 18,317 letters and 1,115 *A*s instead of 18,317 and 1,215, respectively. Figure 4.15, right, shows this plot. This time, no line can pass between the two classes, and the data set is said to be not linearly separable.

Although we cannot draw a line that divides the two classes, there are workarounds to cope with not linearly separable data that we will explain in the next section.

4.6.4 Classification vs. Regression

Regression and classification use a similar formalism, and at this point, it is important to understand their differences. Given an input, regression computes a continuous numerical output. For instance, regression will enable us to compute the number of *A*s occurring in a text in French from the total number of characters.

Having 75,255 characters in Chapter 7, the regression line will predict 5,149 occurrences of *A*s (there are 5,062 in reality).

The output of a classification is a finite set of values. When there are two values, we have a binary classification. Given the number of characters and the number of *A*s in a text, classification will predict the language: French or English. For instance, having the pair (75255, 5062), the classifier will predict French.

This means that given a data set, the dimensions of the feature space will be different. Regression predicts the value of one of the features given the value of $n - 1$ features. Classification predicts the class given the values of n features. Compared to regression in our example, the dimension of the vector space used for the classification is $n + 1$: the n features and the class.

In the next sections, we will examine three categories of linear classifiers from among the most popular and efficient ones: perceptrons, logistic regression, and support vector machines. For the sake of simplicity, we will restrict our presentation to a binary classification with two classes. However, linear classifiers can generalize to handle a multinomial classification, *i.e.* three classes or more, which is the most frequent case in practice. This generalization is outside the scope of this book; see Sect. 4.11 for further references on this topic.

4.7 Perceptron

Given a data set like the one in Table 4.8, where each object is characterized by the feature vector $\mathbf{x}$ and a class, P or N, the perceptron algorithm (Rosenblatt 1958) is a simple method to find a hyperplane splitting the space into positive and negative half-spaces separating the objects. The perceptron uses a sort of gradient descent to iteratively adjust weights $(w_0, w_1, w_2, \ldots, w_n)$ representing the hyperplane until all the objects belonging to P have the property $\mathbf{w} \cdot \mathbf{x} \geq 0$, while those belonging to N have a negative dot product.

4.7.1 The Heaviside Function

As we represent the examples using numerical vectors, it is more convenient in the computations to associate the negative and positive classes, N and P, to a discrete set of two numerical values: $\{0, 1\}$. To carry this out, we pass the result of the dot product to the Heaviside step function (a variant of the *signum* function):

$$H(\mathbf{w} \cdot \mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Using the Heaviside function H, we can reformulate classification. Given a data set: $DS = \{(1, x_1^j, x_2^j, \ldots, x_n^j, y^j) | j : 1..q\}$ of q examples, where $y^j \in \{0, 1\}$, we have:

$$\begin{aligned} \hat{y}(\mathbf{x}^j) &= H(\mathbf{w} \cdot \mathbf{x}^j), \\ &= H(w_0 + w_1 x_1^j + w_2 x_2^j + \ldots + w_n x_n^j). \end{aligned}$$

We use $x_0^j = 1$ to simplify the equations, and the range of y, $\{0, 1\}$, corresponds to the classes {English, French} in Table 4.8.

4.7.2 The Iteration

Let us denote $\mathbf{w}_k$ the weight vector at step k, and $w_{i\,(k)}$, the value of its weight coordinate w_i. The perceptron algorithm starts the iteration with a weight vector $\mathbf{w}_0$ chosen randomly or set to $\mathbf{0}$ and then applies the dot product $\mathbf{w}_k \cdot \mathbf{x}^j$ one object at a time for all the members of the data set, $j : 1..q$:

- If the object is correctly classified, the perceptron algorithm keeps the weights unchanged;
- If the object is misclassified, the algorithm attempts to correct the error by adjusting $\mathbf{w}_k$ using a gradient descent:

$$w_{i\,(k+1)} \leftarrow w_{i\,(k)} + \alpha \sum_{j=1}^{q} x_i^j \times (y^j - \hat{y}^j)$$

until all the objects are correctly classified.

For a misclassified object, we have $y^j - \hat{y}^j$ equals to either $1 - 0$ or $0 - 1$. The update value is then is $\alpha \cdot x_i$ or $-\alpha \cdot x_i$, where α is the learning rate. For an object that is correctly classified, we have $y^j - \hat{y}^j = 0$, corresponding to either $0 - 0$ or $1 - 1$, and there is no weight update. The learning rate is generally set to 1 as a division of the weight vector by a constant does not affect the update rule.

4.7.3 The Two-Dimensional Case

Let us spell out the update rules in a two-dimensional space. We have the feature vectors and weight vectors defined as: $\mathbf{x} = (1, x_1, x_2)$ and $\mathbf{w} = (w_0, w_1, w_2)$. With the stochastic gradient descent, we carry out the updates using the relations:

$$
\begin{aligned}
w_0 &\leftarrow w_0 + y^j - \hat{y}^j \\
w_1 &\leftarrow w_1 + x_1^j \cdot (y^j - \hat{y}^j) \\
w_2 &\leftarrow w_2 + x_2^j \cdot (y^j - \hat{y}^j),
\end{aligned}
$$

where $y^j - \hat{y}^j$ is either, 0, -1, or 1.

4.7.4 Stop Conditions

To find a hyperplane, the objects (i.e., the points) must be separable. This is rarely the case in practice, and we often need to refine the stop conditions. We will stop the learning procedure when the number of misclassified examples is below a certain threshold or we have exceeded a fixed number of iterations.

The perceptron will converge faster if, for each iteration, we select the objects randomly from the data set.

4.8 Support Vector Machines

When a data set is linearly separable, the perceptron algorithm finds a separating hyperplane with a weight vector corresponding to a point in the solution region. Referring back to Fig. 4.14, it can be any point in the white region.

Support vector machines (Boser et al. 1992) are another type of linear classifiers that aim at finding a unique solution in the form of an optimal hyperplane. This optimal hyperplane is defined as the one that maximizes the margins between the two classes. It will be positioned at equal distance between the closest points of each class and will create the largest possible corridor with no points from either classes inside. These closest points are on the border of the corridor and are called the support vectors. In this section, we introduce the mathematical concepts behind support vector machines.

4.8.1 Maximizing the Margin

We know from geometry and vector analysis that the distance from a point $\mathbf{x}^j = (x_1^j, x_2^j, \ldots, x_n^j)$ to a hyperplane Hyp defined by the equation:

$$w_0 + w_1 x_1 + w_2 x_2 + \ldots + w_n x_n = 0$$

is given by the formula:

$$d(\mathbf{x}^j, Hyp) = \frac{|w_0 + w_1 x_1^j + w_2 x_2^j + \ldots + w_n x_n^j|}{\sqrt{w_1^2 + w_2^2 + \ldots + w_n^2}} = \frac{|w_0 + \mathbf{w} \cdot \mathbf{x}^j|}{||\mathbf{w}||},$$

where $\mathbf{w}$ is the weight vector $(w_1, w_2, \ldots w_n)$.

The optimal hyperplane is the one that maximizes this distance for all the points in the data set, $DS = \{(x_1^j, x_2^j, \ldots, x_n^j, y^j) | j : 1..q\}$. It is easier to associate the negative and positive classes, N and P, to the two numerical values: $\{-1, 1\}$ instead of $\{0, 1\}$ as in the perceptron. We can then remove the absolute value and fit the weight vector so that it maximizes the margin M:

$$\begin{aligned} &\max_{w_0, \mathbf{w}} \ M \\ &\text{subject to} \ \ y^j \cdot \frac{w_0 + \mathbf{x}^j \cdot \mathbf{w}}{||\mathbf{w}||} \geqslant M, j : 1..q, \end{aligned}$$

where $y^j \in \{-1, 1\}$.

The weight vector $(w_0, \mathbf{w})$ is defined within a constant factor, and we can set $||\mathbf{w}||$ so that $||\mathbf{w}|| \cdot M = 1$. Using this scaling operation, maximizing M is equivalent to minimizing the norm $||\mathbf{w}||$. We have then

$$\begin{aligned} &\min_{w_0, \mathbf{w}} \ ||\mathbf{w}|| \\ &\text{subject to} \ \ y^j (w_0 + \mathbf{x}^j \cdot \mathbf{w}) \geqslant 1, j : 1..q. \end{aligned}$$

4.8.2 Lagrange Multipliers

The margin maximization can be recast using a Lagrangian (a Lagrange function) (Boser et al. 1992):

$$\begin{aligned} &L(\mathbf{w}, w_0, \boldsymbol{\alpha}) = \frac{1}{2}||\mathbf{w}||^2 - \sum_{j=1}^{q} \alpha_j (y^j (w_0 + \mathbf{x}^j \cdot \mathbf{w}) - 1), \\ &\text{subject to } \alpha_j \geqslant 0, j : 1..q, \end{aligned}$$

and is then equivalent to finding a minimum of $L(\mathbf{w}, w_0, \boldsymbol{\alpha})$ with respect to $\mathbf{w}$ and a maximum with respect to $\boldsymbol{\alpha}$. The $\boldsymbol{\alpha}$ are called Lagrange multipliers.

The maximal margin is reached when the partial derivatives with respect to $\mathbf{w}$ and w_0 are 0. Computing the derivatives, we find:

$$\begin{cases} \sum_{j=1}^{q} \alpha_j y^j x_i^j & = w_i \\ \sum_{j=1}^{q} \alpha_j y^j & = 0 \end{cases}$$

We plug these values back into the Lagrangian, and we obtain:

$$L(\boldsymbol{\alpha}) = \sum_{j=1}^{q} \alpha_j - \frac{1}{2} \sum_{j=1}^{q} \sum_{k=1}^{q} \alpha_j \alpha_k y^j y^k \mathbf{x}^j \mathbf{x}^k,$$

that we maximize with respect to $\boldsymbol{\alpha}$.

We can find a solution to this optimization problem using quadratic programming techniques. Their description is beyond the scope of this book, however. There are many toolkits that we can use to solve practical problems. They include the LIBSVM (Chang and Lin 2011) and LIBLINEAR (Fan et al. 2008) toolkits.

Applying LIBLINEAR to the data in Table 4.8, we find a hyperplane equation separating the two classes so that $||\mathbf{w}|| \cdot M = 1$:

$$-0.006090937 + 0.008155714x - 0.123790484y = 0$$

and when normalizing the coefficients with respect to y we have:

$$0.049203592 - 0.065883207x + y = 0$$

This corresponds to unique point $(w_1, w_0) = (0.065883207, -0.049203592)$ in the weight space in Fig. 4.14.

Support vector machines can also handle not linearly separable examples using kernels or through the soft margin method. See the original papers by Boser et al. (1992) and Cortes and Vapnik (1995) for a presentation.

4.9 Logistic Regression

In their elementary formulation, the perceptron and support vector machines use hyperplanes as absolute, unmitigated boundaries between the classes. In many data sets, however, there are no such clear-cut thresholds to separate the points. Figure 4.15 is an example of this that shows regions where the nonlinearly separable classes have points with overlapping feature values.

Logistic regression is an attempt to define a smoother transition between the classes. Instead of a rigid boundary in the form of a step function, logistic regression uses the logistic curve (Verhulst 1838, 1845) to model the probability of a point $\mathbf{x}$

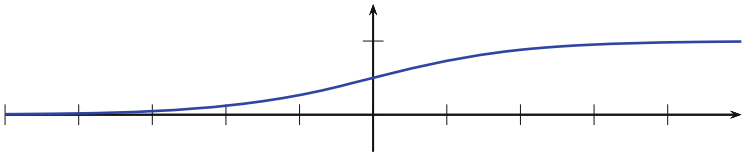


Fig. 4.16 The logistic curve: $f(x) = \dfrac{1}{1+e^{-x}}$

(an observation) to belong to a class. Figure 4.16 shows this curve, whose equation is given by:

$$f(x) = \frac{1}{1+e^{-x}}.$$

Logistic regression was first introduced by Berkson (1944) in an attempt to model the percentage of individuals killed by the intake of a lethal drug. Berkson observed that the higher the dosage of the drug, the higher the mortality, but as some individuals are more resilient than others, there was no threshold value under which all the individuals would have survived and above which all would have died. Intuitively, this fits very well the shape of the logistic curve in Fig. 4.16, where the mortality rate is close to 0 for lower values of x (the drug dosage), then increases, and reaches a mortality rate of 1 for higher values of x.

Berkson used one feature, the dosage x, to estimate the mortality rate, and he derived the probability model:

$$P(y=1|x) = \frac{1}{1+e^{-w_0-w_1 x}},$$

where y denotes the class, either survival or death, with the respective labels 0 and 1, and (w_0, w_1) are weight coefficients that are fit using the maximum likelihood method.

Using this assumption, we can write a general probability model for feature vectors $\mathbf{x}$ of any dimension:

$$P(y=1|\mathbf{x}) = \frac{1}{1+e^{-\mathbf{w}\cdot\mathbf{x}}},$$

where $\mathbf{w}$ is a weight vector.

As we have two classes and the sum of their probabilities is 1, we have:

$$P(y=0|\mathbf{x}) = \frac{e^{-\mathbf{w}\cdot\mathbf{x}}}{1+e^{-\mathbf{w}\cdot\mathbf{x}}}.$$

These probabilities are extremely useful in practice.

The logit transformation corresponding to the logarithm of the odds ratio:

$$\ln \frac{P(y=1|\mathbf{x})}{P(y=0|\mathbf{x})} = \ln \frac{P(y=1|\mathbf{x})}{1-P(y=1|\mathbf{x})} = \mathbf{w}\cdot\mathbf{x}$$

is also frequently used to fit the data to a straight line or a hyperplane.

4.9.1 Fitting the Weight Vector

To build a functional classifier, we need now to fit the weight vector $\mathbf{w}$; the maximum likelihood is a classical way to do this. Given a data set, $DS = \{(1, x_1^j, x_2^j, \ldots, x_n^j, y^j)|j : 1..q\}$, containing a partition in two classes, $P\ (y=1)$ and $N\ (y=0)$, and a weight vector $\mathbf{w}$, the likelihood to have the classification observed in this data set is:

$$L(\mathbf{w}) = \prod_{\mathbf{x}^j \in P} P(y^j = 1|\mathbf{x}^j) \times \prod_{\mathbf{x}^j \in N} P(y^j = 0|\mathbf{x}^j),$$

$$= \prod_{\mathbf{x}^j \in P} P(y^j = 1|\mathbf{x}^j) \times \prod_{\mathbf{x}^j \in N} (1 - P(y^j = 1|\mathbf{x}^j)).$$

We can rewrite the product using y^j as powers of the probabilities as $y^j = 0$, when $\mathbf{x}^j \in N$ and $y^j = 1$, when $\mathbf{x}^j \in P$:

$$L(\mathbf{w}) = \prod_{\mathbf{x}^j \in P} P(y^j = 1|\mathbf{x}^j)^{y^j} \times \prod_{\mathbf{x}^j \in N} (1 - P(y^j = 1|\mathbf{x}^j))^{1-y^j},$$

$$= \prod_{(\mathbf{x}^j, y^j) \in DS} P(y^j = 1|\mathbf{x}^j)^{y^j} \times (1 - P(y^j = 1|\mathbf{x}^j))^{1-y^j}.$$

Maximizing the Likelihood

We fit $\mathbf{w}$, and train a model by maximizing the likelihood of the observed classification:

$$\hat{\mathbf{w}} = \arg\max_{\mathbf{w}} \prod_{\mathbf{x}^j \in DS} P(y^j = 1|\mathbf{x}^j)^{y^j} \times (1 - P(y^j = 1|\mathbf{x}^j))^{1-y^j}.$$

To maximize this term, it is more convenient to work with sums rather than with products, and we take the logarithm of it (log-likelihood):

$$\hat{\mathbf{w}} = \arg\max_{\mathbf{w}} \sum_{(\mathbf{x}^j, y^j) \in DS} y^j \ln P(y^j = 1|\mathbf{x}^j) + (1 - y^j) \ln(1 - P(y^j = 1|\mathbf{x}^j)).$$

Using the logistic curves to express the probabilities, we have:

$$\hat{\mathbf{w}} = \arg\max_{\mathbf{w}} \sum_{(\mathbf{x}^j, y^j) \in DS} y^j \ln \frac{1}{1 + e^{-\mathbf{w} \cdot \mathbf{x}^j}} + (1 - y^j) \ln \frac{e^{-\mathbf{w} \cdot \mathbf{x}^j}}{1 + e^{-\mathbf{w} \cdot \mathbf{x}^j}}.$$

In contrast to linear regression that uses least mean squares, here we fit a logistic curve so that it maximizes the likelihood of the classification – partition – observed in the training set.

4.9.2 The Gradient Ascent

We can use the gradient ascent to compute this maximum. This method is analogous to the gradient descent that we reviewed in Sect. 4.5.2; we move upward instead. A Taylor expansion of the log-likelihood gives us: $\ell(\mathbf{w} + \mathbf{v}) = \ell(\mathbf{w}) + \mathbf{v} \cdot \nabla \ell(\mathbf{w}) + \ldots$ When $\mathbf{w}$ is collinear with the gradient, we have:

$$\ell(\mathbf{w} + \alpha \nabla \ell(\mathbf{w})) \approx \ell(\mathbf{w}) + \alpha ||\nabla \ell(\mathbf{w})||^2.$$

The inequality:

$$\ell(\mathbf{w}) < \ell(\mathbf{w} + \alpha \nabla \ell(\mathbf{w}))$$

enables us to find a sequence of increasing values of the log-likekihood. We use the iteration:

$$\mathbf{w}_{k+1} = \mathbf{w}_k + \alpha \nabla \ell(\mathbf{w}_k)$$

to carry this out until we reach a maximum.

Computing the Gradient

We compute the partial derivatives of the log-likelihood to find the gradient:

$$\frac{\partial \ell(\mathbf{w})}{\partial w_i} = \sum_{(\mathbf{x}^j, y^j) \in DS} y^j (1 + e^{-\mathbf{w}.\mathbf{x}^j}) \frac{x_i^j e^{-\mathbf{w}.\mathbf{x}^j}}{(1 + e^{-\mathbf{w} \cdot \mathbf{x}^j})^2} +$$

$$(1 - y^j) \frac{1 + e^{-\mathbf{w}.\mathbf{x}^j}}{e^{-\mathbf{w}.\mathbf{x}^j}} \cdot \frac{-x_i^j \cdot e^{-\mathbf{w} \cdot \mathbf{x}^j} (1 + e^{-\mathbf{w} \cdot \mathbf{x}^j}) + x_i^j \cdot e^{-\mathbf{w}.\mathbf{x}^j} \cdot e^{-\mathbf{w}.\mathbf{x}^j}}{(1 + e^{-\mathbf{w} \cdot \mathbf{x}^j})^2},$$

$$= \sum_{(\mathbf{x}^j, y^j) \in DS} y^j \frac{x_i^j e^{-\mathbf{w}.\mathbf{x}^j}}{1 + e^{-\mathbf{w}\cdot\mathbf{x}^j}} + (1 - y^j) \cdot \frac{-x_i^j \cdot (1 + e^{-\mathbf{w}\cdot\mathbf{x}^j}) + x_i^j \cdot e^{-\mathbf{w}.\mathbf{x}^j}}{1 + e^{-\mathbf{w}\cdot\mathbf{x}^j}},$$

$$= \sum_{(\mathbf{x}^j, y^j) \in DS} x_i^j \cdot \frac{y^j \cdot (1 + e^{-\mathbf{w}.\mathbf{x}^j}) - 1}{1 + e^{-\mathbf{w}\cdot\mathbf{x}^j}},$$

$$= \sum_{(\mathbf{x}^j, y^j) \in DS} x_i^j \cdot (y^j - \frac{1}{1 + e^{-\mathbf{w}\cdot\mathbf{x}^j}}).$$

Weight Updates

Using the gradient values, we can now compute the weight updates at each step of the iteration. As with linear regression, we can use a stochastic or a batch method. For $DS = \{(1, x_1^j, x_2^j, \ldots, x_n^j, y^j) | j : 1..q\}$, the updates of $\mathbf{w} = (w_0, w_1, \ldots, w_n)$ are:

- With the stochastic gradient ascent:

$$w_{i\,(k+1)} \leftarrow w_{i\,(k)} + \alpha \cdot x_i^j \cdot (y^j - \frac{1}{1 + e^{-\mathbf{w}_k \cdot \mathbf{x}^j}});$$

- With the batch gradient ascent:

$$w_{i\,(k+1)} \leftarrow w_{i\,(k)} + \frac{\alpha}{q} \cdot \sum_{j=1}^{q} x_i^j \cdot (y^j - \frac{1}{1 + e^{-\mathbf{w}_k \cdot \mathbf{x}^j}}).$$

As with the gradient descent, the convergence stops when $||\nabla \ell(\mathbf{w})||$ is less than a predefined threshold.

4.10 Encoding Symbolic Values as Numerical Features

Along with this overview of numerical classification methods, a practical question comes to mind: how can we apply them to symbolic – or nominal – attributes like the ones in Table 4.7?

The answer is that we need to convert the symbolic attributes into numerical vectors before we can use the linear classifiers. The classical way to do this is to represent each attribute domain – the set of the allowed or observed values of an attribute – as a vector of binary digits. Let us exemplify this with the *Outlook* attribute in Table 4.7:

Table 4.10 A representation of the symbolic values in Table 4.7 as numerical vectors

	Attributes										
	Outlook			**Temperature**			**Humidity**		**Windy**		
Object	Sunny	Overcast	Rain	Hot	Mild	Cool	High	Normal	True	False	**Class**
1	1	0	0	1	0	0	1	0	0	1	*N*
2	1	0	0	1	0	0	1	0	1	0	*N*
3	0	1	0	1	0	0	1	0	0	1	*P*
4	0	0	1	0	1	0	1	0	0	1	*P*
5	0	0	1	0	0	1	0	1	0	1	*P*
6	0	0	1	0	0	1	0	1	1	0	*N*
7	0	1	0	0	0	1	0	1	1	0	*P*
8	1	0	0	0	1	0	1	0	0	1	*N*
9	1	0	0	0	0	1	0	1	0	1	*P*
10	0	0	1	0	1	0	0	1	0	1	*P*
11	1	0	0	0	1	0	0	1	1	0	*P*
12	0	1	0	0	1	0	1	0	1	0	*P*
13	0	1	0	1	0	0	0	1	0	1	*P*
14	0	0	1	0	1	0	1	0	1	0	*N*

- *Outlook* has three possible values: $\{sunny, overcast, rain\}$. Its numerical representation is then a three-dimensional vector, (x_1, x_2, x_3), whose axes are tied respectively to *sunny*, *overcast*, and *rain*.
- To reflect the value of the attribute, we set the corresponding coordinate to 1 and the others to 0. Using the examples in Table 4.7, the name–value pair $[Outlook = sunny]$ will be encoded as $(1, 0, 0)$, $[Outlook = overcast]$ as $(0, 1, 0)$, and $[Outlook = rain]$ as $(0, 0, 1)$.

For a given attribute, the dimension of the vector will then be defined by the number of its possible values, and each vector coordinate will be tied to one of the possible values of the attribute.

So far, we have one vector for each attribute. To represent a complete object, we will finally concatenate all these vectors into a larger one characterizing this object. Table 4.10 shows the complete conversion of the data set using vectors of binary values.

If an attribute has from the beginning a numerical value, it does not need to be converted. It is, however, a common practice to scale it so that the observed values in the training set range from 0 to 1 or from -1 to $+1$.

4.11 Further Reading

Information theory is covered by many books, many of them requiring a good mathematical background. The text by Manning and Schütze (1999, Chap. 2) provides a short and readable introduction oriented toward natural language processing.

Machine-learning techniques are now ubiquitous in all the fields of natural language processing. ID3 outputs classifiers in the form of decision trees that are easy to understand. It is a simple and robust algorithm. Logistic regression, perceptrons, and support vector machines are other popular classifiers. Which one to choose has no easy answer as they may have different performances on different data sets. My preferences are leaning toward ID3 and logistic repression, although this does not exclude the others. Supervised machine-learning is a large and evolving domain. In this chapter, we set aside many details and techniques. Hastie et al. (2009), Saporta (2011), Murphy (2012), and James et al. (2013) are mathematical references on classification and statistical learning in general that can complement this chapter. Schölkopf and Smola (2002) is a more focused reference on support vector machines.

We used regression to introduce linear classification techniques. This line-fitting process has a somehow enigmatic name. It is due to Galton (1886) who modeled the transmission of stature from parents to children. He gathered a data set of the heights of children and parents and observed that taller-than-average parents tended to have children shorter than they, and that shorter parents tended to have taller children than they. Galton called this a *regression towards mediocrity*.

A number of machine-learning toolkits are available from the Internet. R is a set of statistical and machine-learning functions with a script language (http://www.r-project.org/). Weka (Hall et al. 2009; Witten and Frank 2005) is a collection of data mining algorithms written in Java (http://www.cs.waikato.ac.nz/ml/weka/). LIBLINEAR (Fan et al. 2008) and LIBSVM (Chang and Lin 2011) are efficient implementations of logistic regression and support vector machines in C (http://www.csie.ntu.edu.tw/~cjlin/liblinear/, http://www.csie.ntu.edu.tw/~cjlin/libsvm/). C4.5 (Quinlan 1993), ID3's successor, is available from its creator's web page (http://www.rulequest.com/Personal/).

Exercises

4.1. Implement the ID3 algorithm in Prolog, Perl, or another language. Test it on the data set in Table 4.7.

4.2. Implement linear regression using the gradient descent. Test it on the data set in Table 4.8 with, respectively, English and French.

4.3. Implement the perceptron algorithm. Test it on the data set in Table 4.8.

4.4. Implement logistic regression. Test it on the data set in Table 4.8.

Chapter 5
Counting Words

> On trouve ainsi qu'un événement étant arrivé de suite, un nombre quelconque de fois, la probabilité qu'il arrivera encore la fois suivante, est égale à ce nombre augmenté de l'unité, divisé par le même nombre augmenté de deux unités. En faisant, par exemple, remonter la plus ancienne époque de l'histoire, à cinq mille ans, ou à 1826213 jours, et le Soleil s'étant levé constamment, dans cet intervalle, à chaque révolution de vingt-quatre heures, il y a 1826214 à parier contre un qu'il se lèvera encore demain.
>
> Pierre-Simon Laplace. *Essai philosophique sur les probabilités*. 1840.
> See explanations in Sect. 5.7.2.

5.1 Counting Words and Word Sequences

We saw in Chap. 2 that words have specific contexts of use. Pairs of words like *strong* and *tea* or *powerful* and *computer* are not random associations but the result of a preference. A native speaker will use them naturally, while a learner will have to learn them from books – dictionaries – where they are explicitly listed. Similarly, the words *rider* and *writer* sound much alike in American English, but they are likely to occur with different surrounding words. Hence, hearing an ambiguous phonetic sequence, a listener will discard the improbable *rider of books* or *writer of horses* and prefer *writer of books* or *rider of horses* (Church and Mercer 1993).

In lexicography, extracting recurrent pairs of words – collocations – is critical to finding the possible contexts of a word and citing real examples of its use. In speech recognition, the statistical estimate of a word sequence – also called a **language model** – is a key part of the recognition process. The language model component of a speech recognition system enables the system to predict the next word given a

P.M. Nugues, *Language Processing with Perl and Prolog*, Cognitive Technologies,
DOI 10.1007/978-3-642-41464-0_5, © Springer-Verlag Berlin Heidelberg 2014

sequence of previous words: *the writer of books, novels, poetry,* etc., rather than of *the writer of hooks, nobles, poultry*.

Knowing the frequency of words and sequences of words is crucial in many fields of language processing. In addition to speech recognition and lexicography, they include parsing, semantic interpretation, and translation. In this chapter, we introduce techniques to obtain word frequencies from a corpus and to build language models. We also describe a set of related concepts that are essential to understand them.

5.2 Text Segmentation

Most language processing techniques, such as language modeling and morphological and syntactic parsing, consider words and sentences. When the input data is a stream of characters, we must first segment it, i.e., identify the words and sentences in it, before we can apply any further operation to the text. We call this step **text segmentation** or **tokenization**. A tokenizer can also remove formatting instructions, such as XML tags, if any.

Originally, early European scripts had no symbols to mark segment boundaries inside a text. Ancient Greeks and Romans wrote their inscriptions as continuous strings of characters flowing from left to right and right to left without punctuation or spaces. The *lapis niger*, one of the oldest remains of the Latin language, is an example of this writing style, also called **boustrophedon** (Fig. 5.1).

As the absence of segmentation marks made texts difficult to read, especially when engraved on a stone, Romans inserted dots to delimit the words and thus improve their legibility. This process created the graphic word as we know it: a sequence of letters between two specific signs. Later white spaces replaced the dots as word boundaries and Middle Ages scholars introduced a set of punctuation signs: commas, full stops, question and exclamation marks, colons, and semicolons, to delimit phrases and sentences.

5.2.1 What Is a Word?

The definition of what a word is, although apparently obvious, is in fact surprisingly difficult. A naïve description could start from its historical origin: a sequence of alphabetic characters delimited by two white spaces. This is an approximation. In addition to white spaces, words can end with commas, question marks, periods, etc. Words can also include dashes and apostrophes that, depending on the context, have a different meaning.

Word boundaries vary according to the language and orthographic conventions. Compare these different spellings: *news stand*, *news-stand*, and *newsstand*. Although the latter one is considered more correct, the two other forms are

Fig. 5.1 Latin inscriptions on the *lapis niger*. *Corpus inscriptionum latinarum*, CIL I, 1 (Picture from Wikipedia)

also frequent. Compare also the convention in German to bind together adjacent nouns as in *Gesundheitsreform*, as opposed to English that would more often separate them, as in *health reform*. Compare finally the ambiguity of punctuation marks, as in the French word *aujourd'hui*, 'today', which forms a single word, and *l'article*, 'the article', where the sequence of an article and a noun must be separated before any further processing.

In corpus processing, text elements are generally called **tokens**. Tokens include words and also punctuation, numbers, abbreviations, or any other similar type of string. Tokens may mix characters and symbols as:

- Numbers: 9,812.345 (English and French from the eighteenth to nineteenth century century), 9 812,345 (current French and German) 9.812,345 (French from the nineteenth to early twentieth century);
- Dates: 01/02/2003 (French and British English), 02/01/2003 (US English), 2003/02/01 (Swedish);
- Abbreviations and acronyms: km/h, m.p.h., S.N.C.F.;
- Nomenclatures: A1-B45, /home/pierre/book.tex;
- Destinations: Paris–New York, Las Palmas–Stockholm, Rio de Janeiro–Frankfurt am Main;
- Telephone numbers: (0046) 46 222 96 40;
- Tables;
- Formulas: $E = mc^2$.

As for the words, the definition of what is a sentence is also tricky. A naïve definition would be a sequence of words ended by a period. Unfortunately, periods are also ambiguous. They occur in numbers and terminate abbreviations, as in *etc.* or *Mr.*, which makes sentence isolation equally complex. In the next sections, we examine techniques to break a text into words and sentences, and to count the words.

5.2.2 *Breaking a Text into Words and Sentences*

Tokenization breaks a character stream, that is, a text file or a keyboard input, into tokens – separated words – and sentences. In Prolog, it results in a list of atoms. For this paragraph, such a list looks like:

```
[['Tokenization', breaks, a, character, stream, (,),
that, is, (,), a, text, file, or, a, keyboard, input,
(,), into, tokens, -, separated, words, -, and,
sentences, '.'], ['In', 'Prolog', it, results, in,
a, list, of, atoms, '.'], ['For', this, paragraph,
(,), such, a, list, looks, like, :]]
```

An basic format to output or store tokenized texts is to print one word per line and have a blank line to separate sentences as in:

```
In
Prolog
,
it
results
in
a
list
of
atoms
.

For
this
paragraph
,
such
a
list
looks
like
:
```

5.3 Tokenizing Words

We now introduce tokenization techniques using two complementary approaches. The first one considers the unit boundaries, and the second one their content. We will then merge them into a more elaborate program in Perl. We will also provide with

an implementation in Prolog. Perl is generally faster and is well suited to process large quantities of text.

5.3.1 Using White Spaces

Tokenizing texts using white spaces as word delimiters is the most elementary technique. It is straightforward in Perl, as shown in the program below: we just replace sequences of white spaces in the text with a new line, and we consider what is between two white spaces to be a word. In the program, we use the `\s` character class to represent the white space:

```
$text = <>;
while ($line = <>) {
  $text .= $line;
}
$text =~ s/\s+/\n/g;
print $text;
```

However, this does not work perfectly, and as with the first lines of the *Odyssey*:

> Tell me, O muse, of that ingenious hero who travelled far and wide after he had sacked the famous town of Troy.

where the commas are not segmented from the words:

```
Tell
me,
O
muse,
of
that
ingenious
hero
...
```

5.3.2 Using White Spaces and Punctuation

The previous program failed to tokenize the punctuation. We improve it with a few regular expressions to separate the punctuation signs from the words and insert white spaces around them. The punctuation we process corresponds to:

1. The dot: `.`
2. Other boundary signs: `,;:?!#$%&-/\`
3. Brackets: `"()[]{}<>`, and

4. Quotes: `` ` '' '.

We then tokenize the text according to white spaces as in the previous section:

```
$text = <>;
while ($line = <>) {
  $text .= $line;
}
$text =~ s/\./ . /g;
$text =~ s/([,;:?!#$%&\-\/\\])/ $1 /g;
$text =~ s/(["\(\)\[\]{}\<\>])/ $1 /g;
$text =~ s/(``|`|''|')/ $1 /g;
# Remove leading spaces
$text =~ s/^ *//g;
$text =~ s/\s+/\n/g;
print $text;
```

Applying it to our small text results in:

```
Tell
me
,
O
muse
,
of
that
ingenious
hero
who
travelled
far
...
```

This second program produces a better result than our first one, although not perfect. Decimal numbers, for example, would not be properly processed. The program would match the point of decimal numbers such as 3.14 and insert new lines between 3 and 14. The apostrophe inside words is another ambiguous sign. The tokenization of auxiliary and negation contractions in English is unpredictable without a morphological analysis. It requires a dictionary with all the forms (Table 5.1).

In French, apostrophes corresponding to the elided *e* have a regular behavior as in

Si j'aime et d'aventure → `si j' aime et d' aventure`

but there are words like *aujourd'hui*, 'today', that correspond to a single entity and are not tokenized.

Table 5.1 Apostrophe tokenization in English

Contracted form	Example	Tokenization	Expanded form
'm	*I'm*	`I 'm`	*I am*
'd	*we'd*	`we 'd`	*we had* or *we would*
'll	*we'll*	`we 'll`	*we will*
're	*you're*	`you 're`	*you are*
've	*I've*	`I 've`	*I have*
n't	*can't*	`can n't`	*cannot*
's	*she's*	`she 's`	*she has* or *she is*
's	*Pierre's book*	`Pierre 's book`	Possessive marking

5.3.3 Defining Contents

Alternatively, we can explicitly define the content of words. We consider then that contiguous sequences of alphanumeric characters, including the dash and the quote, are words, and we isolate them on a single line. We isolate the punctuation symbols on a single line as well. All the other symbols will mark a separation.

We use the `tr` operator now and formulate tokenization as:

- If a character is not a letter or a punctuation sign, then replace it by a new line. Note that the dash character in `tr` as well as in character classes means an interval and that we have to quote it to process it in a text.
- If it is a punctuation sign, then have it on a single line (insert it between two new lines).
- Finally, reduce contiguous sequences of new lines to a single occurrence.

```
$text = <>;
while ($line = <>) {
  $text .= $line;
}
$text =~ tr/a-zåàâäæçéèêëîïôöœùûüÿßA-ZÅÀÂÄÆÇÉÈÊËÎÏÔÖŒÙÛÜŸ'
  ()\-,.?!:;/\n/cs;
$text =~ s/([,.?!:;()'\-])/\n$1\n/g;
$text =~ s/\n+/\n/g;
print $text;
```

5.3.4 Tokenizing Texts in Prolog

We can define a Prolog tokenizer with a grammar, where tokens are sequences of characters of the same class:

- A token is a sequence of alphabetic characters or digits.
- Other characters mark the token termination and consist of carriage returns, blanks, tabulations, punctuation signs, or other ASCII symbols or commands.

The tokenization program `tokenize/2` takes a list of character codes as input and returns a list of tokens. The predicate `char_type/2` determines the type of a character code: alphanumerical, blank, or other. It is a built-in SWI Prolog predicate compatible with the UTF-8 character set (charset). The first `tokenize/2` rule corresponds to the termination condition. The second `tokenize/2` rule tests the type of the head of the list. It skips the blanks. When it reaches an alphanumerical character in the third rule, it calls `make_word/4`, which builds a word out of next letters or digits in the list. When `tokenize/2` encounters another symbol in the fourth rule, it makes a single token out of it.

You can use the `read_file/2` predicate from Appendix A, "An Introduction to Prolog," to read the character codes from a file.

```
% tokenize(+CharCodes, -Tokens)
%  breaks a list of character codes into a list of tokens.
tokenize([], []).
tokenize([CharCode | RestCodes], Tokens) :- % a blank
  char_type(CharCode, space),
  !,
  tokenize(RestCodes, Tokens).
tokenize([CharCode | CharCodes], [Word | Tokens]) :-
  char_type(CharCode, alnum), % an alphanumical
  !,
  make_word([CharCode | CharCodes], alnum, WordCodes, RestCodes),
  name(Word, WordCodes),
  tokenize(RestCodes, Tokens).
tokenize([CharCode | CharCodes], [Char | Tokens]) :- % other
  !,
  name(Char, [CharCode]),
  tokenize(CharCodes, Tokens).

% make_word(+CharCodes, +Type, -WordCodes, -RestCodes)
make_word([CharCode1, CharCode2 | CharCodes], alnum,
    [CharCode1 | WordCodes], RestCodes) :-
  char_type(CharCode2, alnum),
  !,
  make_word([CharCode2 | CharCodes], alnum, WordCodes,
    RestCodes).
make_word([CharCode | RestCodes], alnum, [CharCode],
    RestCodes).
```

5.3.5 *Tokenizing Using Classifiers*

So far, we have carried out tokenization using rules that we have explicitly defined and implemented using regular expressions or Prolog. A second option is to use classifiers such as logistic regression (Sect. 4.9) and to train a tokenizer from a corpus. Given an input queue of characters, we then formulate tokenization as a binary classification: is the current character the end of a token or not? If the classifier predicts a token end, we insert a new line.

Table 5.2 The features extracted from the second *n* in *Pierre Vinken*, the *d* in *old*, and the dot in *Nov.*. The two classes to learn are **inside token** and **token end**

Context	Current char.	Previous pair	Next char.	Next pair	Class	Action
*Vinke**n**,*	*n*	*en*	,	,␣	Token end	New line
*ol**d**,*	*d*	*ld*	,	,␣	Token end	New line
Nov.	*v*	*ov*	.	.␣	Inside token	Nothing

Before we can train our classifier, we need a corpus and an annotation to mark the token boundaries. Let us use the OpenNLP format as an example. The Apache OpenNLP library is an open-source toolkit for natural language processing. It features a classifier-based tokenizer and has defined an annotation for it (Apache OpenNLP Development Community 2012). A training corpus consists of a list of sentences with one sentence per line, where the white spaces are unambiguous token boundaries. The other token boundaries are marked with the `<SPLIT>` tag, as in these two sentences:

```
Pierre Vinken<SPLIT>, 61 years old<SPLIT>, will join the
    board as a nonexecutive director Nov. 29<SPLIT>.
Mr. Vinken is chairman of Elsevier N.V.<SPLIT>, the Dutch
    publishing group<SPLIT>.
```

Note that in the example above, the sentence lengths are too long to fit the size of the book and we inserted two additional breaks and leading spaces to denote a continuing sentence. In the corpus file, every new line corresponds to a new sentence.

Once we have an annotation, we need to define the features we will use for the classifier. We already used features in Sect. 4.4 in the form of letter frequencies to classify the language of a text. For the tokenization, we will follow Reynar (1998, pp. 69–70), who describes a simple feature set consisting of four features:

- The current character,
- The pair formed of the previous and current characters,
- The next character,
- The pair formed of the two next characters.

As examples, Table 5.2 shows the features extracted from three characters in the sentences above: the second *n* in *Pierre Vinken*, the *d* in *old*, and the dot in *Nov.* From these features, the classifier will a create model and discriminate between the two classes: **inside token** and **token end**.

Before we can learn the classifiers, we need a corpus annotated with the `<SPLIT>` tags. We can create one by tokenizing a large text manually – a tedious task – or by reconstructing a nontokenized text from an already tokenized text. See, for example, the Penn Treebank (Marcus et al. 1993) for English.

We extract a training data set from the corpus by reading all the characters and extracting for each character their four features and their class. We then train the

classifier, for instance, using logistic regression, to create a model. Finally, given a nontokenized text, we apply the classifier and the model to each character of the text to decide if it is inside a token or if it is a token end.

5.4 Sentence Segmentation

5.4.1 The Ambiguity of the Period Sign

Sentences usually end with a period, and we will use this sign to recognize boundaries. However, this is an ambiguous symbol that can also be a decimal point or appear in abbreviations or ellipses. To disambiguate it, we introduce now two main lines of techniques identical to those we used for tokenization: rules and classifiers.

Although in this chapter, we describe sentence segmentation after tokenization, most practical systems use them in a sequence, where sentence segmentation is the first step followed by tokenization.

5.4.2 Rules to Disambiguate the Period Sign

We will consider that a period sign either corresponds to a sentence end, a decimal point, or a dot in an abbreviation. Most of the time, we can recognize these three cases by examining a limited number of characters to the right and to the left of the sign. The objective of disambiguation rules is then to describe for each case what can be the left and right context of a period.

The disambiguation is easier to implement as a two-pass search: the first pass recognizes decimal numbers or abbreviations and annotates them with a special marking. The second one runs the detector on the resulting text. In this second pass, we also include the question and exclamation marks as sentence boundary markers.

We can generalize this strategy to improve the sentence segmentation with specific rules recognizing dates, percentages, or nomenclatures that can be run as different processing stages. However, there will remain cases where the program fails, notably with abbreviations.

5.4.3 Using Regular Expressions

Starting from the most simple rule to identity sentence boundaries, a period corresponds to a full stop, Grefenstette and Tapanainen (1994) experimented on

Table 5.3 Recognizing numbers (After Grefenstette and Tapanainen (1994))

Fractions, dates	`[0-9]+(\/[0-9]+)+`
Percent	`([+\-])?[0-9]+(\.)?[0-9]*%`
Decimal numbers	`([0-9]+,?)+(\.[0-9]+|[0-9]+)*`

Table 5.4 Regular expressions to recognize abbreviations and performance breakdown. The *Correct* column indicates the number of correctly recognized instances, *Errors* indicates the number of errors introduced by the regular expression, and *Full stop* indicates abbreviations ending a sentence where the period is a full stop at the same time (After Grefenstette and Tapanainen (1994))

Regex	Correct	Errors	Full stop
`[A-Za-z]\.`	1,327	52	14
`[A-Za-z]\.([A-Za-z0-9]\.)+`	570	0	66
`[A-Z][bcdfghj-np-tvxz]+\.`	1,938	44	26
Totals	3,835	96	106

a set of increasingly complex regular expressions to carry out segmentation. They evaluated them on the Brown corpus (Francis and Kucera 1982).

About 7 % of the sentences in the Brown corpus contain at least one period, which is not a full stop. Using their first rule, Grefenstette and Tapanainen could correctly recognize 93.20 % of the sentences. As a second step, they designed the set of regular expressions in Table 5.3 to recognize numbers and remove decimal points from the list of full stops. They raised to 93.78 % the number of correctly segmented sentences.

Regular expressions in Table 5.3 are designed for English text. French and German decimal numbers would have a different form as they use a comma as decimal point and a period or a space as a thousand separator:

```
([0-9]+(.| )?)*[0-9](,[0-9]+)
```

Finally, Grefenstette and Tapanainen added regular expressions to recognize abbreviations. They used three types of patterns:

- A single capital followed by a period as *A.*, *B.*, *C.*
- A sequence of letters and periods as in *U.S.*, *i.e.*, *m.p.h.*,
- A capital letter followed by a sequence of consonants as in *Mr.*, *St.*, *Ms.*

Table 5.4 shows the corresponding regular expressions as well as the number of abbreviations they recognize and the errors they introduce. Using them together with the regular expressions to recognize decimal numbers, Grefenstette and Tapanainen could increase the correct segmentation rate to 97.66 %.

5.4.4 Improving the Tokenizer Using Lexicons

Grefenstette and Tapanainen (1994) further improved their tokenizer by automatically building an abbreviation lexicon from their corpus. To identify potential

Table 5.5 The features extracted from *Nov.* and *29.* in the example sentences in Sect. 5.3.5. The two classes to learn are **inside sentence** and **end of sentence**

Context	Prefix	Suffix	Previous word	Next word	Prefix abbrev.	Class
Nov.	*Nov*	nil	*director*	*29.*	Yes	Inside sentence
29.	*29*	nil	*Nov.*	*Mr.*	No	End of sentence

abbreviations, they used the following idea: a word ending with a period that is followed by either a comma, a semicolon, a question mark, or a lowercase letter is a likely abbreviation. Grefenstette and Tapanainen (1994) applied this idea to their corpus; however, as they gathered many words that were not abbreviations, they removed all the strings in the list that appeared without a trailing period somewhere else in the corpus. They then reached 98.35 %.

Finally, using a lexicon of words and common abbreviations, *Mr.*, *Sen.*, *Rep.*, *Oct.*, *Fig.*, *pp.*, etc., they could recognize 99.07 % of the sentences. Mikheev (2002) describes another efficient method that learns tokenization rules from the set of ambiguous tokens distributed in a document. While most published experiments have been conducted on English, Kiss and Strunk (2006) present a multilingual statistical method that can be trained on unannotated corpora.

5.4.5 Sentence Detection Using Classifiers

As for tokenization, we can use classifiers, such as decision trees or logistic regression, to segment sentences. The idea is simple: given a period in a text (or a question or an exclamation mark), classify it as the end of a sentence or not. The implementation is identical to that in Sect. 5.3.5 and we can use the same corpus: we just ignore the `<SPLIT>` tags.

Practically, we need to collect a data set and define the features to associate to the periods. Reynar and Ratnaparkhi (1997) proposed a method that we describe here. As corpus, they used the Penn Treebank (Marcus et al. 1993), from which they extracted all the strings separated by white spaces and containing a period. They used a compact set of eight features:

1. The characters in the string to the left of the period (the prefix);
2. The character to the right of the period (the suffix);
3. The word to the left of the string;
4. The word to the right of the string;
5. Whether the prefix (resp. suffix) is on a list of abbreviations;
6. Whether the word to the left (resp. to the right) is on a list of abbreviations.

Table 5.5 shows the features for the periods in *Nov.* and *29.* in the example sentences in Sect. 5.3.5. The first four features are straightforward to extract. We need a list of abbreviations for the rest. We can build this list automatically using the method described in Sect. 5.4.4.

Reynar and Ratnaparkhi (1997) used logistic regression to train their classification models and discriminate between the two classes: **inside sentence** and **end of sentence**.

5.5 *N*-Grams

5.5.1 Some Definitions

The first step of lexical statistics consists in extracting the list of **word types** or **types**, i.e., the distinct words, from a corpus, along with their frequencies. Within the context of lexical statistics, word types are opposed to word tokens, the sequence of running words of the corpus. The excerpt from George Orwell's *Nineteen Eighty-Four*:

> War is peace
> Freedom is slavery
> Ignorance is strength

has nine tokens and seven types. The type-to-token ratio is often used as an elementary measure of a text's density.

Collocations and language models also use the frequency of pairs of adjacent words: **bigrams**, for example, how many *of the* there are in this text; of word triples: **trigrams**; and more generally of fixed sequences of n words: ***n*-grams**. In lexical statistics, single words are called **unigrams**.

Jelinek (1990) exemplified corpus statistics and trigrams with the sentence

> We need to resolve all of the important issues within the next two days

selected from a 90-million-word corpus of IBM office correspondences. Table 5.6 shows each word of this sentence, its rank in the corpus, and other words ranking before it according to a linear combination of trigram, bigram, and unigram probabilities. In this corpus, *We* is the ninth most probable word to begin a sentence. More likely words are *The*, *This*, etc. Following *We*, *need* is the seventh most probable word. More likely bigrams are *We are*, *We will*, *We the*, *We would*.... Knowing that the words *We need* have been written, *to* is the most likely word to come after them. Similarly, *the* is the most probable word to follow *all of*.

5.5.2 A Crash Program to Count Words with Unix

In his famous column, *Programming Pearls*, Bentley et al. (1986) posed the following problem:

> Given a text file and an integer k, print the k most common words in the file (and the number of their occurrences) in decreasing frequency.

Table 5.6 Ranking and generating words using trigrams (After Jelinek (1990))

Word	Rank	More likely alternatives
We	9	*The This One Two A Three Please In*
need	7	*are will the would also do*
to	1	
resolve	85	*have know do...*
all	9	*the this these problems...*
of	2	*the*
the	1	
important	657	*document question first...*
issues	14	*thing point to...*
within	74	*to of and in that...*
the	1	
next	2	*company*
two	5	*page exhibit meeting day*
days	5	*weeks years pages months*

This problem is especially interesting to us now as it is exactly the output of the first row in Table 5.6.

Bentley received two solutions for it: one from Donald Knuth, the prestigious inventor of TEX, and the second in the form of a comment from Doug McIlroy, the developer of Unix pipelines. While Knuth sent an 8-page program, McIlroy proposed a compelling Unix shell script of six lines. We reproduce it here (slightly modified):

1. `tr -cs 'A-Za-z' '\n' <input_file |`
   Tokenize the text in `input_file` using the Unix `tr` command. The Unix `tr` behaves like the Perl `tr` operator that we described in Sects. 2.4.5 and 5.3.3. There will be one word per line, and the output is passed to the next command.
2. `tr 'A-Z' 'a-z' |`
   Translate the uppercase characters into lowercase letters and pass the output to the next command.
3. `sort |`
   Sort the words. The identical words will be grouped together in adjacent lines.
4. `uniq -c |`
   Remove repeated lines. The identical adjacent lines will replaced with one single line. Each unique line in the output will be preceded by the count of its duplicates in the input file (`-c`).
5. `sort -rn |`
   Sort in the reverse (`-r`) numeric (`-n`) order. The most frequent words will be sorted first.
6. `head -5`
   Print the five first lines of the file (the five most frequent words).

The two first `tr` commands do not take into account possible accented characters. To correct it, we just need to modify the character list and include accents.

Nonetheless, we can apply the script as it is to English texts. On the novel *Nineteen Eighty-Four* (Orwell 1949), the output is:

```
6518 the
3491 of
2576 a
2442 and
2348 to
```

In addition, it is easy to extend the counts to bigrams. We need first to create a file, where each line contains a bigram: the words at index i and $i + 1$ on the same line separated with a blank. We use the Unix commands:

1. `tr -cs 'A-Za-z' '\n' <input_file | tr 'A-Za-z' > token_file` Tokenize the input and create a file with the unigrams.
2. `tail +2 < token_file > next_token_file`
   Create a second unigram file starting at the second word of the first tokenized file (`+2`).
3. `paste token_file next_token_file |`
   Merge the lines (the tokens) pairwise. Each line contains the words at index i and $i + 1$ separated with a tabulation.
4. And we count the bigrams as in the previous script.

5.5.3 *Counting Unigrams in Prolog*

As with Unix, counting unigrams in Prolog consists simply in tokenizing a text, sorting the words, and counting the number of times a type occurs in the corpus. We will not use the Prolog predefined `sort/2` predicate because it removes the duplicates. Instead, we can use a predicate implementing the quicksort algorithm or `msort/2` in some Prologs.

The predicate `count_duplicates/2` counts the duplicates. It takes a sorted list of words as input and returns a list of pairs with the frequency of each word `[N, Word]` in the output list:

```
count_duplicates(OrderedList, CountedList) :-
  count_duplicates(OrderedList, 1, [], CountedListRev),
  reverse(CountedListRev, CountedList).

count_duplicates([X, X | Ordered], N, Counting, Counted) :-
  N1 is N + 1,
  !,
  count_duplicates([X | Ordered], N1, Counting, Counted).
count_duplicates([X | Ordered], N, Counting, Counted) :-
  !,
  count_duplicates(Ordered, 1, [[N, X] | Counting], Counted).
count_duplicates([], _, L, L).
```

We get the unigrams with their counts with:

```
?- read_file(myFile, CharacterList),
   tokenize(CharacterList, TokenList),
   msort(TokenList, OrderedTokens),
   count_duplicates(OrderedTokens, UnigramList).
```

5.5.4 Counting Unigrams with Perl

Counting unigrams is straightforward and very fast with Perl. We can obtain them with the following algorithm:

1. Tokenize the text file.
2. Count the words using a hash table.
3. Possibly, sort the words according to their alphabetical order and numerical ranking.

For the first step, we apply any tokenizer from Sect. 5.3 and produce a tokenized file as output. We use the `split` function to assign each word of the text to the elements of an array. As we saw in Chap. 2, `split` takes two arguments: a regular expression, which describes a delimiter, and a string, which is split everywhere the delimiter matches. The resulting fragments are assigned sequentially to an array. Let `$text` be a big string containing the whole text with one word per line. The instruction:

```
@words = split(/\n/, $text);
```

assigns the first line and hence the first word to `$words[0]`, the second word to `$words[1]`, and so on. A useful generalization of this instruction is

```
@words = split(/\s+/, $text);
```

which splits the text at each sequence of white space characters.

Then, we use a hash table or associative array. Instead of being indexed by consecutive numbers, as in classical arrays, hash tables are indexed by strings. The next three lines

```
$wordcount{"a"} = 21;
$wordcount{"And"} = 10;
$wordcount{"the"} = 18;
```

create the hash table `$wordcount` with three indices called the keys: `a`, `And`, `the`, whose values are 21, 10, and 18. Hash keys can be numbers as well as strings. We refer to the whole array using the notation `%wordcount`. The instruction `keys` returns the keys of the array as in

```
keys %wordcount
```

A hash entry is created when a value is assigned to it. Its existence can be tested using the `exists` Boolean function.

The counting program scans the `@words` array and increments the frequency of the words as they occur. We finally introduce two new instructions and functions. The instruction `foreach item (list)` iterates over the items of an array, and `sort(array)` returns a sorted array. The complete program is:

```
$text = <>;
while ($line = <>) {
  $text .= $line;
}
$text =~ s/\n+/\n/g;
@words = split(/\n/, $text);
for ($i = 0; $i <= $#words; $i++) {
  if (!exists($frequency{$words[$i]})) {
    $frequency{$words[$i]} = 1;
  } else {
    $frequency{$words[$i]}++;
  }
}
foreach $word (sort keys %frequency){
  print "$frequency{$word} $word\n";
}
```

5.5.5 *Counting Bigrams with Perl*

We count bigrams and *n*-grams just as we did with unigrams. The only difference is that we create an array of bigrams by concatenating the adjacent words. The input is a tokenized file, and the following Perl program enables us to obtain them:

```
$text = <>;
while ($line = <>) {
  $text .= $line;
}
$text =~ s/\n+/\n/g;
@words = split(/\n/, $text);
for ($i = 0; $i < $#words; $i++) {
  $bigrams[$i] = $words[$i] . " " . $words[$i + 1];
}
for ($i = 0; $i < $#bigrams; $i++) {
  if (!exists($frequency_bigrams{$bigrams[$i]})) {
    $frequency_bigrams{$bigrams[$i]} = 1;
  } else {
```

```
      $frequency_bigrams{$bigrams[$i]}++;
    }
}
foreach $bigram (sort keys %frequency_bigrams) {
  print "$frequency_bigrams{$bigram} $bigram \n";
}
```

5.6 Probabilistic Models of a Word Sequence

5.6.1 The Maximum Likelihood Estimation

We observed in Table 5.6 that some word sequences are more likely than others. Using a statistical model, we can quantify these observations. The model will enable us to assign a probability to a word sequence as well as to predict the next word to follow the sequence.

Let $S = w_1, w_2, \ldots, w_i, \ldots, w_n$ be a word sequence. Given a training corpus, an intuitive estimate of the probability of the sequence, $P(S)$, is the relative frequency of the string $w_1, w_2, \ldots, w_i, \ldots, w_n$ in the corpus. This estimate is called the *maximum likelihood estimate* (MLE):

$$P_{\text{MLE}}(S) = \frac{C(w_1, \ldots, w_n)}{N},$$

where $C(w_1, \ldots, w_n)$ is the frequency or count of the string $w_1, w_2, \ldots, w_i, \ldots, w_n$ in the corpus, and N is the total number of strings of length n.

Most of the time, however, it is impossible to obtain this estimate. Even when corpora reach billions of words, they have a limited size, and it is unlikely that we can always find the exact sequence we are searching. We can try to simplify the computation and decompose $P(S)$ a step further using the chain rule as:

$$\begin{aligned} P(S) &= P(w_1, \ldots, w_n), \\ &= P(w_1)P(w_2|w_1)P(w_3|w_1, w_2)\ldots P(w_n|w_1, \ldots, w_{n-1}), \\ &= \prod_{i=1}^{n} P(w_i|w_1, \ldots, w_{i-1}). \end{aligned}$$

The probability $P(It, was, a, bright, cold, day, in, April)$ from *Nineteen Eighty-Four* by George Orwell corresponds then to the probability of having *It* to begin the sentence, then *was* knowing that we have *It* before, then *a* knowing that we have *It was* before, and so on, until the end of the sentence. It yields the product of conditional probabilities:

$$\begin{aligned} P(S) = {} & P(It) \times P(was|It) \times P(a|It, was) \times P(bright|It, was, a) \times \ldots \\ & \times P(April|It, was, a, bright, \ldots, in). \end{aligned}$$

To estimate $P(S)$, we need to know unigram, bigram, trigram, so far, so good, but also 4-gram, 5-gram, and even 8-gram statistics. Of course, no corpus is big enough to produce them. A practical solution is then to limit the n-gram length to 2 or 3, and thus to approximate them to bigrams:

$$P(w_i|w_1, w_2, \ldots, w_{i-1}) \approx P(w_i|w_{i-1}),$$

or trigrams:

$$P(w_i|w_1, w_2, \ldots, w_{i-1}) \approx P(w_i|w_{i-2}, w_{i-1}).$$

Using a trigram language model, $P(S)$ is approximated as:

$$P(S) \approx P(It) \times P(Pwas|It) \times P(a|It, was) \times P(bright|was, a) \times \ldots \\ \times P(April|day, in).$$

Using a bigram grammar, the general case of a sentence probability is:

$$P(S) \approx P(w_1) \prod_{i=2}^{n} P(w_i|w_{i-1}),$$

with the estimate

$$P_{\text{MLE}}(w_i|w_{i-1}) = \frac{C(w_{i-1}, w_i)}{\sum_w C(w_{i-1}, w)} = \frac{C(w_{i-1}, w_i)}{C(w_{i-1})}.$$

Similarly, the trigram maximum likelihood estimate is:

$$P_{\text{MLE}}(w_i|w_{i-2}, w_{i-1}) = \frac{C(w_{i-2}, w_{i-1}, w_i)}{C(w_{i-2}, w_{i-1})}.$$

And the general case of n-gram estimation is:

$$P_{\text{MLE}}(w_{i+n}|w_{i+1}, \ldots, w_{i+n-1}) = \frac{C(w_{i+1}, \ldots, w_{i+n})}{\sum_w C(w_{i+1}, \ldots, w_{i+n-1}, w)}, \\ = \frac{C(w_{i+1}, \ldots, w_{i+n})}{C(w_{i+1}, \ldots, w_{i+n-1})}.$$

As the probabilities we obtain are usually very low, it is safer to represent them as a sum of logarithms in practical applications. We will then use:

$$\log P(S) \approx \log P(w_1) + \sum_{i=2}^{n} \log P(w_i|w_{i-1}),$$

instead of $P(S)$. Nonetheless, in the following sections, as our example corpus is very small, we will compute the probabilities using products.

5.6.2 Using ML Estimates with Nineteen Eighty-Four

Training and Testing the Language Model

Before computing the probability of a word sequence, we must train the language model. The corpus used to derive the n-gram frequencies is classically called the **training set**, and the corpus on which we apply the model, the **test set**. Both sets should be distinct. If we apply a language model to a word sequence, which is part of the training corpus, its probability will be biased to a higher value, and thus will be inaccurate. The training and test sets can be balanced or not, depending on whether we want them to be specific of a task or more general.

For some models, we need to optimize parameters in order to obtain the best results. Again, it would bias the results if at the same time, we carry out the optimization on the test set and run the evaluation on it. For this reason some models need a separate **development set** to fine-tune their parameters.

In some cases, especially with small corpora, a specific division between training and test sets may have a strong influence on the results. It is then preferable to apply the training and testing procedure several times with different sets and average the results. The method is to randomly divide the corpus into two sets. We learn the parameters from the training set, apply the model to the test set, and repeat the process with a new random division, for instance, ten times. This method is called **cross-validation**, or ten-fold cross-validation if we repeat it ten times. Cross-validation smoothes the impact of a specific partition of the corpus.

Marking up the Corpus

Most corpora use some sort of markup language. The most common markers of N-gram models are the sentence delimiters `<s>` to mark the start of a sentence and `</s>` at its end. For example:

`<s>` *It was a bright cold day in April* `</s>`

Depending on the application, both symbols can be counted in the n-gram frequencies just as the other tokens or can be considered as context cues. Context cues are vocabulary items that appear in the condition part of the probability but are never predicted – they never occur in the right part. In many models, `<s>` is a context cue and `</s>` is part of the vocabulary. We will adopt this convention in the next examples.

The Vocabulary

We have defined language models that use a predetermined and finite set of words. This is never the case in reality, and the models will have to handle out-of-vocabulary (OOV) words. Training corpora are typically of millions, or even billions, of words. However, whatever the size of a corpus, it will never have a complete coverage of the vocabulary. Some words that are unseen in the training corpus are likely to occur in the test set. In addition, frequencies of rare words will not be reliable.

There are two main types of methods to deal with OOV words:

- The first method assumes a **closed vocabulary**. All the words both in the training and the test sets are known in advance. Depending on the language model settings, any word outside the vocabulary will be discarded or cause an error. This method is used in some applications, like voice control of devices.
- The **open vocabulary** makes provisions for new words to occur with a specific symbol, `<UNK>`, called the unknown token. All the OOV words are mapped to `<UNK>`, both in the training and test sets.

The vocabulary itself can come from an external dictionary. It can also be extracted directly from the training set. In this case, it is common to exclude the rare words, notably those seen only once – the *hapax legomena*. The vocabulary will then consist of the most frequent types of the corpus, for example, the 20,000 most frequent types. The other words, unseen or with a frequency lower than a cutoff value, 1, 2, or up to 5, will be mapped to `<UNK>`.

Computing a Sentence Probability

We trained a bigram language model on a very small corpus consisting of the three chapters of *Nineteen Eighty-Four*. We kept the appendix, "The Principles of Newspeak," as the test set and we selected this sentence from it:

> `<s>` *A good deal of the literature of the past was, indeed, already being transformed in this way* `</s>`

We first normalized the text: we created a file with one sentence per line. We inserted automatically the delimiters `<s>` and `</s>`. We removed the punctuation, parentheses, quotes, stars, dashes, tabulations, and double white spaces. We set all the words in lowercase letters. We counted the words, and we produced a file with the unigram and bigram counts.

The training corpus has 115,212 words; 8,635 types, including 3,928 hapax legomena; and 49,524 bigrams, where 37,365 bigrams have a frequency of 1. Table 5.7 shows the unigram and bigram frequencies for the words of the test sentence.

All the words of the sentence have been seen in the training corpus, and we can compute a probability estimate of it using the unigram relative frequencies:

Table 5.7 Frequencies of unigrams and bigrams. We excluded the <s> symbols from the word counts

w_i	$C(w_i)$	# words	$P_{\text{MLE}}(w_i)$	w_{i-1}, w_i	$C(w_{i-1}, w_i)$	$C(w_{i-1})$	$P_{\text{MLE}}(w_i \mid w_{i-1})$
<s>	7,072	–	–	–	–		
a	2,482	108,140	0.023	<s> *a*	133	7,072	0.019
good	53	108,140	0.00049	*a good*	14	2,482	0.006
deal	5	108,140	4.62×10^{-5}	*good deal*	0	53	0.0
of	3,310	108,140	0.031	*deal of*	1	5	0.2
the	6,248	108,140	0.058	*of the*	742	3,310	0.224
literature	7	108,140	6.47×10^{-5}	*the literature*	1	6,248	0.00016
of	3,310	108,140	0.031	*literature of*	3	7	0.429
the	6,248	108,140	0.058	*of the*	742	3,310	0.224
past	99	108,140	0.00092	*the past*	70	6,248	0.011
was	2,211	108,140	0.020	*past was*	4	99	0.040
indeed	17	108,140	0.00016	*was indeed*	0	2,211	0.0
already	64	108,140	0.00059	*indeed already*	0	17	0.0
being	80	108,140	0.00074	*already being*	0	64	0.0
transformed	1	108,140	9.25×10^{-6}	*being transformed*	0	80	0.0
in	1,759	108,140	0.016	*transformed in*	0	1	0.0
this	264	108,140	0.0024	*in this*	14	1,759	0.008
way	122	108,140	0.0011	*this way*	3	264	0.011
</s>	7,072	108,140	0.065	*way* </s>	18	122	0.148

$$P(S) \approx P(a) \times P(good) \times \ldots \times P(way) \times P(\text{</s>}),$$
$$\approx 3.67 \times 10^{-48}.$$

As $P(\text{<s>})$ is a constant that would scale all the sentences by the same factor, whether we use unigrams or bigrams, we excluded it from the $P(S)$ computation.

The bigram estimate is defined as:

$$P(S) \approx P(a|\text{<s>}) \times P(good|a) \times \ldots \times P(way|this) \times P(\text{</s>}|way).$$

and has a zero probability. This is due to **sparse data**: the fact that the corpus is not big enough to have all the bigrams covered with a realistic estimate. We shall see in the next section how to handle them.

5.7 Smoothing *N*-Gram Probabilities

5.7.1 Sparse Data

The approach using the maximum likelihood estimation has an obvious disadvantage because of the unavoidably limited size of the training corpora. Given

a vocabulary of 20,000 types, the potential number of bigrams is $20{,}000^2 = 400{,}000{,}000$, and with trigrams, it amounts to the astronomic figure of $20{,}000^3 = 8{,}000{,}000{,}000{,}000$. No corpus yet has the size to cover the corresponding word combinations.

Among the set of potential n-grams, some are almost impossible, except as random sequences generated by machines; others are simply unseen in the corpus. This phenomenon is referred to as **sparse data**, and the maximum likelihood estimator gives no hint on how to estimate their probability.

In this section, we introduce **smoothing** techniques to estimate probabilities of unseen n-grams. As the sum of probabilities of all the n-grams of a given length is 1, smoothing techniques also have to rearrange the probabilities of the observed n-grams. Smoothing allocates a part of the probability mass to the unseen n-grams that, as a counterpart, it shifts – or **discounts** – from the other n-grams.

5.7.2 *Laplace's Rule*

Laplace's rule (Laplace 1820, p. 17) is probably the oldest published method to cope with sparse data. It just consists in adding one to all the counts. For this reason, some authors also call it the add-one method.

Laplace wanted to estimate the probability of the sun to rise tomorrow and he imagined this rule: he set both event counts, rise and not rise, arbitrarily to one, and he incremented them with the corresponding observations. From the beginning of time, humans had seen the sun rise every day. Laplace derived the frequency of this event from what he believed to be the oldest epoch of history: five thousand years or 1,826,213 days. As nobody observed the sun not rising, he obtained the chance for the sun to rise tomorrow of 1,826,214 to 1.

Laplace's rule states that the frequency of unseen n-grams is equal to 1 and the general estimate of a bigram probability is:

$$P_{\text{Laplace}}(w_i|w_{i-1}) = \frac{C(w_{i-1}, w_i) + 1}{\sum_w (C(w_{i-1}, w) + 1)} = \frac{C(w_{i-1}, w_i) + 1}{C(w_{i-1}) + Card(V)},$$

where $Card(V)$ is the number of word types. The denominator correction is necessary to have the probability sum equal to 1.

With Laplace's rule, we can use bigrams to compute the sentence probability (Table 5.8):

$$\begin{aligned} P_{\text{Laplace}}(S) &\approx P(a|\texttt{<s>}) \times P(good|a) \times \ldots \times P(\texttt{</s>}|way), \\ &\approx 4.62 \times 10^{-57}. \end{aligned}$$

Laplace's method is easy to understand and implement. It has an obvious drawback however: it shifts an enormous mass of probabilities to the unseen

Table 5.8 Frequencies of bigrams using Laplace's rule

w_{i-1}, w_i	$C(w_{i-1}, w_i) + 1$	$C(w_{i-1}) + Card(V)$	$P_{\text{Lap}}(w_i \mid w_{i-1})$
<s> *a*	133 + 1	7,072 + 8,635	0.0085
a good	14 + 1	2,482 + 8,635	0.0013
good deal	0 + 1	53 + 8,635	0.00012
deal of	1 + 1	5 + 8,635	0.00023
of the	742 + 1	3,310 + 8,635	0.062
the literature	1 + 1	6,248 + 8,635	0.00013
literature of	3 + 1	7 + 8,635	0.00046
of the	742 + 1	3,310 + 8,635	0.062
the past	70 + 1	6,248 + 8,635	0.0048
past was	4 + 1	99 + 8,635	0.00057
was indeed	0 + 1	2,211 + 8,635	0.000092
indeed already	0 + 1	17 + 8,635	0.00012
already being	0 + 1	64 + 8,635	0.00011
being transformed	0 + 1	80 + 8,635	0.00011
transformed in	0 + 1	1 + 8,635	0.00012
in this	14 + 1	1,759 + 8,635	0.0014
this way	3 + 1	264 + 8,635	0.00045
way </s>	18 + 1	122 + 8,635	0.0022

n-grams and gives them a considerable importance. The frequency of the unlikely bigram *the of* will be 1, a quarter of the much more common *this way*.

The **discount** value is the ratio between the smoothed frequencies and their actual counts in the corpus. The bigram *this way* has been discounted by $0.011/0.00045 = 24.4$ to make place for the unseen bigrams. This is unrealistic and shows the major drawback of this method. For this small corpus, Laplace's rule applied to bigrams has a result opposite to what we wished. It has not improved the sentence probability over the unigrams. This would mean that a bigram language model is worse than words occurring randomly in the sentence.

If adding 1 is too much, why not try less, for instance, 0.5? This is the idea of Lidstone's rule. This value is denoted λ. The new formula is then:

$$P_{\text{Lidstone}}(w_i \mid w_{i-1}) = \frac{C(w_{i-1}, w_i) + \lambda}{C(w_{i-1}) + \lambda Card(V)},$$

which, however, is not a big improvement.

5.7.3 Good–Turing Estimation

The Good–Turing estimation (Good 1953) is one of the most efficient smoothing methods. As with Laplace's rule, it reestimates the counts of the n-grams observed in the corpus by discounting them, and it shifts the probability mass it has shaved

to the unseen bigrams. The discount factor is variable, however, and depends on the number of times a n-gram has occurred in the corpus. There will be a specific discount value to n-grams seen once, another one to bigrams seen twice, a third one to those seen three times, and so on.

Let us denote N_c the number of n-grams that occurred exactly c times in the corpus. N_0 is the number of unseen n-grams, N_1 the number of n-grams seen once, N_2 the number of n-grams seen twice, and so on. If we consider bigrams, the value N_0 is $Card(V)^2$ minus all the bigrams we have seen.

The Good–Turing method reestimates the frequency of n-grams occurring c times using the formula:

$$c^* = (c+1)\frac{E(N_{c+1})}{E(N_c)},$$

where $E(x)$ denotes the expectation of the random variable x. This formula is usually approximated as:

$$c^* = (c+1)\frac{N_{c+1}}{N_c}.$$

To understand how this formula was designed, let us take the example of the unseen bigrams with $c = 0$. Let us suppose that we draw a sequence of bigrams to build our training corpus, and the last bigram we have drawn was unseen before. From this moment, there is one occurrence of it in the training corpus and the count of bigrams in the same case is N_1. Using the maximum likelihood estimation, the probability to draw such an unseen bigram is then the count of bigrams seen once divided by the total count of the bigrams seen so far: N_1/N. We obtain the probability to draw one specific unseen bigram by dividing this term by the count of unseen bigrams:

$$\frac{1}{N} \times \frac{N_1}{N_0}.$$

Hence, the Good–Turing reestimated count of an unseen n-gram is $c^* = \frac{N_1}{N_0}$. Similarly, we would have $c^* = \frac{2N_2}{N_1}$ for an n-gram seen once in the training corpus.

The three chapters in *Nineteen Eighty-Four* contain 37,365 unique bigrams and 5,820 bigrams seen twice. Its vocabulary of 8,635 words generates $8{,}635^2 = 74{,}563{,}225$ bigrams, of which 74,513,701 are unseen. The Good–Turing method reestimates the frequency of each unseen bigram to $37{,}365/74{,}513{,}701 = 0.0005$, and unique bigrams to $2 \times (5{,}820/37{,}365) = 0.31$. Table 5.9 shows the complete the reestimated frequencies for the n-grams up to 9.

In practice, only high values of N_c are reliable, which correspond to low values of c. In addition, above a certain threshold, most frequencies of frequency will be

Table 5.9 The reestimated frequencies of the bigrams

Frequency of occurrence	N_c	c^*
0	74,513,701	0.0005
1	37,365	0.31
2	5,820	1.09
3	2,111	2.02
4	1,067	3.37
5	719	3.91
6	468	4.94
7	330	6.06
8	250	6.44
9	179	8.94

equal to zero. Therefore, the Good–Turing estimation is applied for $c < k$, where k is a constant set to 5, 6, ..., or 10. Other counts are not reestimated. See Katz (1987) for the details.

The probability of a n-gram is given by the formula:

$$P_{\mathrm{GT}}(w_1, \ldots, w_n) = \frac{c^*(w_1, \ldots, w_n)}{N},$$

where c^* is the reestimated count of $w_1 \ldots w_n$, and N the original count of n-grams in the corpus. The conditional frequency is

$$P_{\mathrm{GT}}(w_n | w_1, \ldots, w_{n-1}) = \frac{c^*(w_1, \ldots, w_n)}{C(w_1, \ldots, w_{n-1})}.$$

Table 5.10 shows the conditional probabilities, where only frequencies less than 10 have been reestimated. The sentence probability using bigrams is 2.56×10^{-50}. This is better than with Laplace's rule, but as the corpus is very small, still greater than the unigram probability.

5.8 Using N-Grams of Variable Length

In the previous section, we used smoothing techniques to reestimate the probability of n-grams of constant length, whether they occurred in the training corpus or not. A property of these techniques is that they assign the same probability to all the unseen n-grams.

Another strategy is to rely on the frequency of observed sequences but of lesser length: $n-1$, $n-2$, and so on. As opposed to smoothing, the estimate of each unseen n-gram will be specific to the words it contains. In this section, we introduce two techniques: the linear interpolation and Katz's back-off model.

Table 5.10 The conditional frequencies using the Good–Turing method. We have not reestimated the frequencies when they are greater than 9

w_{i-1}, w_i	$C(w_{i-1}, w_i)$	$c^*(w_{i-1}, w_i)$	$C(w_{i-1})$	$P_{\text{GT}}(w_i \mid w_{i-1})$
<s> *a*	133	133	7,072	0.019
a good	14	14	2,482	0.006
good deal	0	0.0005	53	9.46×10^{-6}
deal of	1	0.31	5	0.062
of the	742	742	3,310	0.224
the literature	1	0.31	6,248	4.99×10^{-5}
literature of	3	2.02	7	0.29
of the	742	742	3,310	0.224
the past	70	70	6,248	0.011
past was	4	3.37	99	0.034
was indeed	0	0.0005	2,211	2.27×10^{-7}
indeed already	0	0.0005	17	2.95×10^{-5}
already being	0	0.0005	64	7.84×10^{-6}
being transformed	0	0.0005	80	6.27×10^{-6}
transformed in	0	0.0005	1	0.00050
in this	14	14	1,759	0.008
this way	3	2.02	264	0.0077
way </s>	18	18	122	0.148

5.8.1 Linear Interpolation

Linear interpolation, also called deleted interpolation (Jelinek and Mercer 1980), combines linearly the maximum likelihood estimates from length 1 to n. For trigrams, it corresponds to:

$$P_{\text{Interpolation}}(w_n \mid w_{n-2}, w_{n-1}) = \lambda_3 P_{\text{MLE}}(w_n \mid w_{n-2}, w_{n-1}) + \lambda_2 P_{\text{MLE}}(w_n \mid w_{n-1}) + \lambda_1 P_{\text{MLE}}(w_n),$$

where $0 \leq \lambda_i \leq 1$ and $\sum_{i=1}^{3} \lambda_i = 1$.

The values can be constant and set by hand, for instance, $\lambda_3 = 0.6$, $\lambda_2 = 0.3$, and $\lambda_1 = 0.1$. They can also be trained and optimized from a corpus (Jelinek 1997).

Table 5.11 shows the interpolated probabilities of bigrams with $\lambda_2 = 0.7$ and $\lambda_1 = 0.3$. The sentence probability using these interpolations is 9.46×10^{-45}.

We can now understand why bigram *we the* is ranked so high in Table 5.6 after *we are* and *we will*. Although it can occur in English, as in the American constitution, *We the people*..., it is not a very frequent combination. In fact, the estimation has been obtained with an interpolation where the term $\lambda_1 P_{\text{MLE}}(the)$ boosted the bigram to the top because of the high frequency of *the*.

Table 5.11 Interpolated probabilities of bigrams using the formula $\lambda_2 P_{\text{MLE}}(w_i|w_{i-1}) + \lambda_1 P_{\text{MLE}}(w_i)$, $\lambda_2 = 0.7$, and $\lambda_1 = 0.3$. The total number of words is 108,140

w_{i-1}, w_i	$C(w_{i-1}, w_i)$	$C(w_{i-1})$	$P_{\text{MLE}}(w_i\|w_{i-1})$	$P_{\text{MLE}}(w_i)$	$P_{\text{Interp}}(w_i\|w_{i-1})$
<s> *a*	133	7,072	0.019	0.023	0.020
a good	14	2,482	0.006	0.00049	0.0041
good deal	0	53	0.0	4.62×10^{-5}	1.38×10^{-5}
deal of	1	5	0.2	0.031	0.149
of the	742	3,310	0.224	0.058	0.174
the literature	1	6,248	0.00016	6.47×10^{-5}	0.000131
literature of	3	7	0.429	0.031	0.309
of the	742	3,310	0.224	0.058	0.174
the past	70	6,248	0.011	0.00092	0.00812
past was	4	99	0.040	0.020	0.0344
was indeed	0	2,211	0.0	0.00016	4.71×10^{-5}
indeed already	0	17	0.0	0.00059	0.000177
already being	0	64	0.0	0.00074	0.000222
being transformed	0	80	0.0	9.25×10^{-6}	2.77×10^{-6}
transformed in	0	1	0.0	0.016	0.00488
in this	14	1,759	0.008	0.0024	0.0063
this way	3	264	0.011	0.0011	0.00829
way </s>	18	122	0.148	0.065	0.123

5.8.2 Back-Off

The idea of the back-off model is to use the frequency of the longest available n-grams, and if no n-gram is available to back off to the $(n-1)$-grams, and then to $(n-2)$-grams, and so on. If n equals 3, we first try trigrams, then bigrams, and finally unigrams. For a bigram language model, the back-off probability can be expressed as:

$$P_{\text{Backoff}}(w_i|w_{i-1}) = \begin{cases} P(w_i|w_{i-1}), & \text{if } C(w_{i-1}, w_i) \neq 0, \\ \alpha P(w_i), & \text{otherwise.} \end{cases}$$

So far, this model does not tell us how to estimate the n-gram probabilities to the right of the formula. A first idea would be to use the maximum likelihood estimate for bigrams and unigrams. With $\alpha = 1$, this corresponds to:

$$P_{\text{Backoff}}(w_i|w_{i-1}) = \begin{cases} P_{\text{MLE}}(w_i|w_{i-1}) = \dfrac{C(w_{i-1}, w_i)}{C(w_{i-1})}, & \text{if } C(w_{i-1}, w_i) \neq 0, \\ P_{\text{MLE}}(w_i) = \dfrac{C(w_i)}{\#\text{words}}, & \text{otherwise.} \end{cases}$$

and Table 5.12 shows the probability estimates we can derive from our small corpus. They yield a sentence probability of 2.11×10^{-40} for our example.

Table 5.12 Probability estimates using an elementary backoff technique

w_{i-1}, w_i	$C(w_{i-1}, w_i)$		$C(w_i)$	$P_{\text{Backoff}}(w_i \mid w_{i-1})$
<s>			7,072	—
<s> *a*	133		2,482	0.019
a good	14		53	0.006
good deal	0	backoff	5	4.62×10^{-5}
deal of	1		3,310	0.2
of the	742		6,248	0.224
the literature	1		7	0.00016
literature of	3		3,310	0.429
of the	742		6,248	0.224
the past	70		99	0.011
past was	4		2,211	0.040
was indeed	0	backoff	17	0.00016
indeed already	0	backoff	64	0.00059
already being	0	backoff	80	0.00074
being transformed	0	backoff	1	9.25×10^{-6}
transformed in	0	backoff	1,759	0.016
in this	14		264	0.008
this way	3		122	0.011
way </s>	18		7,072	0.148

This back-off technique is relatively easy to implement and Brants et al. (2007) applied it to 5-grams on a corpus of three trillion tokens with a back-off factor $\alpha = 0.4$. They used the recursive definition:

$$P_{\text{Backoff}}(w_i \mid w_{i-k}, \ldots, w_{i-1}) = \begin{cases} P_{\text{MLE}}(w_i \mid w_{i-k}, \ldots, w_{i-1}), & \text{if } C(w_{i-k}, \ldots, w_i) \neq 0, \\ \alpha P_{\text{Backoff}}(w_i \mid w_{i-k+1}, \ldots, w_{i-1}), & \text{otherwise.} \end{cases}$$

However, the result is not a probability as the sum of all the probabilities, $\sum_{w_i} P(w_i \mid w_{i-1})$, can be greater than 1. In the next section, we describe Katz's (1987) back-off model that provides an efficient and elegant solution to this problem.

5.8.3 Katz's Back-Off Model

As with linear interpolation in Sect. 5.8.1, back-off combines n-grams of variable length while keeping a probability sum of 1. This means that for a bigram language model, we need to discount the bigram estimates to make room for the unigrams and then weight these unigrams to ensure that the sum of probabilities is equal to 1.

This is precisely the definition of Katz's model, where Katz (1987) replaced the maximum likelihood estimates for bigrams with Good–Turing's estimates:

$$P_{\text{Katz}}(w_i|w_{i-1}) = \begin{cases} \tilde{P}(w_i|w_{i-1}), & \text{if } C(w_{i-1}, w_i) \neq 0, \\ \alpha P(w_i), & \text{otherwise.} \end{cases}$$

We first use the Good–Turing estimates to discount the observed bigrams,

$$\tilde{P}(w_i|w_{i-1}) = \frac{c^*(w_{i-1}, w_i)}{C(w_{i-1})},$$

for instance, with the values in Tables 5.9 and 5.10 for our sentence. We then assign the remaining probability mass to the unigrams.

To compute α, we add the two terms of Katz's back-off model, the discounted probabilities of the observed bigrams, and, for the unseen bigrams, the weighted unigram probabilities:

$$\begin{aligned} \sum_{w_i} P_{\text{Katz}}(w_i|w_{i-1}) &= \sum_{w_i, C(w_{i-1}, w_i)>0} \tilde{P}(w_i|w_{i-1}) + \alpha \sum_{w_i, C(w_{i-1}, w_i)=0} P_{\text{MLE}}(w_i), \\ &= 1. \end{aligned}$$

We know that this sum equals 1, and we derive α from it:

$$\alpha = \alpha(w_{i-1}) = \frac{1 - \sum_{w_i, C(w_{i-1}, w_i)>0} \tilde{P}(w_i|w_{i-1})}{\sum_{w_i, C(w_{i-1}, w_i)=0} P_{\text{MLE}}(w_i)}.$$

For trigrams or n-grams of higher order, we apply Katz's model recursively:

$$P_{\text{Katz}}(w_i|w_{i-2}, w_{i-1}) = \begin{cases} \tilde{P}(w_i|w_{i-2}, w_{i-1}), & \text{if } C(w_{i-2}, w_{i-1}, w_i) \neq 0, \\ \alpha(w_{i-2}, w_{i-1}) P_{\text{Katz}}(w_i|w_{i-1}), & \text{otherwise.} \end{cases}$$

5.9 Industrial *N*-Grams

The Internet made it possible to put together collections of n-gram of a size unimaginable a few years ago. Examples of such collections include the Google n-grams (Franz and Brants 2006) and Microsoft Web n-gram service (Huang et al. 2010; Wang et al. 2010).

The Google n-grams were extracted from a corpus of one trillion words and include unigram, bigram, trigram, 4-gram, and 5-gram counts. The excerpt below shows an example of trigram counts:

```
ceramics collectables collectibles 55
ceramics collectables fine 130
ceramics collected by 52
ceramics collectible pottery 50
ceramics collectibles cooking 45
ceramics collection , 144
ceramics collection . 247
ceramics collection </S> 120
ceramics collection and 43
```

Both companies, Google and Microsoft, use these n-grams in a number of applications and made them available to the public as well.

5.10 Quality of a Language Model

5.10.1 Intuitive Presentation

We can compute the probability of sequences of any length or of whole texts. As each word in the sequence corresponds to a conditional probability less than 1, the product will naturally decrease with the length of the sequence. To make sense, we normally average it by the number of words in the sequence and extract its nth root. This measure, which is a sort of a per-word probability of a sequence L, is easier to compute using a logarithm:

$$H(L) = -\frac{1}{n} \log_2 P(w_1, \ldots, w_n).$$

We have seen that trigrams are better predictors than bigrams, which are better than unigrams. This means that the probability of a very long sequence computed with a bigram model will normally be higher than with a unigram one. The log measure will then be lower.

Intuitively, this means that the $H(L)$ measure will be a quality marker for a language model where lower numbers will correspond to better models. This intuition has mathematical foundations, as we will see in the two next sections.

5.10.2 Entropy Rate

We used entropy with characters in Chap. 3. We can use it with any symbols such as words, bigrams, trigrams, or any n-grams. When we normalize it by the length of the word sequence, we define the **entropy rate**:

$$H(L) = -\frac{1}{n} \sum_{w_1,\ldots,w_n \in L} p(w_1,\ldots,w_n) \log_2 p(w_1,\ldots,w_n),$$

where L is the set of all possible sequences of length n.

It has been proven that when $n \to \infty$ or n is very large and under certain conditions, we have

$$\begin{aligned} H(L) &= \lim_{n\to\infty} -\frac{1}{n} \sum_{w_1,\ldots,w_n \in L} p(w_1,\ldots,w_n) \log_2 p(w_1,\ldots,w_n), \\ &= \lim_{n\to\infty} -\frac{1}{n} \log_2 p(w_1,\ldots,w_n), \end{aligned}$$

which means that we can compute $H(L)$ from a very long sequence, ideally infinite, instead of summing of all the sequences of a definite length.

5.10.3 Cross Entropy

We can also use cross entropy, which is measured between a text, called the language and governed by an unknown probability p, and a language model m. Using the same definitions as in Chap. 3, the cross entropy of m on p is given by:

$$H(p,m) = -\frac{1}{n} \sum_{w_1,\ldots,w_n \in L} p(w_1,\ldots,w_n) \log_2 m(w_1,\ldots,w_n).$$

As for the entropy rate, it has been proven that, under certain conditions

$$\begin{aligned} H(p,m) &= \lim_{n\to\infty} -\frac{1}{n} \sum_{w_1,\ldots,w_n \in L} p(w_1,\ldots,w_n) \log_2 m(w_1,\ldots,w_n), \\ &= \lim_{n\to\infty} -\frac{1}{n} \log_2 m(w_1,\ldots,w_n). \end{aligned}$$

In applications, we generally compute the cross entropy on the complete word sequence of a test set, governed by p, using a bigram or trigram model, m, derived from a training set.

In Chap. 3, we saw the inequality $H(p) \leq H(p,m)$. This means that the cross entropy will always be an upper bound of $H(p)$. As the objective of a language model is to be as close as possible to p, the best model will be the one yielding the lowest possible value. This forms the mathematical background of the intuitive presentation in Sect. 5.10.1.

5.10.4 Perplexity

The perplexity of a language model is defined as:

$$PP(p, m) = 2^{H(p,m)}.$$

Perplexity is interpreted as the average *branching factor* of a word: the statistically weighted number of words that follow a given word. Perplexity is equivalent to entropy. The only advantage of perplexity is that it results in numbers more comprehensible for human beings. It is therefore more popular to measure the quality of language models. As is the case for entropy, the objective is to minimize it: the better the language model, the lower the perplexity.

5.11 Collocations

Collocations are recurrent combinations of words. Palmer (1933), one of the first to study them comprehensively, defined them as:

> succession[s] of two or more words that must be learnt as an integral whole and not pieced together from its component parts

or as *comings-together-of-words*. Collocations are ubiquitous and arbitrary in English, French, German, and other languages. Simplest collocations are fixed n-grams such as *The White House* and *Le Président de la République*. Other collocations involve some morphological or syntactic variation such as the one linking *make* and *decision* in American English: *to make a decision*, *decisions to be made*, *make an important decision*.

Collocations underlie word preferences that most of the time cannot easily be explained by a syntactic or semantic reasoning: they are merely resorting to usage. As a teacher of English in Japan, Palmer (1933) noted their importance for language learners. Collocations are in the mind of a native speaker. S/he can recognize them as valid. On the contrary, nonnative speakers may make mistakes when they are not aware of them or try to produce word-for-word translations. For this reason, many second language learners' dictionaries describe most frequent associations. In English, the *Oxford Advanced Learner's Dictionary*, *The Longman Dictionary of Contemporary English*, and *The Collins COBUILD* carefully list verbs and prepositions or particles commonly associated such as phrasal verbs *set up*, *set off*, and *set out*.

Lexicographers used to identify collocations by introspection and by observing corpora, at the risk of forgetting some of them. Statistical tests can automatically extract associated words or "sticky" pairs from raw corpora. We introduce three of these tests in this section together with programs in Perl to compute them.

Table 5.13 Collocates of *surgery* extracted from the Bank of English using the mutual information test. Note the misspelled word *pioneeing*

Word	Frequency	Bigram word + *surgery*	Mutual information
arthroscopic	3	3	11.822
pioneeing	3	3	11.822
reconstructive	14	11	11.474
refractive	6	4	11.237
rhinoplasty	5	3	11.085

5.11.1 Word Preference Measurements

Mutual Information

Mutual information (Church and Hanks 1990; Fano 1961) is a statistical measure that is widely used to quantify the strength of word associations.[1] Mutual information for the bigram w_i, w_j is defined as:

$$I(w_i, w_j) = \log_2 \frac{P(w_i, w_j)}{P(w_i)P(w_j)}.$$

Using the maximum likelihood estimate, this corresponds to:

$$I(w_i, w_j) = \log_2 \frac{N \cdot C(w_i, w_j)}{C(w_i)C(w_j)},$$

where $C(w_i)$ and $C(w_j)$ are, respectively, the frequencies of word w_i and word w_j in the corpus, $C(w_i, w_j)$ is the frequency of bigram w_i, w_j, and N is the total number of words in the corpus.

Instead of just bigrams, where $j = i + 1$, we can count the number of times the two words w_i and w_j occur together sufficiently close, but not necessarily adjacently. $C(w_i, w_j)$ is then the number of times the word w_i is followed of preceded by w_j in a window of k words, where k typically ranges from 1 to 10, or within a sentence.

Table 5.13 shows collocates of the word *surgery*. High mutual information tends to show pairs of words occurring together but generally with a lower frequency, such as technical terms.

[1] Some authors now use the term *pointwise mutual information* to mean *mutual information*. Neither Fano (1961) nor Church and Hanks (1990) used this term and we kept the original one.

Table 5.14 Collocates of set extracted from Bank of English using the *t*-score

Word	Frequency	Bigram *set* + word	*t*-score
up	134,882	5,512	67.980
a	1,228,514	7,296	35.839
to	1,375,856	7,688	33.592
off	52,036	888	23.780
out	12,3831	1,252	23.320

t-Scores

Given two words, the *t*-score (Church and Mercer 1993) compares the hypothesis that the words form a collocation with the *null hypothesis* that posits that the cooccurrence is only governed by chance, that is $P(w_i, w_j) = P(w_i) \times P(w_j)$.

The *t*-score computes the difference between the two hypotheses, respectively, $mean(P(w_i, w_j))$ and $mean(P(w_i))mean(P(w_j))$, and divides it by the variances. It is defined by the formula:

$$t(w_i, w_j) = \frac{mean(P(w_i, w_j)) - mean(P(w_i))mean(P(w_j))}{\sqrt{\sigma^2(P(w_i, w_j)) + \sigma^2(P(w_i)P(w_j))}}.$$

The hypothesis that w_i and w_j are a collocation gives us a mean of $\frac{C(w_i, w_j)}{N}$; with the null hypothesis, the mean product is $\frac{C(w_i)}{N} \times \frac{C(w_j)}{N}$; and using a binomial assumption, the denominator is approximated to $\sqrt{\frac{C(w_i, w_j)}{N^2}}$. We have then:

$$t(w_i, w_j) = \frac{C(w_i, w_j) - \frac{1}{N}C(w_i)C(w_j)}{\sqrt{C(w_i, w_j)}}.$$

Table 5.14 shows collocates of *set* extracted from the Bank of English using the *t*-score. High *t*-scores show recurrent combinations of grammatical or very frequent words such as *of the*, *and the*, etc. Church and Mercer (1993) hint at the threshold value of 2 or more.

Likelihood Ratio

Dunning (1993) criticized the *t*-score test and proposed an alternative measure based on binomial distributions and likelihood ratios. Assuming that the words have a binomial distribution, we can express the probability of having k counts of a word w in a sequence of N words knowing that w's probability is p as:

$$f(k; N, p) = \binom{N}{k} p^k (1-p)^{N-k},$$

where

$$\binom{N}{k} = \frac{N!}{k!(N-k)!}.$$

The formula reflects the probability of having k counts of a word w, p^k, and $N-k$ counts of not having w, $(1-p)^{N-k}$. The binomial coefficient $\binom{N}{k}$ corresponds to the number of different ways of distributing k occurrences of the word w in a sequence of N words.

In the case of collocations, rather than measuring the distribution of single words, we want to evaluate the likelihood of the $w_i w_j$ bigram distribution. To do this, we can reformulate the binomial formula considering the word preceding w_j, which can either be w_i or a different word that we denote $\neg w_i$.

Let n_1 be the count of w_i and k_1, the count of the bigram $w_i w_j$ in the word sequence (the corpus). Let n_2 be the count of $\neg w_i$, and k_2, the count of the bigram $\neg w_i w_j$, where $\neg w_i w_j$ denotes a bigram in which the first word is not w_i and the second word is w_j. Let p_1 be the probability of w_j knowing that we have w_i preceding it, and p_2 be the probability of w_j knowing that we have $\neg w_i$ before it. The binomial distribution of observing the pairs $w_i w_j$ and $\neg w_i w_j$ in our sequence is:

$$f(k_1; n_1, p_1) f(k_2; n_2, p_2) = \binom{n_1}{k_1} p_1^{k_1} (1-p_1)^{n_1-k_1} \binom{n_2}{k_2} p_2^{k_2} (1-p_2)^{n_2-k_2}.$$

The basic idea to evaluate the collocation strength of a bigram $w_i w_j$ is to test two hypotheses:

- The two words w_i and w_j are part of a collocation. In this case, we will have $p_1 = P(w_j|w_i) \neq p_2 = P(w_j|\neg w_i)$ (Dependence hypothesis, H_{dep}).
- The two words w_i and w_j occur independently. In this case, we will have $p_1 = P(w_j|w_i) = p_2 = P(w_j|\neg w_i) = P(w_j) = p$ (Independence hypothesis, H_{ind}).

The logarithm of the hypothesis ratio corresponds to:

$$\begin{aligned}
-2\log\lambda &= 2\log\frac{H_{dep}}{H_{ind}}, \\
&= 2\log\frac{f(k_1; n_1, p_1) f(k_2; n_2, p_2)}{f(k_1; n_1, p) f(k_2; n_2, p)}, \\
&= 2(\log f(k_1; n_1, p_1) + \log f(k_2; n_2, p_2) - \log f(k_1; n_1, p) \\
&\quad - \log f(k_2; n_2, p)),
\end{aligned}$$

where $k_1 = C(w_i, w_j)$, $n_1 = C(w_i)$, $k_2 = C(w_j) - C(w_i, w_j)$, $n_2 = N - C(w_i)$, and $\log f(k; N, p) = k \log p + (N-k)\log(1-p)$.

Table 5.15 A contingency table containing bigram counts, where $\neg w_i w_j$ represents bigrams in which the first word is not w_i and the second word is w_j. N is the number of words in the corpus

	$\mathbf{w_i}$	$\mathbf{\neg w_i}$
$\mathbf{w_j}$	$C(w_i, w_j)$	$C(\neg w_i, w_j) = C(w_j) - C(w_i, w_j)$
$\mathbf{\neg w_j}$	$C(w_i, \neg w_j) = C(w_i) - C(w_i, w_j)$	$C(\neg w_i, \neg w_j) = N - C(w_i, w_j)$

Using the counts in Table 5.15 and the maximum likelihood estimate, we have

$$\begin{aligned} p &= P(w_j) &&= \frac{C(w_j)}{N}, \\ p_1 &= P(w_j|w_i) &&= \frac{C(w_i, w_j)}{C(w_i)}, \text{ and} \\ p_2 &= P(w_j|\neg w_i) &&= \frac{C(w_j) - C(w_i, w_j)}{N - C(w_i)}, \end{aligned}$$

where N is the number of words in the corpus.

5.11.2 Extracting Collocations with Perl

Both programs use unigram and bigram statistics. To compute them, we must first tokenize the text, and count words and bigrams using the tools we have described before:

```
$text = <>;
while ($line = <>) {
  $text .= $line;
}
$text =~ s/\n+/\n/g;
@words = split(/\n/, $text);
for ($i = 0; $i < $#words; $i++) {
  $bigrams[$i] = $words[$i] . " " . $words[$i + 1];
}
for ($i = 0; $i <= $#words; $i++) {
  $frequency{$words[$i]}++;
}
for ($i = 0; $i < $#words; $i++) {
  $frequency_bigrams{$bigrams[$i]}++;
}
```

Finally, we must know the number of words in the corpus. This corresponds to the size of the `word` array: `$#word`.

Mutual Information

The Perl program iterates over the `word` array and applies the mutual information formula. The program is not optimal and computes the same value several times:

```
for ($i = 0; $i < $#words; $i++) {
  $mutual_info{$bigrams[$i]} = log(($#words + 1) *
    $frequency_bigrams{$bigrams[$i]}/
      ($frequency{$words[$i]} *
        $frequency{$words[$i + 1]}))/log(2);
}

foreach $bigram (keys %mutual_info){
  @bigram_array = split(/ /, $bigram);
  print $mutual_info{$bigram}, " ", $bigram, "\t",
  $frequency_bigrams{$bigram}, "\t",
  $frequency{$bigram_array[0]}, "\t",
  $frequency{$bigram_array[1]}, "\n";
}
```

t-Scores

The program is similar to the previous one except the formula:

```
for ($i = 0; $i < $#words; $i++) {
  $t_scores{$bigrams[$i]} =
    ($frequency_bigrams{$bigrams[$i]} -
      $frequency{$words[$i]} *
        $frequency{$words[$i + 1]}/($#words + 1))
         /sqrt($frequency_bigrams{$bigrams[$i]});
}

foreach $bigram (keys %t_scores ){
  @bigram_array = split(/ /, $bigram);
  print $t_scores{$bigram}, " ", $bigram, "\t",
  $frequency_bigrams{$bigram}, "\t",
  $frequency{$bigram_array[0]}, "\t",
  $frequency{$bigram_array[1]}, "\n";
}
```

Log Likelihood Ratio

The program is similar to the previous one except the formula:

```
for ($i = 0; $i < $#words; $i++) {
  $p = $frequency{$words[$i + 1]}/$#words;
  $p1 = $frequency_bigrams{$bigrams[$i]}/
    $frequency{$words[$i]};
  $p2 = ($frequency{$words[$i + 1]} -
    $frequency_bigrams{$bigrams[$i]})/
      ($#words - $frequency{$words[$i]});
  if (($p1 != 1) && ($p2 != 0)) {
    $likelihood_ratio{$bigrams[$i]} = 2*(
      $frequency_bigrams{$bigrams[$i]} * log($p1) +
      ($frequency{$words[$i]} -
      $frequency_bigrams{$bigrams[$i]}) * log(1 - $p1)
      + ($frequency{$words[$i + 1]} -
      $frequency_bigrams{$bigrams[$i]}) * log($p2) +
      ($#words - $frequency{$words[$i]} -
      $frequency{$words[$i + 1]} +
      $frequency_bigrams{$bigrams[$i]}) * log(1 - $p2)
      - $frequency_bigrams{$bigrams[$i]} * log($p) +
     ($frequency{$words[$i]} -
     $frequency_bigrams{$bigrams[$i]}) * log(1 - $p)
     - ($frequency{$words[$i + 1]} -
     $frequency_bigrams{$bigrams[$i]}) * log($p) +
     ($#words - $frequency{$words[$i]} -
     $frequency{$words[$i + 1]} +
     $frequency_bigrams{$bigrams[$i]}) *
     log(1 - $p));
  }
}

foreach $bigram (keys %likelihood_ratio ){
  @bigram_array = split(/ /, $bigram);
  print $likelihood_ratio{$bigram}, " ", $bigram,
    "\t", $frequency_bigrams{$bigram}, "\t",
    $frequency{$bigram_array[0]}, "\t",
    $frequency{$bigram_array[1]}, "\n";
}
```

5.12 Application: Retrieval and Ranking of Documents on the Web

The advent of the Web in the mid-1990s made it possible to retrieve automatically billions of documents at a very modest cost. Companies providing such a service are among the most popular sites of the Internet. Google and Bing are among the most notable ones.

Table 5.16 An inverted index. Each word in the dictionary is linked to a posting list that gives all the documents in the collection where this word occurs and its positions in a document. Here, the position is the word index in the document. In the examples, a word occurs at most once in a document. This can be easily generalized to multiple occurrences

Words	Posting lists
America	(D1, 7)
Chrysler	(D1, 1) → (D2, 1)
in	(D1, 5) → (D2, 5)
investments	(D1, 4) → (D2, 4)
Latin	(D1, 6)
major	(D2, 3)
Mexico	(D2, 6)
new	(D1, 3)
plans	(D1, 2) → (D2, 2)

Web search systems or engines are based on "spiders" or "crawlers" that visit internet addresses, follow links they encounter, and collect all the pages they traverse. Crawlers can amass billions of pages every month.

5.12.1 Document Indexing

All the pages the crawlers download are tokenized and undergo a full text indexing. To carry out this first step, an indexer extracts all the words of the documents in the collection and builds a dictionary. It then links each word in the dictionary to the list of documents where this word occurs in. Such a list is called a *posting list*, where each posting in the list contains a document identifier and the word's positions in the corresponding document. The resulting data structure is called an *inverted index* and Table 5.16 shows an example of it with the two documents:

D1: Chrysler plans new investments in Latin America.
D2: Chrysler plans major investments in Mexico.

An inverted index is pretty much like a book index except that it considers all the words. When a user asks for a specific word, the search system answers with the pages that contain it. See Baeza-Yates and Ribeiro-Neto (2011) and Manning et al. (2008) for more complete descriptions.

5.12.2 Representing Documents as Vectors

Once indexed, search engines compare, categorize, and rank documents using statistical or popularity models. The vector space model (Salton 1988) is a widely used representation to carry this out. The idea is to represent the documents in a

Table 5.17 The vectors representing the two documents in Sect. 5.12.1. The words have been normalized in lowercase letters

D#\ Words	america	chrysler	in	investments	latin	major	mexico	new	plans
1	1	1	1	1	1	0	0	1	1
2	0	1	1	1	0	1	1	0	1

Table 5.18 The word by document matrix. Each cell (w_i, D_j) contains the frequency of w_i in document D_j

D#\Words	w_1	w_2	w_3	...	w_m
D_1	$C(w_1, D_1)$	$C(w_2, D_1)$	$C(w_3, D_1)$	...	$C(w_m, D_1)$
D_2	$C(w_1, D_2)$	$C(w_2, D_2)$	$C(w_3, D_2)$	...	$C(w_m, D_2)$
...					
D_n	$C(w_1, D_1 n)$	$C(w_2, D_n)$	$C(w_3, D_n)$	...	$C(w_m, D_n)$

vector space whose axes are the words. Documents are then vectors in a space of words. As the word order plays no role in the representation, it is often called a *bag-of-word model*.

Let us first suppose that the document coordinates are the occurrence counts of each word. A document would be represented as: $\mathbf{d} = (C(w_1), C(w_2), C(w_3), \ldots, C(w_n))$. Table 5.17 shows the document vectors representing the examples in Sect. 5.12.1, and Table 5.18 shows a general matrix representing a collection of documents, where each cell (w_i, D_j) contains the frequency of w_i in document D_j.

Using the vector space model, we can measure the similarity between two documents by the angle they form in the vector space. It is easier to compute the cosine of the angle, which is formulated as:

$$\cos(\mathbf{q}, \mathbf{d}) = \frac{\sum_{i=1}^{n} q_i d_i}{\sqrt{\sum_{i=1}^{n} q_i^2} \sqrt{\sum_{i=1}^{n} d_i^2}}.$$

5.12.3 Vector Coordinates

In fact, most of the time, the rough word counts that are used as coordinates in the vectors are replaced by a more elaborate term: the term frequency times the inverted document frequency, better known as $tf \times idf$ (Salton 1988). To examine how it works, let us take the phrase *internet in Somalia* as an example.

A document that contains many *internet* words is probably more relevant than a document that has only one. The frequency of a term i in a document j reflects this. It is a kind of a "mass" relevance. For each vector, the term frequencies $tf_{i,j}$ are often normalized by the sum of the frequencies of all the terms in the document and defined as:

$$tf_{i,j} = \frac{t_{i,j}}{\sum_i t_{i,j}},$$

or as the Euclidean norm:

$$tf_{i,j} = \frac{t_{i,j}}{\sqrt{\sum_i t_{i,j}^2}},$$

where $t_{i,j}$ is the frequency of term i in document j – the number of occurrences of term i in document j.

Instead of a sum, we can also use the maximum count over all the terms as normalization factor. The term frequency of the term i in document j is then defined as:

$$tf_{i,j} = \frac{t_{i,j}}{\max_i t_{i,j}}.$$

However, since *internet* is a very common word, it is not specific. The number of documents that contain it must downplay its importance. This is the role of

$$idf_i = \log(\frac{N}{n_i}),$$

where N is the total number of documents in the collection – the total number of pages the crawler has collected – divided by the number of pages n_i, where a term i occurs at least once. *Somalia* probably appears in fewer documents than *internet* and idf_i will give it a chance. The weight of a term i in document j is finally defined as

$$tf_{i,j} \times \log(\frac{N}{n_i}).$$

In this section, we gave one definition of $tf \times idf$. In fact, this formula can vary depending on the application. Salton and Buckley (1987) reported 287 variants of it and compared their respective merits. BM25 and BM25F (Zaragoza et al. 2004) are extensions of $tf \times idf$ that take into account the document length.

5.12.4 Ranking Documents

The user may query a search engine with a couple of words or a phrase. Most systems will then answer with the pages that contain all the words and any of the words of the question. Some questions return hundreds or even thousands of

valid documents. Ranking a document consists in projecting the space to that of the question words using the cosine. With this model, higher cosines will indicate better relevance. In addition to $tf \times idf$, search systems may employ heuristics such as giving more weight to the words in the title of a page (Mauldin and Leavitt 1994).

Google's PageRank algorithm (Brin and Page 1998) uses a different technique that takes into account the page popularity. PageRank considers the "backlinks", the links pointing to a page. The idea is that a page with many backlinks is likely to be a page of interest. Each backlink has a specific weight, which corresponds to the rank of the page it comes from. The page rank is simply defined as the sum of the ranks of all its backlinks. The importance of a page is spread through its forward links and contributes to the popularity of the pages it points to. The weight of each of these forward links is the page rank divided by the count of the outgoing links. The ranks are propagated in a document collection until they converge.

5.12.5 Categorizing Text

Text categorization (or classification) is a task related to ranking, but instead of associating documents to queries, we assign one or more classes to a text. The text size can range from a few words to entire books. In sentiment analysis (or opinion mining), the goal is to classify judgments or emotions expressed, for instance, in product reviews collected from consumer forums, into three base categories: positive, negative, or neutral; in spam detection, the categorizer classifies electronic messages into two classes: *spam* or *no spam*.

The Reuters corpus of newswire articles provides another example of a text collection that also serves as a standardized benchmark for categorization algorithms (Lewis et al. 2004). This corpus consists of 800,000 economic newswires in English and about 500,000 in 13 other languages, where each newswire is manually annotated with one or more topics selected from a set of 103 predefined categories, such as:

C11: STRATEGY/PLANS,
C12: LEGAL/JUDICIAL,
C13: REGULATION/POLICY,
C14: SHARE LISTINGS
etc.

Using manually-categorized corpora, like the Reuters corpus, and the vector space model, we can apply supervised machine-learning techniques to train classifiers (see Sect. 4.4). The training procedure uses a bag-of-word representation of the documents, either with Boolean features, term frequencies, or $tf \times idf$, and their classes as input. Support vector machines and logistic regression are two efficient techniques to carry out text classification. Joachims (2002) describes a state-of-the-art classifier based on support vector machines, while LibShortText

(Yu et al. 2013) is an open source library consisting of support vector machine and logistic regression algorithms, and different types of preprocessing and feature representations.

5.13 Further Reading

Language models and statistical techniques were applied first to speech recognition, lexicography, and later to other domains of linguistics. Their use had been a matter of debate because they opposed Chomsky's competence model. For a supporting review and a historical turning point, see the special issues of *Computational Linguistics* (1993, 1 and 2).

Interested readers will find additional details on language modeling techniques in Chen and Goodman (1998), and on χ^2 tests and likelihood ratios to improve collocation detection in Dunning (1993). Manning and Schütze (1999, Chapter 5) is a good reference on collocations, while Brown et al. (1992) describe other methods to create semantic clusters.

There are several toolkits available from the Internet to carry out tokenization, sentence detection, and language modeling:

1. Apache OpenNLP is a complete suite of logistic regression-based modules that includes, *inter alia*, a sentence detector, a tokenizer, and a document categorizer (http://opennlp.apache.org/).
2. The SRI Language Modeling collection (Stolcke 2002) is a C++ package to create and experiment with language models (http://www.speech.sri.com/).
3. The CMU-Cambridge Statistical Language Modeling Toolkit (Clarkson and Rosenfeld 1997) is another set of tools (http://svr-www.eng.cam.ac.uk/~prc14/toolkit.html).

Retrieval and ranking of documents have experienced a phenomenal growth since the beginning of the Web, making search sites the most popular services of the Internet. For complete reviews of techniques on information retrieval, see Manning et al. (2008) or Baeza-Yates and Ribeiro-Neto (2011).

Lucene is a popular open-source library for information retrieval. It is used in scores of web sites such as Twitter and Wikipedia to carry out document indexing and search (http://lucene.apache.org/).

Exercises

5.1. Write a sentence detector and a tokenizer using logistic regression.

5.2. Retrieve a text you like on the Internet. Give the five most frequent words.

5.3. Write a Prolog program that connects to a web site, and explore hypertext web links using a breadth-first strategy.

5.4. Implement a Prolog program to obtain bigrams and their statistics.

5.5. Implement a Prolog program to obtain trigrams and their statistics.

5.6. Retrieve a text you like on the Internet. Give the five most frequent bigrams and trigrams.

5.7. Retrieve a text you like on the Internet. Divide it into a training set and a test set. Implement the Laplace rule either in Perl or in Prolog. Learn the probabilities on the training set and compute the perplexity of the test set.

5.8. Retrieve a text you like on the Internet. Divide it into a training set and a test set. Implement the Good–Turing estimation either in Perl or in Prolog. Learn the probabilities on the training set and compute the perplexity of the test set.

5.9. Implement the mutual information test in Prolog.

5.10. Implement the t-score in Prolog.

5.11. Implement the likelihood ratio in Prolog.

5.12. Implement the mutual information test with a window of five words to the left and to the right of the word.

Chapter 6
Words, Parts of Speech, and Morphology

> Partes orationis quot sunt? Octo. Quae? Nomen, pronomen, verbum, adverbium, participium, coniunctio, praepositio, interiectio.
>
> Aelius Donatus, *Ars grammatica. Ars minor*, Fourth century.

6.1 Words

6.1.1 Parts of Speech

We can divide the lexicon into **parts of speech** (POS), that is, classes whose words share common grammatical properties. The concept of part of speech dates back to the classical antiquity philosophy and teaching. Plato made a distinction between the verb and the noun. After him, the word classification further evolved, and parts of speech grew in number until Dionysius Thrax fixed and formulated them in a form that we still use today. Aelius Donatus popularized the list of the eight parts of speech: noun, pronoun, verb, adverb, participle, conjunction, preposition, and interjection, in his work *Ars grammatica*, a reference reading in the Middle Ages.

The word parsing comes from the Latin phrase *partes orationis* 'parts of speech'. It corresponds to the identification of the words' parts of speech in a sentence. In natural language processing, POS tagging is the automatic annotation of words with grammatical categories, also called POS tags. Parts of speech are also sometimes called lexical categories.

Most European languages have inherited the Greek and Latin part-of-speech classification with a few adaptations. The word categories as they are taught today roughly coincide in English, French, and German in spite of some inconsistencies. This is not new. To manage the nonexistence of articles in Latin, Latin grammarians tried to get the Greek article into the Latin pronoun category.

P.M. Nugues, *Language Processing with Perl and Prolog*, Cognitive Technologies,
DOI 10.1007/978-3-642-41464-0_6, © Springer-Verlag Berlin Heidelberg 2014

Table 6.1 Closed class categories

Part of speech	English	French	German
Determiners	*the, several, my*	*le, plusieurs, mon*	*der, mehrere, mein*
Pronouns	*he, she, it*	*il, elle, lui*	*er, sie, ihm*
Prepositions	*to, of*	*vers, de*	*nach, von*
Conjunctions	*and, or*	*et, ou*	*und, oder*
Auxiliaries and modals	*be, have, will, would*	*être, avoir, pouvoir*	*sein, haben, können*

Table 6.2 Open class categories

Part of speech	English	French	German
Nouns	*name, Frank*	*nom, François*	*Name, Franz*
Adjectives	*big, good*	*grand, bon*	*groß, gut*
Verbs	*to swim*	*nager*	*schwimmen*
Adverbs	*rather, very, only*	*plutôt, très, uniquement*	*fast, nur, sehr, endlich*

The definition of the parts of speech is sometimes arbitrary and has been a matter of debate. From Dionysius Thrax, tradition has defined the parts of speech using morphological and grammatical properties. We shall adopt essentially this viewpoint here. However, words of a certain part of speech share semantic properties, and some grammars contain statements like a noun denotes a thing and a verb an action.

Parts of speech can be clustered into two main classes: the **closed class** and the **open class**. Closed class words are relatively stable over time and have a functional role. They include words such as articles, like English *the*, French *le*, or German *der*, which change very slowly. Among the closed class, there are the determiners, the pronouns, the prepositions, the conjunctions, and the auxiliary and modal verbs (Table 6.1).

Open class words form the bulk of a vocabulary. They appear or disappear with the evolution of the language. If a new word is created, say a *hedgedog*, a cross between a hedgehog and a Yorkshire terrier, it will belong to an open class category: here a noun. The main categories of the open class are the nouns, the adjectives, the verbs, and the adverbs (Table 6.2). We can add interjection to this list. Interjections are words such as *ouch*, *ha*, *oh*, and so on, that express sudden surprise, pain, or pleasure.

6.1.2 Grammatical Features

Basic categories can be further refined, that is **subcategorized**. Nouns, for instance, can be split into singular nouns and plural nouns. In French and German, nouns can also be split according to their gender: masculine and feminine for French, and masculine, feminine, and neuter for German.

Table 6.3 Features of common nouns

Features\Values	English	French	German
Number	singular, plural *waiter/waiters, book/books*	singular, plural *serveur/serveurs, livre/livres*	singular, plural *Buch/Bücher*
Gender		masculine, feminine *serveur/table*	masculine, feminine, neuter *Ober/Gabel/Tuch*
Case			nominative, accusative, genitive, dative *Junge/Jungen/Jungen/Jungen*

Genders do not correspond in these languages and can shape different visions of the world. Sun is a masculine entity in French – *le soleil* – and a feminine one in German – *die Sonne*. In contrast, moon is a feminine entity in French – *la lune* – and a masculine one in German – *der Mond*.

Additional properties that can further specify main categories are often called the **grammatical features**. Grammatical features vary among European languages and include notably the number, gender, person, case, and tense. Each feature has a set of possible values; for instance, the number can be singular or plural.

Grammatical features are different according to their parts of speech. In English, a verb has a tense, a noun has a number, and an adjective has neither tense nor number. In French and German, adjectives have a number but no tense. The feature list of a word defines its part of speech together with its role in the sentence.

6.1.3 Two Significant Parts of Speech: The Noun and the Verb

The Noun

Nouns are divided into proper and common nouns. Proper nouns are names of persons, people, countries, companies, and trademarks, such as *England, Robert, Citroën*. Common nouns are the rest of the nouns. Common nouns are often used to qualify persons, things, and ideas.

A noun definition referring to semantics is a disputable approximation, however. More surely, nouns have certain syntactic features, namely the number, gender, and case (Table 6.3). A noun group is marked with these features, and other words of the group, that is, determiners, adjectives, must agree with the features they share.

While number and gender are probably obvious, case might be a bit obscure for non-German speakers. Case is a function marker that inflects words such as nouns or adjectives. In German, there are four cases: nominative, accusative, genitive, and dative. The nominative case corresponds to the subject function, the accusative case to the direct object function, and the dative case to the indirect object function.

Table 6.4 Auxiliary verbs

English	French	German
***to be**: am, are, is, was, were* ***to have**: has, have, had* ***to do**: does, did, done*	***être**: suis, es, est, sommes, sont, étais, était* ***avoir**: ai, as, a, avons, ont, avais, avait, avions*	***sein**: bin, bist, ist, war, waren* ***haben**: habe, hast, hat, haben, habt* ***werden**: werde, wirst, wird, wurde*

Table 6.5 Modal verbs

English	French (semiauxiliaries)	German
can, could, must, may, might, shall, should	***pouvoir**: peux, peut, pouvons, pourrai, pourrais* ***devoir**: dois, doit, devons, devrai, devrais* ***vouloir**: veux, veut, voulons, voudrai, voudrais*	***können**: kann, können, konnte* ***dürfen**: darf, dürfen, dürfte* ***mögen**: mag, mögen, möchte* ***müssen**: muß, müssen, mußte* ***sollen**: soll, sollen, sollte*

Genitive denotes a possession relation. These cases are still marked in English and French for pronouns.

In addition to these features, the English language makes a distinction between nouns that can have a plural: count nouns, and nouns that cannot: mass nouns. *Milk*, *water*, *air* are examples of mass nouns.

Verbs

Semantically, verbs often describe an action, an event, a state, etc. More positively, and as for the nouns, verbs in European languages are marked by their morphology. This morphology is quite elaborate in a language like French, notably due to the tense system. Verbs can be basically classified into three main types: auxiliaries, modals, and main verbs.

Auxiliaries are helper verbs such as *be* and *have* that enable us to build some of the main verb tenses (Table 6.4). Modal verbs are verbs immediately followed by another verb in the infinitive. They usually indicate a modality, a possibility (Table 6.5). Modal verbs are more specific to English and German. In French, semiauxiliaries correspond to a similar category.

Main verbs are all the other verbs. Traditionally, main verbs are categorized according to their complement's function (Table 6.6):

- Copula or link verb – verbs linking a subject to an (adjective) complement. Copulas include **verbs of being** such as *be, être, sein* when not used as auxiliaries, and other verbs such as *seem, sembler, scheinen*.
- Intransitive – verbs taking no object.

Table 6.6 Verb types

	English	French	German
Copulas	*Man **is** mortal* *She **seems** intelligent*	*L'homme **est** mortel* *Elle **paraît** intelligente*	*Der Mensch **ist** sterblich* *Sie **scheint** intelligent*
Intransitive verbs	*Frank **sleeps*** *Charlotte **runs***	*François **dort*** *Charlotte **court***	*Franz **schläft*** *Charlotte **rennt***
Transitive verbs	*You **take** the book* *Susan **reads** the paper*	*Tu **prends** le livre* *Suzanne **lit** l'article*	*Du **nimmst** das Buch* *Susan **liest** den Artikel*
Ditransitive verbs	*I **give** my neighbors the notes*	*Je **donne** les notes à mon voisin*	*Ich **gebe** die Notizen meinem Nachbarn*

Table 6.7 Features common to verbs and nouns

Features\Values	English	French	German
Person	1, 2, and 3	1, 2, and 3	1, 2, and 3
	I am	*je suis*	*ich bin*
	you are	*tu es*	*du bist*
	she is	*elle est*	*sie ist*
Number	singular, plural	singular, plural	singular, plural
	I am/we are	*je suis/nous sommes*	*ich bin/wir sind*
	She eats/they eat	*elle mange/elles mangent*	*sie ißt/sie essen*
Gender		masculine, feminine	
	–	*il est mangé/elle est mangée*	–

- Transitive – verbs taking an object.
- Ditransitive – verbs taking two objects.

Verbs have more features than other parts of speech. First, the verb group shares certain features of the noun (Table 6.7). These features must agree with corresponding ones of the verb's subject.

Verbs have also specific features, namely the tense, the mode, and the voice:

- **Tense** locates the verb, and the sentence, in time. Tense systems are elaborate in English, French, and German, and do not correspond. Tenses are constructed using form variations (Table 6.8) or auxiliaries (Table 6.9). Tenses are a source of significant form variation in French.
- **Mood** enables the speaker to present or to conceive of the action in various ways (Table 6.10).
- **Voice** characterizes the sequence of syntactic groups. Active voice corresponds to the "subject, verb, object" sequence. The reverse sequence corresponds to the passive voice. This voice is possible only for transitive verbs. Some constructions in French and German use a reflexive pronoun. They correspond to the pronominal voice.

Table 6.8 Tenses constructed using inflection

	English	French	German
Base	*I like to **sing***	*j'aime **chanter***	*Ich **singe** gern*
Present	***I sing** every day*	*Je **chante** tous les jours*	*Ich **singe** alltags*
Preterit (Simple past)	***I sang** in my youth*	*Je **chantai** dans ma jeunesse*	*Ich **sang** in meiner Jugend*
Imperfect	–	*Je **chantais** dans ma jeunesse*	–
Future	–	*Je **chanterai** plus tard*	–
Present participle	*I am **singing***	*En **chantant** tous les jours*	***Singend***
Past participle	*I have **sung** before*	*J'ai **chanté***	*Ich habe **gesungen***

Table 6.9 Some tenses constructed using auxiliaries. Values do not correspond across languages

	English	French	German
Present progressive	*I am singing*	–	–
Future	*I shall (will) sing*	–	*Ich werde singen*
Present perfect	*I have sung*	*J'ai chanté*	*Ich habe gesungen*
Pluperfect	*I had sung*	*J'avais chanté*	*Ich hatte gesungen*
Passé antérieur	–	*J'eus chanté*	–
Future perfect	*I will have sung*	*J'aurai chanté*	*Ich werde gesungen haben*
Futur antérieur	*I would have sung*	*J'aurais chanté*	*Ich würde gesungen haben*
Past progressive	*I was singing*	–	–
Future progressive	*I will be singing*	–	–
Present perfect progressive	*I have been singing*	–	–
Future perfect progressive	*I will have been singing*	–	–
Past perfect progressive	*I had been singing*	–	–

Table 6.10 Moods (Present only)

	English	French	German
Indicative	*I am singing*	*Je chante*	*Ich singe*
Imperative	*sing*	*chante*	*singe*
Conditional	*I should (would) sing*	*Je chanterais*	*Ich würde singen*
Subjunctive	Rare, it appears in expressions such as: *God save the Queen*	*Il faut que je chante*	*Ich singe*

6.2 Lexicons

A lexicon is a list of words, and in this context, lexical entries are also called the **lexemes**. Lexicons often cover a particular domain. Some focus on a whole language, like English, French, or German, while some specialize in specific areas such as proper names, technology, science, and finance. In some applications,

Table 6.11 Word ambiguity

	English	French	German
Part of speech	*can* modal	*le* article	*der* article
	can noun	*le* pronoun	*der* pronoun
Semantic	*great* big	*grand* big	*groß* big
	great notable	*grand* notable	*groß* notable

lexicons try to be as exhaustive as is humanly possible. This is the case of internet crawlers, which index all the words of all the web pages they can find. Computerized lexicons are now embedded in many popular applications such as in spelling checkers, thesauruses, or definition dictionaries of word processors. They are also the first building block of most language processing programs.

Several options can be taken when building a computerized lexicon. They range from a collection of words – a word list – to words carefully annotated with their pronunciation, morphology, and syntactic and semantic labels. Words can also be related together using semantic relationships and definitions.

A key point in lexicon building is that many words are ambiguous both syntactically and semantically. Therefore, each word may have as many entries as it has syntactic or semantic readings. Table 6.11 shows words that have two or more parts of speech and senses. In this chapter, we only examine the syntactic part. Chapter 15 will cover semantic issues.

Many computerized lexicons are now available from industry and from sources on the Internet. English sources are the most numerous at present, but the situation is rapidly changing for other languages. Most notable ones in English include word lists derived from the *Longman Dictionary of Contemporary English* (Procter 1978) and the *Oxford Advanced Learner's Dictionary* (Hornby 1974). Table 6.12 shows the first lines of letter *A* of an electronic version of the OALD.

BDLex – standing for *Base de Données Lexicale* – is an example of a simple French lexicon (Pérennou and de Calmès 1987). BDLex features a list of words in a lemmatized form together with their part of speech and a syntactic type (Table 6.13).

6.2.1 Encoding a Dictionary

Letter trees (de la Briandais 1959) or tries (pronounce try ees) are a useful data structures to store large lexicons and to search words quickly. The idea behind a trie is to store the words as trees of characters and to share branches as far as the letters of two words are identical. Tries can be seen as finite-state automata, and Fig. 6.1 shows a graphical representation of a trie encoding the words *bin*, *dark*, *dawn*, *tab*, *table*, *tables*, and *tablet*.

In Prolog, we can represent this trie as embedded lists, where each branch is a list. The first element of a branch is the root letter: the first letter of all the subwords that correspond to the branch. The leaves of the trie are the lexical entries, here the

Table 6.12 The first lines of the *Oxford Advanced Learner's Dictionary*

Word	Pronunciation	Syntactic tag	Syllable count or verb pattern (for verbs)
a	@	S-*	1
a	EI	Ki$	1
a fortiori	eI ,fOtI'OraI	Pu$	5
a posteriori	eI ,p0sterI'OraI	OA$,Pu$	6
a priori	eI ,praI'OraI	OA$, Pu$	4
a's	Eiz	Kj$	1
ab initio	&b I'nISI@U	Pu$	5
abaci	'&b@saI	Kj$	3
aback	@'b&k	Pu%	2
abacus	'&b@k@s	K7%	3
abacuses	'&b@k@sIz	Kj%	4
abaft	@'bAft	Pu$,T-$	2
abandon	@'b&nd@n	H0%,L@%	36A,14
abandoned	@'b&nd@nd	Hc%,Hd%,OA%	36A,14
abandoning	@'b&nd@nIN	Hb%	46A,14
abandonment	@'b&nd@nm@nt	L@%	4
abandons	@'b&nd@nz	Ha%	36A,14
abase	@'beIs	H2%	26B
abased	@'beIst	Hc%,Hd%	26B
abasement	@'beIsm@nt	L@%	3

words themselves that we represent as atoms. Of course, these entries could contain more information, such as the part of speech, the pronunciation, etc.

```
[
 [b, [i, [n, bin]]]
 [d, [a, [r, [k, dark]],
         [w, [n, dawn]]]]
 [t, [a, [b, tab,
             [l, [e, table,
                     [s, tables],
                     [t, tablet]]]]]]]
]
```

6.2.2 Building a Trie in Prolog

The `make_trie/2` predicate builds a trie from a lexicon represented as an ordered list of atoms.

```
% make_trie(+WordList, -Trie)
make_trie([Word | WordList], Trie) :-
```

Table 6.13 An excerpt from BDLex. Digits encode accents on letters. The syntactical tags of the verbs correspond to their conjugation type taken from the *Bescherelle* reference

Entry	Part of speech	Lemma	Syntactic tag
a2	Prep	a2	Prep_00_00;
abaisser	Verbe	abaisser	Verbe_01_060_**;
abandon	Nom	abandon	Nom_Mn_01;
abandonner	Verbe	abandonner	Verbe_01_060_**;
abattre	Verbe	abattre	Verbe_01_550_**;
abbe1	Nom	abbe1	Nom_gn_90;
abdiquer	Verbe	abdiquer	Verbe_01_060_**;
abeille	Nom	abeille	Nom_Fn_81;
abi3mer	Verbe	abi3mer	Verbe_01_060_**;
abolition	Nom	abolition	Nom_Fn_81;
abondance	Nom	abondance	Nom_Fn_81;
abondant	Adj	abondant	Adj_gn_01;
abonnement	Nom	abonnement	Nom_Mn_01;
abord	Nom	abord	Nom_Mn_01;
aborder	Verbe	aborder	Verbe_01_060_**;
aboutir	Verbe	aboutir	Verbe_00_190_**;
aboyer	Verbe	aboyer	Verbe_01_170_**;
abre1ger	Verbe	abre1ger	Verbe_01_140_**;
abre1viation	Nom	abre1viation	Nom_Fn_81;
abri	Nom	abri	Nom_Mn_01;
abriter	Verbe	abriter	Verbe_01_060_**;

```
    make_trielist(Word, Word, WordTrie),
    make_trie(WordList, [WordTrie], Trie).

% make_trie(+WordList, -Trie, -FinalTrie)
make_trie([], T, T) :- !.
make_trie([Word | WordList], Trie, FinalTrie) :-
    insert_word_in_trie(Word, Word, Trie, NewTrie),
    make_trie(WordList, NewTrie, FinalTrie).
```

The `make_trie/2` predicate uses `make_trielist/3` to transform an atom into a trie representing a single word. The `make_trielist/3` predicate takes the word and the lexical entry as an input:

```
?- make_trielist(tab, noun, TL).
TL = [t, [a, [b, noun]]]

%make_trielist(+Word, +Leave, -WordTtrie)
% Creates the trie for a single word.
% Leaf contains the type of the word.
make_trielist(Word, Leaf, WordTrie) :-
    atom_chars(Word, CharList),
```

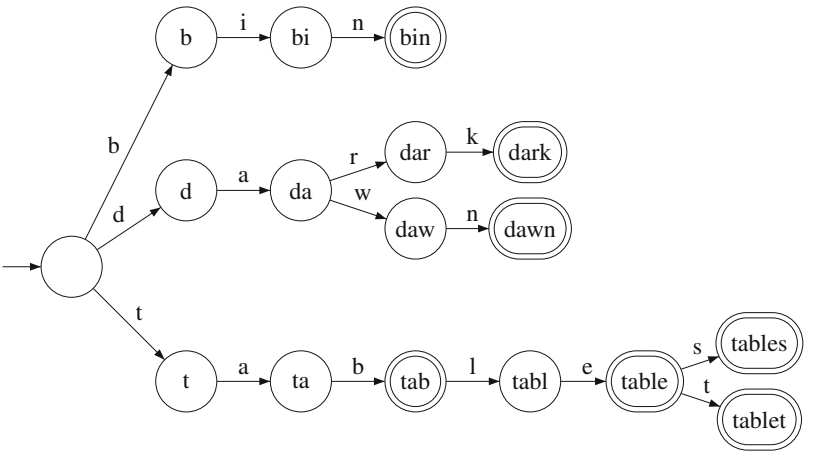


Fig. 6.1 A letter tree encoding the words *tab*, *table*, *tablet*, and *tables*

```
    make_trielist_aux(CharList, Leaf, WordTrie).

  make_trielist_aux([X], Leaf, [X, Leaf]) :- !.
  make_trielist_aux([X | L], Leaf, [X | [LS]]) :-
    make_trielist_aux(L, Leaf, LS).
```

Finally, `make_trie/2` inserts a word trie into the lexicon trie using `insert_word_in_trie/4`:

```
  %Inserts a word in a trie.
  %The Leaf argument contains the type of the word
  %insert_word_in_trie(+Word, +Leaf, +Trie, -NewTrie)
  insert_word_in_trie(Word, Leaf, Trie, NewTrie) :-
    make_trielist(Word, Leaf, WordTrie),
    insert_wordtrie_in_trie(WordTrie, Trie, NewTrie).

  %Inserts a word trie in a trie
  %insert_wordtrie_in_trie(+WordTrie, +Trie, -NewTrie)
  insert_wordtrie_in_trie([H | [T]],
      [[H, Leaf | BT] | LT], [[H, Leaf | NB] | LT]) :-
    atom(Leaf),
    !,
    insert_wordtrie_in_trie(T, BT, NB).
  % Traverses a segment shared between the trie and
  % the word and encounters a leaf.
  % It assumes that the leaf is an atom.

  insert_wordtrie_in_trie([H | [T]], [[H | BT] | LT],
```

```
      [[H | NB] | LT]) :-
    !,
    insert_wordtrie_in_trie(T, BT, NB).
% Traverses a segment shared between the trie and
% the word.

insert_wordtrie_in_trie([H | T], [[HT | BT] | LT],
      [[HT | BT] | NB]) :-
    !,
    insert_wordtrie_in_trie([H | T], LT, NB).
% Traverses a nonshared segment

insert_wordtrie_in_trie(RW, RT, NB) :-
    append(RT, [RW], NB),
    !.
% Appends the remaining part of the word to the trie.
```

6.2.3 Finding a Word in a Trie

The rules to find a word in a trie are easier to write. A first rule compares the first letter of the word to the trie and unifies with the branch starting with this letter. It continues recursively with the remaining characters of the word. A second rule extracts the lexical entries that we assume to be atoms.

```
% Checks if a word is in a trie
% is_word_in_trie(+WordChars, +Trie, -Lex)
is_word_in_trie([H | T], Trie, Lex) :-
    member([H | Branches], Trie),
    is_word_in_trie(T, Branches, Lex).
is_word_in_trie([], Trie, LexList) :-
    findall(Lex, (member(Lex, Trie), atom(Lex)), LexList),
    LexList \= [].
% We assume that the word lexical entry is an atom
```

6.3 Morphology

6.3.1 Morphemes

From a morphological viewpoint, a language is a set of morphemes divided into **lexical** and **grammatical** morphemes. Lexical morphemes correspond to the word stems and form the bulk of the vocabulary. Grammatical morphemes include

Table 6.14 Morpheme decomposition. We replaced the stems with the corresponding lemmas

	Word	Morpheme decomposition
English	*disentangling*	*dis*+*en*+***tangle***+*ing*
	rewritten	*re*+***write***+*en*
French	*désembrouillé*	*dé*+*em*+***brouiller***+*é*
	récrite	*re*+***écrire***+*te*
German	*entwirrend*	*ent*+***wirren***+*end*
	wiedergeschrieben	*wieder*+*ge*+***schreiben***+*en*

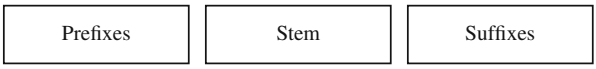


Fig. 6.2 Concatenative morphology where prefixes and suffixes are concatenated to the stem

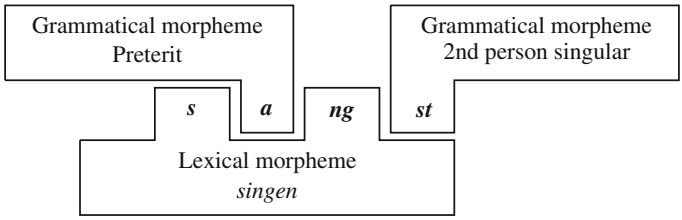


Fig. 6.3 Embedding of the stem into the grammatical morphemes in the German verb *sangst* (second-person preterit of *singen*) (After Simone (2007, p. 144))

grammatical words and the affixes. In European languages, words are made of one or more morphemes (Table 6.14). The affixes are concatenated to the stem (bold): before it – the prefixes (underlined) – and after it – the suffixes (double underlined). When a prefix and a suffix surrounding the stem are bound together, it is called a circumfix, as in the German part participle (wavy underlines).

Affixing grammatical morphemes to the stem is general property of most European languages, which is **concatenative morphology** (Fig. 6.2). Although there are numerous exceptions, it enables us to analyze the structure of most words.

Concatenative morphology is not universal, however. The Semitic languages, like Arabic or Hebrew, for instance, have a **templatic morphology** that interweaves the grammatical morphemes to the stem. There are also examples of nonconcatenative patterns in European languages like in irregular verbs of German. The verb *singen* 'sing' has the forms *sangst* 'you sang' and *gesungen* 'sung' where the stem [s–ng] is embedded into the grammatical morphemes [–a–st] for the second-person preterit (Fig. 6.3) and [ge–u–en] for the past participle (Fig. 6.4).

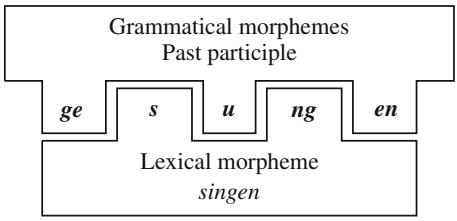


Fig. 6.4 Embedding of the stem into the grammatical morphemes in the German verb *gesungen* (past participle of *singen*) (After Simone (2007, p. 144))

Table 6.15 Plural morphs

	Plural of nouns	**Morpheme decomposition**
English	*hedgehogs*	*hedgehog+s*
	churches	*church+es*
	sheep	*sheep+Ø*
French	*hérissons*	*hérisson+s*
	chevaux	*cheval+ux*
German	*Gründe*	*Grund+(¨)e*
	Hände	*Hand+(¨)e*
	Igel	*Igel+Ø*

6.3.2 Morphs

Grammatical morphemes represent syntactic or semantic functions whose realizations in words are called **morphs**. Using an object-oriented terminology, morphemes would be the classes, while morphs would be the objects. The **allomorphs** correspond to the set of all the morphs in a morpheme class.

The plural morpheme of English and French nouns is generally realized with an *s* suffix – an *s* added at the end of the noun. It can also be *es* or nothing (Ø) in English and *ux* in French. In German, the plural morpheme can take several shapes, such as suffixes *e*, *en* , *er*, *s*, or an umlaut on the first vowel of the word (Table 6.15):

- In English, suffixes *-s*, *-es*, etc.
- In French, *-s*, *-ux*, etc.
- In German, an umlaut on the first vowel and the *-e* suffix, or simply the *-e* suffix.

Plurals also offer exceptions. Many of the exceptions, such as *mouse* and *mice*, are not predictable and have to be listed in the lexicon.

6.3.3 Inflection and Derivation

Some Definitions

We saw in Chap. 1 that morphology can be classified into **inflection**, the form variation of a word according to syntactic features such as gender, number, person, tense, etc., and **derivation**, the creation of a new word – a new meaning – by

Table 6.16 Verb inflection with past participle

	English	**French**	**German**
Base form	*work*	*travailler, chanter*	*arbeiten*
	sing	*paraître*	*singen*
Past participle (regular)	*worked*	*travaillé, chanté*	*gearbeitet*
Past participle (exception)	*sung*	*paru*	*gesungen*

concatenating a word with a specific affix. A last form of construction is the **composition (compounding)** of two words to give a new one, for instance, *part of speech*, *can opener*, *pomme de terre.* Composition is more obvious in German, where such new words are not separated with a space, for example, *Führerschein*. In English and French, some words are formed in this way, such as *bedroom*, or are separated with a hyphen, *centre-ville*. However, the exact determination of other compounded words – separated with a space – can be quite tricky.

Inflection

Inflection corresponds to the application of a grammatical feature to a word, such as putting a noun into the plural or a verb into the past participle (Table 6.16). It is also governed by its context in the sentence; for instance, the word is bound to agree in number with some of its neighbors.

Inflection is relatively predictable – regular – depending on the language. Given a lemma, its part of speech, and a set of grammatical features, it is possible to construct a word form using rules, for instance, gender, plural, or conjugation rules. The past participle of regular English, French, and German verbs can be respectively formed with an *ed* suffix, an *é* suffix, and the *ge* prefix and the *t* suffix. Morphology also includes frequent exceptions that can sometimes also be described by rules.

Inflectional systems are similar in European languages but show differences according to the syntactic features. In English, French, and German, nouns are inflected with plurals and are consequently decorated with a specific suffix. However, in French and other Romance languages, verbs are inflected with future. Verb *chanterons* is made of two morphs: *chant* 'sing' and *-erons*. The first one is the stem (root) of *chanter*, and the second one is a suffix indicating the future tense, the first person, and the plural number. In English and German, this tense is rendered with an auxiliary: *we shall sing* or *wir werden singen*.

Derivation

Derivation is linked to lexical semantics and involves another set of affixes (Table 6.17). Most affixes can only be attached to a specific lexical category (part of speech) of words: some to nouns, others to verbs, etc. Some affixes leave the derived word in the same category, while some others entail a change of category. For

Table 6.17 Derivational affixes

	English	French	German
Prefixes	***fore**see*, ***un**pleasant*	***pré**voir*, ***dé**plaisant*	***vorher**sehen*, ***un**angenehm*
Suffixes	*manage**able**, rigor**ous***	*gér**able**, rigour**eux***	*vorsicht**ich**, streit**bar***

Table 6.18 Derivation related to part of speech

	Adjectives	Adverbs	Nouns	Adjectives	Verbs	Nouns
English	*recent*	*recently*	*air*	*aerial*	*compute*	*computation*
	frank	*frankly*	*base*	*basic*		
French	*récent*	*récemment*	*lune*	*lunaire*	*calculer*	*calcul*
	franc	*franchement*	*air*	*aérien*		
German	*glücklich*	*glücklicherweise*	*Luft*	*luftig*	*rechnen*	*Rechnung*
	möglich	*möglicherweise*	*Grund*	*gründlich*		

Table 6.19 Word derivation

	Word	Contrary	Possibility
English	*pleasant do*	*unpleasant undo*	**pleasable doable*
French	*plaisant faire*	*déplaisant défaire*	**plaisable faisable*
German	*angenehm tun*	*unangenehm *untun*	**angenehmbar tunlichst*

instance, some affixes transform adjectives into adverbs, nouns into adjectives, and verbs into nouns (Table 6.18). Derivation rules can be combined and are sometimes complex. For instance, the word *disentangling* features two prefixes: *dis-* and *en-*, and a suffix *-ing*.

Some semantic features of words, such as the contrary or the possibility, can be roughly associated to affixes, and so word meaning can be altered using them (Table 6.19). However, derivation is very irregular. Many words cannot be generated as simply, because the word does not exist or sounds weird. In addition, some affixes cannot be mapped to clear semantic features.

Compounding is a feature of German, Dutch, and the Scandinavian languages. It resembles the English noun sequences with the difference that nouns are not separated with a white space. English open compounds (e.g., *word processor*) are.

Morphological Processing

Morphological processing includes parsing and generation (Table 6.20). Parsing consists in splitting an inflected, derived, or compounded word into morphemes; this process is also called a **lemmatization**. Lemmatization refers to transforming a word into its canonical dictionary form, for example, *retrieving* into *retrieve*, *recherchant* into *rechercher*, or *suchend* into *suchen*. Stemming consists of removing the suffix from the rest of the word. Taking the previous examples, this yields *retriev*,

Table 6.20 Morphological generation and parsing

Generation →

English		**French**		**German**	
dog+s	*dogs*	*chien+s*	*chiens*	*Hund+e*	*Hunde*
work+ing	*working*	*travailler+ant*	*travaillant*	*arbeiten+end*	*arbeitend*
un+do	*undo*	*dé+faire*	*défaire*		

← Parsing

Table 6.21 Open class word morphology, where `*` denotes zero or more elements and `?` denotes an optional element

English and French	`prefix* stem suffix* inflection?`
German	`inflection? prefix* stem* suffix* inflection?`

recherch, and *such*. Lemmatization and stemming are often mistaken. Conversely, generation consists of producing a word – a lexical form – from a set of morphemes.

In French, English, and German, derivation operates on open class words. In English and French, a word of this class consists of a stem preceded by zero or more derivational prefixes and followed by zero or more derivational suffixes. An inflectional suffix can be appended to the word. In German, a word consists of one or more stems preceded by zero or more derivational prefixes and followed zero or more derivational suffixes. An inflectional prefix and an inflectional suffix can be appended to the word (Table 6.21). As we saw earlier, these rules are general principles of concatenative morphology that have exceptions.

Ambiguity

Word lemmatization is often ambiguous. An isolated word can lead to several readings: several bases and morphemes, and in consequence several categories and features as exemplified in Table 6.22.

Lemmatization ambiguities are generally resolved using the word context in the sentence. Usually only one reading is syntactically or semantically possible, and others are not. The correct reading of a word's part of speech is determined considering the word's relations with the surrounding words and with the rest of the sentence. From a human perspective, this corresponds to determining the word's function in the sentence. As we saw in the introduction, this process has been done by generations of pupils dating as far back as the schools of ancient Greece and the Roman Empire.

Table 6.22 Lemmatization ambiguities

	Words	Words in context	Lemmatization
English	*Run*		
		1. *A **run** in the forest*	1. **run**: noun singular
		2. *Sportsmen **run** every day*	2. **run**: verb present third person plural
French	*Marche*		
		1. *Une **marche** dans la forêt*	1. **marche**: noun singular feminine
		2. *Il **marche** dans la cour*	2. **marcher**: verb present third person singular
German	*Lauf*		
		1. *Der **Lauf** der Zeit*	1. **Der Lauf**: noun, sing, masc
		2. ***Lauf** schnell!*	2. **laufen**: verb, imperative, singular

6.3.4 Language Differences

Paper lexicons do not include all the words of a language but only lemmas. Each lemma is fitted with a morphological class to relate it to a model of inflection or possible exceptions. A French verb will be given a class of conjugation or its exception pattern – one among a hundred. English or German verbs will be marked as regular or strong and in this latter case will be given their irregular forms. Then, a reader can apply morphological rules to produce all the lexical forms of the language.

Automatic morphological processing tries to mimic this human behavior. Nevertheless, it has not been so widely implemented in English as in other languages. Programmers have often preferred to pack all the English words into a single dictionary instead of implementing a parser to do the job. This strategy is possible for European languages because morphology is finite: there is a finite number of noun forms, adjective forms, or verb forms. It is clumsy, however, to extend it to languages other than English because it considerably inflates the size of dictionaries.

Statistics from Xerox (Table 6.23) show that techniques available for storing English words are very costly for many other languages. It is not a surprise that the most widespread morphological parser – KIMMO – was originally built for Finnish, one of the most inflection-rich languages. In addition, while English inflection is tractable by means of storing all the forms in a lexicon, it is often necessary to resort to a morphological parser to deal with forms such as: *computer*, *computerize*, *computerization*, *recomputerize* (Antworth 1994), which cannot all be foreseen by lexicographers.

Table 6.23 Some language statistics from a Xerox promotional flyer

Language	Number of stems	Number of inflected forms	Lexicon size (kb)
English	55,000	240,000	200–300
French	50,000	5,700,000	200–300
German	50,000	350,000 or Infinite (compounding)	450
Japanese	130,000	200 suffixes 20,000,000 word forms	500 500
Spanish	40,000	3,000,000	200–300

Table 6.24 Surface and lexical forms

	Generation: Lexical to surface form →	
English	*dis+en+tangle+ed*	*disentangled*
	happy+er	*happier*
	move+ed	*moved*
French	*dés+em+brouiller+é*	*désembrouillé*
	dé+chanter+erons	*déchanterons*
German	*ent+wirren+end*	*entwirrend*
	wieder+ge+schreiben+en	*wiedergeschrieben*
		Parsing: ← Surface to lexical form

6.4 Morphological Parsing

6.4.1 Two-Level Model of Morphology

Using a memory expensive method, lemmatization can be accomplished with a lexicon containing all the words with all their possible inflections. A dictionary lookup then yields the lemma of each word in a text. Although it has often been used for English, this method is not very efficient for many other languages. We now introduce the two-level model of Kimmo Koskenniemi (1983), which is universal and has been adopted by many morphological parsers.

The two-level morphology model enables us to link the **surface form** of a word – the word as it is actually in a text – to its **lexical** or **underlying form** – its sequence of morphemes. Karttunen (1983) did the first implementation of this model, which he named KIMMO. A later implementation – PC-KIMMO 2 – was carried out by Antworth (1995) in C. PC-KIMMO 2 is available from the Summer Institute of Linguistics through the Internet.

Table 6.24 shows examples of correspondence between surface forms and lexical forms. Morpheme boundaries in lexical forms are denoted by +.

In the two-level model, the mapping between the surface and lexical forms is synchronous. Both strings need to be aligned with a letter-for-letter correspondence. That is, the first letter of the first form is mapped to the first letter of the second form, and so on. To maintain the alignment, possible null symbols are inserted in either

Table 6.25 Correspondence between lexical and surface forms

English	dis+en+tangle+ed	happy+er	move+ed
	↕↕ …	↕↕ …	↕↕ …
	dis0en0tangl00ed	happi0er	mov00ed
French	dé+chanter+erons	cheval+ux	cheviller+é
	↕↕ …	↕↕ …	↕↕ …
	dé0chant000erons	cheva00ux	chevill000é
German	singen+st	Grund+¨e	Igel+Ø
	↕↕ …	↕↕ …	↕↕ …
	singe00st	Gründ00e	Igel00

form and are denoted ε or 0, if the Greek letters are not available. They reflect a letter deletion or insertion. Table 6.25 shows aligned surface and lexical forms.

6.4.2 *Interpreting the Morphs*

Considering inflection only, it is easier to interpret the morphological information using grammatical features rather than morphs. Most morphological parsers represent the lexical form as a concatenation of the stem and its features instead of morphs. For example, the Xerox parser output for *disentangled*, *happier*, and *Gründe* is:

```
disentangle+Verb+PastBoth+123SP
happy+Adj+Comp
Grund+Noun+Masc+Pl+NomAccGen
```

where the feature `+Verb` denotes a verb, `+PastBoth`, either past tense or past participle, and `+123SP` any person, singular or plural; `+Adj` denotes an adjective and `+Comp`, a comparative; `+Noun` denotes a noun, `+Masc` masculine, `+Pl`, plural, and `+NomAccGen` either nominative, accusative, or genitive. (All these forms are ambiguous, and the Xerox parser shows more than one interpretation per form.)

Given these new lexical forms, the parser has to align the feature symbols with letters or null symbols. The principles do not change, however (Fig. 6.5).

6.4.3 *Finite-State Transducers*

The two-level model is commonly implemented using finite-state transducers (FST). Transducers are automata that accept, translate, or generate pairs of strings. The arcs are labeled with two symbols: the first symbol is the input and the second is the output. The input symbol is transduced into the output symbol as a transition occurs

Lexical:	d	i	s	e	n	t	a	n	g	l	e	+Verb	+PastBoth	+123sp
Surface:	d	i	s	e	n	t	a	n	g	l	0	0	e	d

Lexical:	h	a	p	p	y	+Adj	+Comp
Surface:	h	a	p	p	i	e	r

Lexical:	G	r	u	n	d	+Noun	+Masc	+Pl	+NomAccGen
Surface:	G	r	ü	n	d	0	0	0	e

Fig. 6.5 Alignments with features

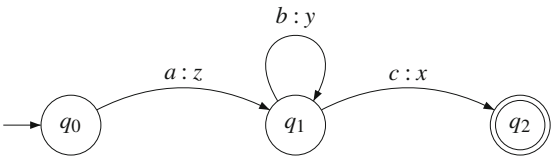


Fig. 6.6 A transducer

on the arc. For instance, the transducer in Fig. 6.6 accepts or generates the string *abbbc* and translates into *zyyyx*.

Finite-state transducers have a formal definition, which is similar to that of finite-state automata. A FST consists of five components $(Q, \Sigma, q_0, F, \delta)$, where:

1. Q is a finite set of states.
2. Σ is a finite set of symbol or character pairs $i : o$, where i is a symbol of the input alphabet and o of the output alphabet. As we saw, both alphabets may include epsilon transitions.
3. q_0 is the start state, $q_0 \in Q$.
4. F is the set of final states, $F \subseteq Q$.
5. δ is the transition function $Q \times \Sigma \rightarrow Q$, where $\delta(q, i, o)$ returns the state where the automaton moves when it is in state q and consumes the input symbol pair $i : o$.

The quintuple, which defines the automaton in Fig. 6.6 is $Q = \{q_0, q_1, q_2\}$, $\Sigma = \{a : z, b : y, c : x\}$, $\delta = \{\delta(q_0, a : z) = q_1, \delta(q_1, b : y) = q_1, \delta(q_1, c : x) = q_2\}$, and $F = \{q_2\}$.

Table 6.26 Future tense of French verb *chanter*

Number\Person	First	Second	Third
singular	*chanterai*	*chanteras*	*chantera*
plural	*chanterons*	*chanterez*	*chanteront*

Table 6.27 Aligned lexical and surface forms

Number\Pers.	First	Second	Third
singular	`chanter+erai`	`chanter+eras`	`chanter+era`
	`chant000erai`	`chant000eras`	`chant000era`
plural	`chanter+erons`	`chanter+erez`	`chanter+eront`
	`chant000erons`	`chant000erez`	`chant000eront`

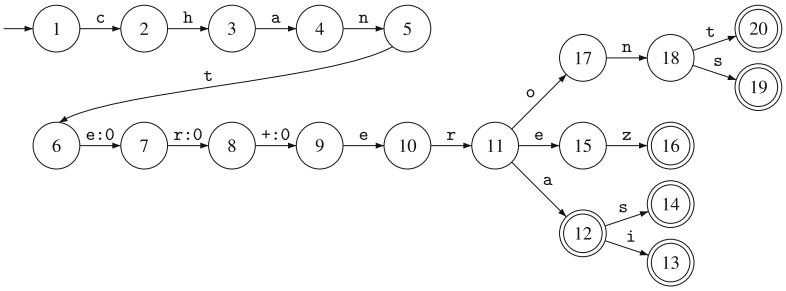


Fig. 6.7 A finite-state transducer describing the future tense of *chanter*

6.4.4 Conjugating a French Verb

Morphological FSTs encode the lexicon and express all the legal transitions. Arcs are labeled with pairs of symbols representing letters of the surface form – the word – and the lexical form – the set of morphs.

Table 6.26 shows the future tense of regular French verb *chanter* 'sing', where suffixes are specific to each person and number, but are shared by all the verbs of the so-called first group. The first group accounts for the large majority of French verbs. Table 6.27 shows the aligned forms and Fig. 6.7 the corresponding transducer. The arcs are annotated by the input/output pairs, where the left symbol corresponds to the lexical form and the right one to the surface form. When the lexical and surface characters are equal, as in `c:c`, we just use a single symbol in the arc.

This transducer can be generalized to any regular French verb of the first group by removing the stem part and inserting a self-looping transition on the first state (Fig. 6.8).

The transducer in Fig. 6.8 also parses and generates forms that do not exist. For instance, we can forge an imaginary French verb **palimoter* that still can be conjugated by the transducer. Conversely, the transducer will successfully parse the

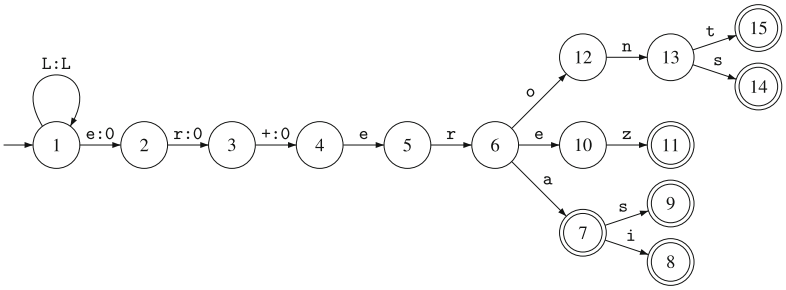


Fig. 6.8 A finite-state transducer describing the future tense of French verbs of the first group

improbable **palimoterons*. This process is called overgeneration (both in parsing and generation).

Overgeneration is not that harmful, provided that inputs are well formed. However, it can lead to some wrong parses. Consider English and German comparatives that are formed with *-er* suffix. Raw implementation of a comparative transducer would rightly parse *greater* as *great+er* but could also parse *better* or *reader*. Overgeneration is reduced by a lexical lookup, where the parse result is searched in a dictionary. This eliminates nonexistent words. It can also be limited by a set of constraints on affixes restricting the part of speech of the word to which they can be appended – here adjectives.

6.4.5 Prolog Implementation

Finite-state transducers can easily be implemented in Prolog. In this section, we implement the future tense of regular French verbs corresponding to Fig. 6.8, and we remove null symbols by inserting a mute transition in the surface form. The transducer has four parameters: the start state, normally 1, a final state, together with a lexical form and a surface one:

```
transduce(+Start, ?Final, ?Lexical, ?Surface).
```

The transducer parses surface forms:

```
?- transduce(1, Final, Lexical, [r, ê, v, e, r, a]).
  Final = 7,
  Lexical = [r, ê, v, e, r, +, e, r, a]
```

It also generates surface forms from lexical ones:

```
?- transduce(1, Final,
    [r, ê, v, e, r, +, e, r, e, z], Surface).
  Final = 11,
```

```
  Surface = [r, ê, v, e, r, e, z]
```

Finally, the transducer conjugates verbs (generates the verbal forms):

```
?- transduce(1, 11, [r, ê, v, e, r | L], Surface).
  L = [+, e, r, e, z],
  Surface = [r, ê, v, e, r, e, z]
```

Here is the Prolog code:

```
% arc(Start, End, LexicalChar, SurfaceChar)
%  describes the automaton

arc(1, 1, C, C) :- letter(C).
arc(1, 2, e, 0).   arc(2, 3, r, 0).  arc(3, 4, +, 0).
arc(4, 5, e, e).   arc(5, 6, r, r).  arc(6, 7, a, a).
arc(7, 8, i, i).   arc(7, 9, s, s).
arc(6, 10, e, e).  arc(10, 11, z, z).
arc(6, 12, o, o).  arc(12, 13, n, n).
arc(13, 14, s, s). arc(13, 15, t, t).

% final_state(S)
%  gives the stop condition

final_state(7).   final_state(8).   final_state(9).
final_state(11).  final_state(14).  final_state(15).

% letter(+L)
%  describes the French lower-case letters

letter(L) :-
  name(L, [Code]),
  97 =< Code, Code =< 122, !.
letter(L) :-
  member(L,
    [à, â, ä, ç, é, è, ê, ë, î, ï, ô, ö, ù, û, ü, 'œ']),
  !.

% transduce(+Start, ?Final, ?LexicalString, ?SurfaceString)
%  describes the transducer. The first and second rules
%  include mute transitions and enable to remove 0s

transduce(Start, Final, [U | LexicalString], SurfaceString) :-
  arc(Start, Next, U, 0),
  transduce(Next, Final,LexicalString,SurfaceString).
transduce(Start, Final, LexicalString,
    [S | SurfaceString]) :-
  arc(Start, Next, 0, S),
  transduce(Next, Final,LexicalString,SurfaceString).
```

Table 6.28 Future tense of Italian verb *cantare* and Spanish and Portuguese verbs *cantar*, 'sing'

Language	Number\Person	First	Second	Third
Italian				
	singular	*canterò*	*canterai*	*canterà*
	plural	*canteremo*	*canterete*	*canteranno*
Spanish				
	singular	*cantaré*	*cantarás*	*cantará*
	plural	*cantaremos*	*cantaréis*	*cantarán*
Portuguese				
	singular	*cantarei*	*cantarás*	*cantará*
	plural	*cantaremos*	*cantareis*	*cantarão*

```
transduce(Start, Final, [U | LexicalString],
    [S | SurfaceString]) :-
  arc(Start, Next, U, S),
  U \== 0,
  S \== 0,
  transduce(Next, Final,LexicalString,SurfaceString).
transduce(Final, Final, [], []) :-
  final_state(Final).
```

We can associate a final state to a part of speech. For instance, state 11 corresponds to the second-person plural of the future.

6.4.6 Application to Romance Languages

The transducer we created for the conjugation of French verbs can be easily transposed to other Romance languages such as Italian, Spanish, or Portuguese, as shown in Table 6.28.

6.4.7 Ambiguity

In the transducer for future tense, there is no ambiguity. That is, a surface form has only one lexical form with a unique final state. This is not the case with the present tense (Table 6.29), and

(je) chante 'I sing'
(il) chante 'he sings'

have the same surface form but correspond, respectively, to the first- and third-person singular.

This corresponds to the transducer in Fig. 6.9, where final states 5 and 7 are the same. The implementation in Prolog is similar to that of the future tense.

Table 6.29 Present tense of French verb *chanter*

Number\Person	First	Second	Third
singular	*chante*	*chantes*	*chante*
plural	*chantons*	*chantez*	*chantent*

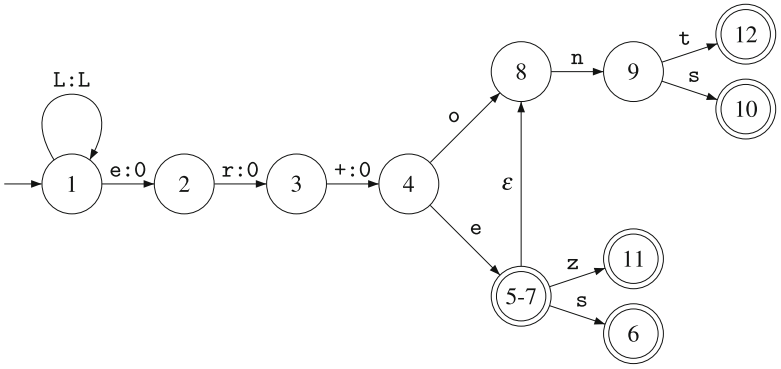


Fig. 6.9 A finite-state transducer encoding the present tense of verbs of the first group

Using backtracking, the transducer can yield all the final states reflecting the morphological ambiguity.

6.4.8 Operations on Finite-State Transducers

Finite-state transducers have mathematical properties similar to those of finite-state automata. In addition, they can be inverted and composed:

- Let T be a transducer. The inversion T^{-1} reverses the input and output symbols of the transition function. The transition function of the transducer in Fig. 6.6 is then $\delta = \{\delta(q_0, z : a) = q_1, \delta(q_1, y : b) = q_1, \delta(q_1, x : c) = q_2\}$.
- Let T_1 and T_2 be two transducers. The composition $T_1 \circ T_2$ is a transducer, where the output of T_1 acts as the input of T_2.

Both the inversion and composition operations result in new transducers. This is obvious for the inversion. The proof is slightly more complex for the composition. Let $T_1 = (\Sigma, Q_1, q_1, F_1, \delta_1)$ and $T_2 = (\Sigma, Q_2, q_2, F_2, \delta_2)$ be two transducers. The composition $T_3 = T_1 \circ T_2$ is defined by $(\Sigma, Q_1 \times Q_2, \langle q_1, q_2 \rangle, F_1 \times F_2, \delta_3)$. The transition function δ_3 is built using the transition functions δ_1 and δ_2, and generating all the pairs where they interact (Kaplan and Kay 1994):

$$\delta_3(\langle s_1, s_2 \rangle, i, o) = \{\langle t_1, t_2 \rangle | \exists c \in \Sigma \cup \varepsilon, t_1 \in \delta_1(s_1, i, c) \wedge t_2 \in \delta_2(s_2, c, o)\}.$$

The inversion property enables transducers to operate in generating or parsing mode. They accept both surface and lexical strings. Each symbol of the first string is mapped to the symbol of the second string. So you can walk through the automaton and retrieve the lexical form from the surface form, or conversely, as we saw with the Prolog example.

Composition enables us to break down morphological phenomena. It is sometimes easier to formulate a solution then using intermediate forms between the surface and lexical forms. The correspondence between the word form and the sequence of morphemes is not direct but is obtained as a cascade of transductions. Composition enables us to compact the cascade and to replace the transducers involved in it by a single one (Karttunen et al. 1992). We will see an example of it with French irregular verbs in Sect. 6.5.3.

6.5 Morphological Rules

6.5.1 Two-Level Rules

Originally, Koskenniemi (1983) used declarative rules to describe morphology. These two-level rules enumerate the correspondences between lexical characters and surface ones and the context where they occur. Context corresponds to left and right characters of the current character and can often be expressed in terms of vowels (*V*) or consonants (*C*).

In the two-level formalism, a rule is made of a correspondence pair (`lexical:surface`), a rule operator, and the immediate left and right context. Operators can be ⇒, ⇐, ⇔, or /⇐, and mean, respectively, only in that context, always in that context, always and only, and never in that context. Left and right contexts where the rule applies are separated by the symbol __ (Table 6.30).

In English, the comparative *happier* is decomposed into two morphemes *happy* + *er*, where the lexical *y* corresponds to a surface *i* (Table 6.31). This correspondence occurs more generally when *y* is preceded by a consonant and followed by *-er*, *-ed*, or *-s*. This can be expressed by three rules, where `C` represents any consonant:

1. `y:i ⇐ C:C __ +:0 e:e r:r`
2. `y:i ⇐ C:C __ +:e s:s`
3. `y:i ⇐ C:C __ +:0 e:e d:d`

Once written, all the rules are applied in parallel. This parallel application is the main distinctive feature of the two-level morphology compared with other, older models. This means that when processing a string, every rule must be successfully applied to the current pair of characters `lexical:surface` before moving to the next pair (Fig. 6.10).

Table 6.30 Two-level rules

Rules			Description
`a:b`	⇒	`lc __ rc`	`a` is transduced as `b` **only** when it has `lc` to the left and `rc` to the right
`a:b`	⇐	`lc __ rc`	`a` is **always** transduced as `b` when it has `lc` to the left and `rc` to the right
`a:b`	⇔	`lc __ rc`	`a` is transduced as `b` **always and only** when it has `lc` to the left and `rc` to the right
`a:b`	/⇐	`lc __ rc`	`a` is **never** transduced as `b` when it has `lc` to the left and `rc` to the right

Table 6.31 The `y:i` transduction rules

Examples	`happy+er`	`party+s`	`marry+ed`
	`happi0er`	`parties`	`marri0ed`
Rules	`Cy+er`	`Cy+s`	`Cy+ed`
	`Ci0er`	`Cies`	`Ci0ed`

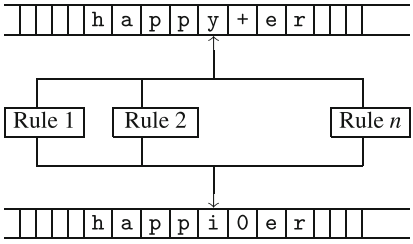


Fig. 6.10 Applying the rules in parallel

The left and right contexts of a rule can use a wildcard, the ANY symbol `@`, which stands for any alphabetical character, as in

```
y:x ⇐ __ @:c
```

This rule means that a lexical `y` corresponds to a surface `x` when it is before a surface `c`. The corresponding lexical character in the right context is not specified in the rule, however, the unspecified character represented by the ANY symbol must be compatible with the correspondence rule that can apply to it. The ANY symbol is not, strictly speaking, any character then, but any character so that it forms a "feasible pair", here with `c`.

6.5.2 Rules and Finite-State Transducers

It has been demonstrated that any two-level rule can be compiled into an equivalent transducer (Johnson 1972; Kaplan and Kay 1994). Rule 1, for instance, corresponds to the automaton in Fig. 6.11, where the pair `@:@` denotes any pair that cannot pass the other transitions.

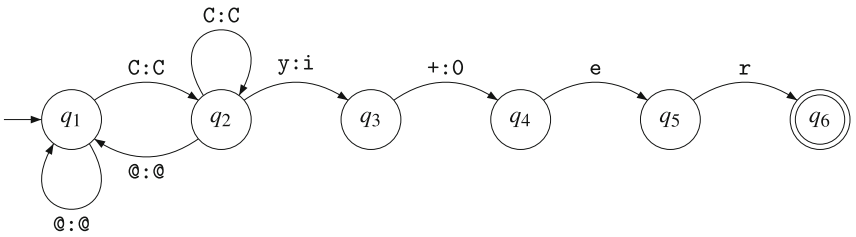


Fig. 6.11 A transducer to parse the `y:i` correspondence

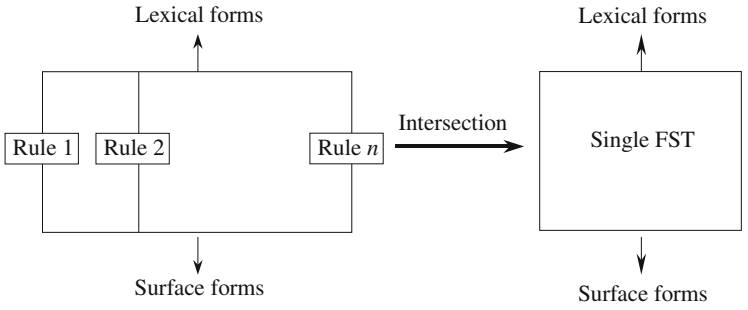


Fig. 6.12 A set of two-level rules intersected into a single FST (After Karttunen et al. (1992))

In practice, morphological phenomena are easier to describe and to understand using individual rules rather than writing a complex transducer. For this reason, the development of parsers based on the two-level method uses this strategy (Karttunen 1994). It consists in writing a collection of rules to model a language's morphology and compiling them into as many transducers. The parallel transducers are then combined into a single one using the transducer intersection (Fig. 6.12).

However, while the intersection of two finite automata defines a finite-state automaton, it is not always the case for finite-state transducers. Kaplan and Kay (1994) demonstrated that when surface and lexical pairs have the same length – without ε – the intersection is a transducer. This property is sufficient to intersect the rules in practical applications. In fact, transducers obtained from two-level rules are intersected by treating the ε symbol as an ordinary symbol (Beesley and Karttunen 2003, p. 55). Parallel application of rules or the transducer intersection removes one of their major harmful side effects: their application outside of their intended context.

Originally, rules were compiled by hand. However, this problem can quickly become intractable, especially when it comes to managing conflicting rules or when rule contexts interfere with transduced symbols. To solve it, we can use a compiler that creates transducers automatically from two-level rules. The Xerox XFST is one such compiler. It is a publicly available tool, and to date it is the only serious implementation of a morphological rule compiler.

Table 6.32 Conjugation of irregular French verbs, present tense. *Courir* has regular suffixes in *underlined bold characters*. In the other verbs, irregular infections are shown in *bold characters*

Infinitive	**courir**	**dormir**	**battre**	**peindre**	**écrire**
First person singular	cour**s**	**dors**	**bats**	**peins**	écris
Second person singular	cour**s**	**dors**	**bats**	**peins**	écris
Third person singular	cour**t**	**dort**	**bat**	**peint**	écrit
First person plural	cour**ons**	dormons	battons	peignons	**écrivons**
Second person plural	cour**ez**	dormez	battez	peignez	**écrivez**
Third person plural	cour**ent**	dorment	battent	peignent	**écrivent**

6.5.3 *Rule Composition: An Example with French Irregular Verbs*

When developing a complete morphological parser, it is often convenient to introduce intermediate levels between the lexical and surface strings. This is especially true when the lexical and surface forms are distant and involve complex morphological relations. Intermediate levels enable us then to decompose the morphological system into smaller parts that are easier to treat.

Chanod (1994) gives an example of decomposition with the notoriously difficult morphology of French irregular verbs (Bescherelle 1980). The French verb system has about 100 models of inflection – paradigms. Two of them are said to be regular, the first and second group, and gather the vast majority of the verbs. The third group is made of irregular verbs and gathers the rest. The irregular group contains the most frequent verbs: *faire* 'do', *savoir* 'know', *connaître* 'know', *dormir* 'sleep', *courir* 'run', *battre* 'beat', *écrire* 'write', etc.

Table 6.32 shows the conjugation of some irregular verbs. We can see that there is a set of regular suffixes: *s*, *s*, *t*, *ons*, *ez*, and *ent*, and that most irregularities, also called alternations, occur at the junction of the stem and the suffix. The stem and suffix can be directly concatenated, as in *courir*, but not in *dormir*, *peindre*, or *battre*.

Although apparently complex, general rules can model these alternations using local contexts corresponding to specific substrings. In the case of *dormir*, a general principle in French makes it impossible to have an *m* followed by an *s* or *t*. It then must be deleted in the three singular persons. For *battre*, the pairs *tt* or *dt* do not occur in the end of a word or before a final *s*. Such rules are not tied to one specific verb but can be applied across a variety of inflection paradigms and persons. Figure 6.13 shows the rule sequence that produces the correct surface form of *dors*.

The verbs *peindre* and *écrire* are more complex cases because their conjugation uses two stems: *pein* and *peign* – *écri* and *écriv*. Chanod (1994) solves these difficulties using a transduction between the infinitive and a first intermediate form that will then be regular. Then *peindre*+IndP+SG+P1 is associated to *peign*+IndP+SG+P1, and *écrire*+IndP+SG+P1 to *écriv*+IndP+SG+P1. The second intermediate form uses two-level rules to obtain the correct surface forms: *v* or *gn* must be followed by a vowel or deleted (Fig. 6.14). The rule that Chanod uses is, in fact:

Lexical form: stem	dormir	+IndP +SG +P1
Intermediate form: inflection	dorm	+IndP +SG +P1
Intermediate form: deletion of *m* followed by *s*	dorm	s
Surface form:	dor	s

Fig. 6.13 Sequence of rules applied to *dormir* (After Chanod (1994))

Lexical form: stem	peindre	+IndP +SG +P1
Intermediate form: inflection	peign	+IndP +SG +P1
Intermediate form: Depalatalisation of *gn*	peign	s
Surface form:	pein	s

Fig. 6.14 Sequence of rules applied to *peindre* (After Chanod (1994))

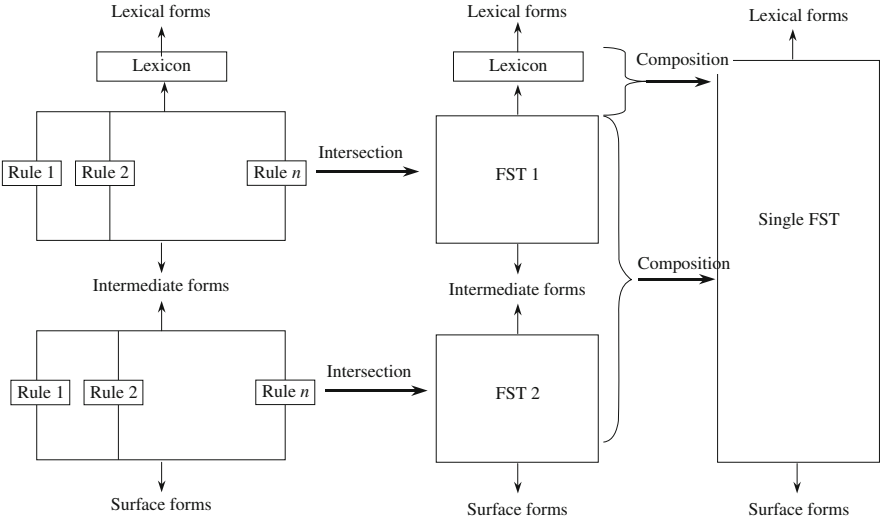


Fig. 6.15 Intersection and composition of finite-state transducers (After Karttunen (1994))

```
n:0 ⇔ g __ [s|t]
```

The FST resulting from the surface, lexical, and intermediate levels are ultimately combined with the lexicon and composed into a single transducer (Fig. 6.15).

6.6 The CoNLL Format

The Conference on Natural Language Learning (CoNLL) is an annual conference dedicated to statistical and machine-learning techniques in language analysis. In addition to the classical contributions in the form of articles found in scientific conferences, CoNLL organizes a "shared task" to evaluate language processing systems on a specific problem. As hinted by the conference name, the competing systems should use machine-learning techniques, essentially supervised learning. The participants are given a training set to train their models and are evaluated on a test set. CoNLL makes these data sets available in a column-based format as shown in Tables 6.33 and 6.34.

The CoNLL format has become very popular to share annotated corpora. It can be used to represent any layer of a linguistic analysis: part-of-speech tagging, morphological parsing, dependency parsing, semantic parsing, coreference, etc. Although the number of columns may vary depending on the task, this format shows a common structure across the years. In this section, we introduce it and we focus on the morphological and part-of-speech layers. We will review additional columns in the next chapters of this book.

Table 6.33 exemplifies the CoNLL format with the Spanish sentence *La reestructuración de los otros bancos checos se está acompañando por la reducción del personal* 'The restructuring of Czech banks is accompanied by the reduction of personnel' (Palomar et al. 2004). One of the main characteristics of the CoNLL format is that it has one word per line with the word properties and annotation shown on the same line in separate columns:

- The FORM column corresponds to the word;
- The LEMMA column contains the lemma and the phrase *los otros bancos* starting at index 4 is lemmatized as *el otro banco*;
- The CPOS and POS columns correspond to a coarse version of the part of speech and a more detailed one: *los* is a determiner (d) as well as *otros*; *bancos* is a noun (n); the POS codes are specific to this Spanish corpus, the Cast3LB treebank, and described in Civit Torruella (2002);
- Finally, the FEATS column corresponds to the grammatical features that are listed as an unordered set separated by vertical bars. The word *bancos* 'banks' has a masculine gender (gen = m) and a plural number (num = p).

The columns are delimited by a tabulation character, and the sentences by a blank line.

Table 6.34 shows a similar annotation with a sentence from a French corpus (Abeillé and Clément 2003; Abeillé et al. 2003) converted to the CoNLL format by Candito et al. (2009).

Table 6.33 Annotation of the Spanish sentence: *La reestructuración de los otros bancos checos se está acompañando por la reducción del personal* 'The restructuring of Czech banks is accompanied by the reduction of personnel' (Palomar et al. 2004) using the CoNLL 2006 format

ID	FORM	LEMMA	CPOS	POS	FEATS
1	La	el	d	da	num=s|gen=f
2	reestructuración	reestructuración	n	nc	num=s|gen=f
3	de	de	s	sp	for=s
4	los	el	d	da	gen=m|num=p
5	otros	otro	d	di	gen=m|num=p
6	bancos	banco	n	nc	gen=m|num=p
7	checos	checo	a	aq	gen=m|num=p
8	se	se	p	p0	_
9	está	estar	v	vm	num=s|per=3|mod=i|tmp=p
10	acompañando	acompañar	v	vm	mod=g
11	por	por	s	sp	for=s
12	la	el	d	da	num=s|gen=f
13	reducción	reducción	n	nc	num=s|gen=f
14	del	del	s	sp	gen=m|num=s|for=c
15	personal	personal	n	nc	gen=m|num=s
16	.	.	F	Fp	_

Table 6.34 Annotation of the French sentence: *À cette époque, on avait dénombré cent quarante candidats* 'At that time, we had counted one hundred and forty candidates' (Abeillé and Clément 2003; Abeillé et al. 2003) following the CoNLL 2006 format

Index	Form	Lemma	CPOS	POS	Features
1	À	à	P	P	_
2	cette	ce	D	DET	g=f|n=s|s=dem
3	époque	époque	N	NC	g=f|n=s|s=c
4	,	,	PONCT	PONCT	s=w
5	on	on	CL	CLS	g=m|n=s|p=3|s=suj
6	avait	avoir	V	V	m=ind|n=s|p=3|t=impft
7	dénombré	dénombrer	V	VPP	g=m|m=part|n=s|t=past
8	cent_quarante	cent_quarante	D	DET	g=m|n=p|s=card
9	candidats	candidat	N	NC	g=m|n=p|s=c
10	.	.	PONCT	PONCT	s=s

6.7 Application Examples

The Xerox language tools give a good example of what morphological parsers and part-of-speech taggers can do. These parsers are available for demonstration on the Internet using a web browser. Xerox tools let you enter English, French, German, Italian, Portuguese, and Spanish words, and the server returns the context-free morphological analysis for each term (Tables 6.35–6.37). You can also type in phrases or sentences and Xerox taggers will disambiguate their part of speech.

Table 6.35 Xerox morphological parsing in English

Input Term(s): works	Input Term(s): round	Input Term(s): this
work+Vsg3	round+Vb	this+Psg
work+Npl	round+Prep	this+Dsg
	round+Adv	this+Adv
	round+Adj	
	round+Nsg	

Table 6.36 Xerox morphological parsing in French

Input Term(s): étions	Input Term(s): porte
être+IndI+PL+P1+Verb	porter+SubjP+SG+P1+Verb
	porter+SubjP+SG+P3+Verb
	porter+Imp+SG+P2+Verb
	porter+IndP+SG+P1+Verb
	porter+IndP+SG+P3+Verb
	porte+Fem+SG+Noun

Table 6.37 Xerox morphological parsing in German

Input Term(s): arbeite	Input Term(s): die
arbeiten+V+IMP+PRÄS+SG2	die+ART+DEF+PL+NOM
arbeiten+V+IND+PRÄS+SG1	die+ART+DEF+SG+AKK+FEM
arbeiten+V+KONJ+PRÄS+SG1	die+ART+DEF+SG+NOM+FEM
arbeiten+V+KONJ+PRÄS+SG3	die+ART+DEF+PL+AKK
	die+PRON+DEM+PL+AKK
	die+PRON+DEM+PL+NOM
	die+PRON+DEM+SG+AKK+FEM
	die+PRON+DEM+SG+NOM+FEM
	die+PRON+RELAT+PL+AKK
	die+PRON+RELAT+PL+NOM
	die+PRON+RELAT+SG+AKK+FEM
	die+PRON+RELAT+SG+NOM+FEM

In addition to demonstrations, Xerox lists examples of industrial applications that make use of its tools.

6.8 Further Reading

Dionysius Thrax fixed the parts of speech for Greek in the second century BCE. They have not changed since and his grammar is still interesting to read, see Lallot (1998). A short and readable introduction in French to the history of parts of speech is Ducrot and Schaeffer (1995).

Accounts on finite-state morphology can be found in Sproat (1992) and Ritchie et al. (1992). Roche and Schabes (1997) is a useful book that describes fundamental algorithms and applications of finite-state machines in language processing, espe-

cially for French. Kornai (1999) covers other aspects and languages. Kiraz (2001) on the morphology of Semitic languages: Syriac, Arabic, and Hebrew. Beesley and Karttunen (2003) is an extensive description of the two-level model in relation with the Xerox tools. It contains a CD-ROM with the Xerox rule compiler.

Antworth (1995) provides a free implementation of KIMMO named PC-KIMMO 2 with source and executable programs. The system is available from the Internet (http://www.sil.org/). It comes with an English lexicon and English morphological rules. It is open to extensions and modifications. General-purpose finite-state transducers toolkits are also available. They include the FSA utilities (van Noord and Gerdemann 2001), the FSM library (Mohri et al. 1998) and its follower, OpenFst (http://www.openfst.org/), the Helsinki Finite-State Transducer Technology (https://kitwiki.csc.fi/twiki/bin/view/KitWiki/HfstHome), and Unitex (http://www-igm.univ-mlv.fr/~unitex/) (Paumier 2006).

Although most lemmatizers use transducers and rules, it is possible to formulate lemmatization with classifiers that we can train on annotated corpora. See Chrupała (2006) for an interesting account on these techniques and Björkelund et al. (2010) for an implementation.

Exercises

6.1. Find a dictionary on the Web in English, French, German, or another language you would like to study and extract all the articles, conjunctions, prepositions, and pronouns.

6.2. Implement a morphological parser to analyze regular plurals of nouns in English or French.

6.3. Add a lexical look-up to Exercise 6.2.

6.4. Implement a morphological parser to analyze plurals of nouns in English or French, taking a list of exceptions into account.

6.5. Implement a morphological parser to analyze regular preterits of verbs in English or German.

6.6. Implement a morphological parser to conjugate French verbs of first group in the imperfect tense.

6.7. Implement a morphological parser to conjugate regular German verbs in the present tense.

6.8. Build a morphological parser implementing regular English verb inflection: *-s*, *-ed*, *-ing*.

6.9. Some verbs have their final *-e* deleted, for instance, *chase* (*chase+ed, chase+ing*). In the KIMMO formalism, the *-e* deletion rule is expressed as e:0

⇔ C:C __ 0:+ V:V. Draw the corresponding transducer and write the Prolog rules that will parse these verbs.

6.10. Break the following words into morphemes: *computer*: *computers*, *computerize, computerization*, *recomputerize*.

6.11. Build a morphological parser that will parse words derived from *computer*: *computers*, *computerize, computerization*, *recomputerize*.

6.12. Break the following words into morphemes: *chanter*: *enchanter*, *rechanter*, *déchanter*, *désenchanter*.

6.13. Build a morphological parser that will parse words derived from *chanter*: *enchanter*, *rechanter*, *déchanter*, and *désenchanter*.

Chapter 7
Part-of-Speech Tagging Using Rules

7.1 Resolving Part-of-Speech Ambiguity

7.1.1 A Manual Method

We saw that looking up a word in a lexicon or carrying out a morphological analysis on a word can leave it with an ambiguous part of speech. The word *chair*, which can be assigned two tags, noun or verb, is an example of ambiguity. It is a noun in the phrase *a chair*, and a verb in *to chair a session*. Ambiguity resolution, that is, retaining only one part of speech (POS) and discarding the others, is generally referred to as POS tagging or POS annotation.

As children, we learned to carry out a manual disambiguation by considering the grammatical context of the word. In the first phrase, *chair* is preceded by an article and therefore is part of a noun phrase. Since there is no other word here, *chair* is a noun. In the second phrase, *chair* is preceded by *to*, which would not precede a noun, and therefore *chair* is a verb.

Voutilainen and Järvinen (1995) describe a more complex example with the sentence

> That round table might collapse.

While the correct part-of-speech tagging is:

> *That*/determiner *round*/adjective *table*/noun *might*/modal verb *collapse*/verb.

a simple dictionary lookup or a morphological analysis produces many ambiguities, as shown in Table 7.1.

P.M. Nugues, *Language Processing with Perl and Prolog*, Cognitive Technologies,
DOI 10.1007/978-3-642-41464-0_7, © Springer-Verlag Berlin Heidelberg 2014

Table 7.1 Ambiguities in part-of-speech annotation with the sentence: *That round table might collapse*

Words	Possible tags	Example of use
that	Subordinating conjunction	*That he can swim is good*
	Determiner	*That white table*
	Adverb	*It is not that easy*
	Pronoun	*That is the table*
	Relative pronoun	*The table that collapsed*
round	Verb	*Round up the usual suspects*
	Preposition	*Turn round the corner*
	Noun	*A big round*
	Adjective	*A round box*
	Adverb	*He went round*
table	Noun	*That white table*
	Verb	*I table that*
might	Noun	*The might of the wind*
	Modal verb	*She might come*
collapse	Noun	*The collapse of the empire*
	Verb	*The empire can collapse*

7.1.2 Which Method to Use to Automatically Assign Parts of Speech

Grammatical constraints are not always sufficient to resolve ambiguous tags. Church and Mercer (1993) exemplify this with the phrase *I see a bird*, which can be annotated as

I/noun *see*/noun *a*/noun *bird*/noun

This tagging corresponds to: *I*/letter of alphabet, *see*/noun as in *Holy See*, *a*/letter of alphabet, *bird*/noun. Although this tag sequence makes no sense here, it cannot be ruled out as syntactically ill formed, because the parser must accept sequences of four nouns in other situations, as in *city school committee meeting*. The proper tagging is, of course, *I*/pronoun *see*/verb *a*/article *bird*/noun.

Semantic rules could implement common-sense reasoning and prevent inconsistencies. However, this method is no longer favored. It would imply writing many rules that could operate in very specific applications, and not on unrestricted texts.

Instead of using general grammar rules, we can consider word preferences. Most words taken from a dictionary have only one part of speech or have a strong preference for only one of them, although frequent words tend to be more ambiguous. From text statistics based on different corpora, in English and in French, Merialdo (1994) and Vergne (1999) report that 50 to 60 % of words have a unique possible tag, and 15 to 25 % have only two tags. In both languages, tagging a word with its most common part of speech yields a success rate of more than 75 %. Charniak (1993) reports a score of more than 90 % for English. This figure is called

the **baseline**. It corresponds to the accuracy obtained with a minimal algorithm, here the word annotation with its most frequent tag.

Two efficient methods applied locally have emerged to improve this figure and to solve reasonably well POS tagging. The first one uses rule-based constraints. Rules consider the left and right context of the word to disambiguate, that is, either discard or replace a wrong part of speech. Rules are symbolic and can be designed by hand or derived automatically from hand-annotated corpora.

The second method is based on statistics. Sequence statistics are automatically learned from hand-annotated corpora, and probabilistic models are applied that assign the most likely tags to words of a sentence. Both methods enable us to successfully tag more than 95 % of the words of a text. We will describe the first one in this chapter and the second one in the next chapter.

7.2 Baseline

Before we start writing elaborate tagging algorithms, we describe first a baseline technique that requires extremely limited efforts. The baseline term is widely used in natural language processing to refer to a starting point that is usually easy to implement. We use then the results obtained from the baseline to assess the improvements brought by more elaborate algorithms.

In part-of-speech tagging, the baseline is to tag each word with its most frequent part of speech. We can derive the frequencies from a part-of-speech annotated corpus, such as the Penn Treebank for English. Surdeanu et al. (2008) converted it to the CoNLL format that we described in Sect. 6.6. Using Unix commands such as those in Sect. 5.5 makes the task very easy. In CoNLL 2008, the word forms and their parts of speech are respectively in column 2 and 5. We extract them with the `cut -f2,5` command and then we sort and count the lines. This can be done with the command line:

```
cut -f2,5 <conll_corpus_file | sort | uniq -c
```

The next lines show an excerpt of the results where the first column is the frequency, the second one, the word, and the third one, the part-of-speech code. The complete tagset is shown in Table 7.12:

```
    6 campus      NN
    2 campuses    NNS
  908 can         MD
    3 can         NN
    5 canal       NN
    1 canals      NNS
   13 cancel      VB
    7 canceled    VBD
   26 canceled    VBN
    3 cancellation       NN
```

Table 7.2 Initial step of Brill's algorithm

	Likely tags yielding a correct tagging	Likely tags yielding a wrong tagging
English	*I*/pro *can*/modal *see*/verb *a*/art *bird*/noun	*The*/art *can*/**modal** *rusted*/verb
French	*Je*/pro *donne*/verb *le*/art *cadeau*/noun	*Je*/pro *le*/**art** *fais*/verb *demain*/adv
German	*Der*/art *Mann*/noun *kommt*/verb	*Wer*/pro *ist*/verb *der*/art *Mann*/noun , *der*/**art** *kommt*/verb *?*

From these frequencies, the baseline algorithm will tag *can* as a modal (MD) and not as a noun (NN) and *canceled* as a verb past participle (VBN) and not as a verb preterit (VBD).

7.3 Tagging with Rules

Part-of-speech tagging with rules is relatively old (Klein and Simmons 1963). In the beginning, rules were hand-coded and yielded good results at the expense of thoroughly and painfully crafting the rules (Voutilainen et al. 1992). The field has been completely renewed by Brill (1995), who proposed a very simple scheme to tag a text with rules and an algorithm to automatically learn the rules from annotated corpora. A good deal of the current work on part-of-speech tagging with rules is now inspired by his foundational work.

7.3.1 Brill's Tagger

Brill's tagger uses a dictionary and assumes that it contains all the words to tag. Each word in the dictionary is labeled with its most likely (frequent) part of speech and includes the list of its other legal – possible – parts of speech. Part-of-speech distributions and statistics for each word can be derived from annotated corpora and by using methods described in Chaps. 2 and 5.

The tagger first assigns each word with its most likely part of speech as with the baseline. It does not depend on a morphological parser, although it could use one as a preprocessor. It also features a module to tag unknown words that we will examine in Sect. 7.4. Examples of likely tags assigned to words are given in Table 7.2.

The tagger then applies a list of transformations to alter the initial tagging. Transformations are contextual rules that rewrite a word tag into a new one. The transformation is performed only if the new tag of the word is legal – is in the dictionary. If so, the word is assigned the new tag. Transformations are executed sequentially and each transformation is applied to the text from left to right. Examples of transformations are:

1. In English: Change the tag from modal to noun if the previous word is an article.
2. In French: Change the tag from article to pronoun if the previous word is a pronoun.

Table 7.3 Contextual rule templates, where A, B, C, and D denotes parts of speech, members of the POS tagset

Rules	Explanation
`alter(A, B, prevtag(C))`	Change A to B if preceding tag is C
`alter(A, B, nexttag(C))`	Change A to B if the following tag is C
`alter(A, B, prev2tag(C))`	Change A to B if tag two before is C
`alter(A, B, next2tag(C))`	Change A to B if tag two after is C
`alter(A, B, prev1or2tag(C))`	Change A to B if one of the two preceding tags is C
`alter(A, B, next1or2tag(C))`	Change A to B if one of the two following tags is C
`alter(A, B, prev1or2or3tag(C))`	Change A to B if one of the three preceding tags is C
`alter(A, B, next1or2or3tag(C))`	Change A to B if one of the three following tags is C
`alter(A, B, surroundingtag(C, D))`	Change A to B if surrounding tags are C and D
`alter(A, B, nextbigram(C, D))`	Change A to B if next bigram tag is C D
`alter(A, B, prevbigram(C, D))`	Change A to B if previous bigram tag is C D

3. In German: Change the tag from article to pronoun if the previous word is a noun (or a comma.)

These rules applied to the sentences in Table 7.2 yield:

1. In English: *The*/art *can*/noun *rusted*/verb
2. In French: *Je*/pro *le*/pro *fais*/verb *demain*/adv
3. In German: *Wer*/pro *ist*/verb *der*/art *Mann*/noun , *der*/pro *kommt*/verb *?*

Rules conform to a limited number of transformation types, called templates. For example, the rule

> Change the tag from modal to noun if the previous word is an article.

corresponds to template:

> Change the tag from X to Y if the previous tag is Z.

The tagger uses in total 11 templates shown in Table 7.3. Brill reports that less than 500 rules – instantiated templates – are needed in English to obtain an accuracy of 97 %.

7.3.2 Implementation in Prolog

We will exemplify the tagging algorithm with an implementation of two rule templates:

```
alter(A, B, prevtag(C))
alter(A, B, prev1or2tag(C))
```

These rules being instantiated in the form of:

```
alter(verb, noun, prevtag(art)).
alter(verb, noun, prev1or2tag(art)).
```

The first rule changes the tag from `verb` to `noun` if the previous word is an `article`, and the second changes the tag from `verb` to `noun` if one of the two previous words is an `article`. The second rule is more general than the first one. We give the code of the first one because it is easier to start with it.

The `tag` predicate enables us to alter an initially tagged text:

```
?- tag([the/art, holy/adj, see/verb], L).
L = [the/art, holy/adj, see/noun]

% tag(+InitialTaggedText, -TaggedText)
% Implementation of Brill's algorithm

tag(InitialTaggedText, TaggedText) :-
  bagof(alter(FromPOS, ToPOS, Condition),
  alter(FromPOS, ToPOS, Condition), Rules),
  forall(Rules, InitialTaggedText, TaggedText).

% Collect all the rules and apply them sequentially

forall([Rule | Rules], Text, TaggedText) :-
  apply(Rule, Text, AlteredText),
  forall(Rules, AlteredText, TaggedText).
forall([], TaggedText, TaggedText).

%Apply prevtag template
apply(alter(FromPOS, ToPOS, prevtag(POS)),
  [PrevWord/POS, Word/FromPOS | RemainingText],
  [PrevWord/POS, Word/ToPOS | RemainingText1] ) :-
  !,
  apply(alter(FromPOS, ToPOS, prevtag(POS)),
    [Word/ToPOS | RemainingText],
    [Word/ToPOS | RemainingText1] ).
apply(alter(FromPOS, ToPOS, prevtag(POS)),
  [X, Y| RemainingText], [X, Y| RemainingText1] ) :-
  apply(alter(FromPOS, ToPOS, prevtag(POS)),
    [Y| RemainingText], [Y | RemainingText1] ).
apply(alter(_, _, prevtag(_)), [X], [X]).

% Apply prev1or2tag template
% The first two rules take into account that the rule
% can apply to the second word of the text
```

```
apply(alter(FromPOS, ToPOS, prev1or2tag(POS)),
  [FirstWord/POS, Word/FromPOS | RemainingText],
  [FirstWord/POS, Word/ToPOS | RemainingText1] ) :-
  apply_aux(alter(FromPOS, ToPOS, prev1or2tag(POS)),
    [FirstWord/POS, Word/ToPOS | RemainingText],
    [FirstWord/POS, Word/ToPOS | RemainingText1] ).
apply(alter(FromPOS, ToPOS, prev1or2tag(POS)),
  [X, Y| RemainingText], [X, Y| RemainingText1] ) :-
  apply_aux(alter(FromPOS, ToPOS, prev1or2tag(POS)),
    [X, Y| RemainingText], [X, Y| RemainingText1] ).

apply_aux(alter(FromPOS, ToPOS, prev1or2tag(POS)),
  [Prev2Word/POS, Prev1Word/POS1, Word/FromPOS |
   RemainingText],
  [Prev2Word/POS, Prev1Word/POS1, Word/ToPOS |
   RemainingText1] ) :-
  !,
   apply_aux(alter(FromPOS, ToPOS, prev1or2tag(POS)),
    [Prev1Word/POS1, Word/ToPOS | RemainingText],
    [Prev1Word/POS1, Word/ToPOS | RemainingText1] ).
apply_aux(alter(FromPOS, ToPOS, prev1or2tag(POS)),
  [Prev2Word/POS2, Prev1Word/POS, Word/FromPOS |
  RemainingText], [Prev2Word/POS2, Prev1Word/POS,
  Word/ToPOS | RemainingText1] ) :-
  !,
  apply_aux(alter(FromPOS, ToPOS, prev1or2tag(POS)),
    [Prev1Word/POS, Word/ToPOS | RemainingText],
    [Prev1Word/POS, Word/ToPOS | RemainingText1] ).
apply_aux(alter(FromPOS, ToPOS, prev1or2tag(POS)),
  [X, Y, Z | RemainingText],
  [X, Y, Z| RemainingText1] ) :-
  apply_aux(alter(FromPOS, ToPOS, prev1or2tag(POS)),
    [Y, Z| RemainingText], [Y, Z | RemainingText1] ).
apply_aux(alter(FromPOS, ToPOS, prev1or2tag(POS)),
   [PrevWord/POS, Word/FromPOS],
     [PrevWord/POS, Word/ToPOS]).
apply_aux(alter(_, _, prev1or2tag(_)), [X,Y], [X,Y]).

%The ordered contextual rules
alter(verb, noun, prevtag(art)).
alter(verb, noun, prev1or2tag(art)).
```

Table 7.4 Brill's learning algorithm

St.	Operation	Input	Output
1.	Annotate each word of the corpus with its most likely part of speech	*Corpus*	*AnnotatedCorpus(1)*
2.	Compare pairwise the part of speech of each word of the *AnnotationReference* and *AnnotatedCorpus(i)*	*AnnotationReference* *AnnotatedCorpus(i)*	List of errors
3.	For each error, instantiate the rule templates to correct the error	List of errors	List of tentative rules
4.	For each instantiated rule, compute on *AnnotatedCorpus(i)* the number of good transformations minus the number of bad transformations the rule yields	*AnnotatedCorpus(i)* Tentative rules	Scored tentative rules
5.	Select the rule that has the greatest error reduction and append it to the ordered list of transformations	Tentative rules	*Rule(i)*
6.	Apply *Rule(i)* to *AnnotatedCorpus(i)*	*AnnotatedCorpus(i)* *Rule(i)*	*AnnotatedCorpus(i+1)*
7.	If number of errors is under predefined threshold, end the algorithm else go to step 2.	–	List of rules

7.3.3 Deriving Rules Automatically

One of the most interesting features of Brill's rules is that they can be learned automatically from a hand-annotated corpus. This type of algorithm is called transformation-based learning (TBL). Let us denote *Corpus* this corpus and *AnnotationReference* its hand-annotation. In the context, the hand-annotation is often called the **Gold Standard**.

The TBL algorithm first assigns the most likely (frequent) tag to each word. It produces errors, and all rules templates are instantiated for each tagging error measured against *AnnotationReference.* The rule that yields the greatest error reduction is selected and applied to alter the *Corpus* tagging. This process is iterated as long as the annotation results are not close enough to *AnnotationReference*.

Table 7.4 shows the steps of the algorithm. *Corpus* annotated at iteration i of the process is denoted *AnnotatedCorpus(i).* Each iteration enables us to derive a new rule, which is denoted *Rule(i).*

As hand-annotated corpus, Brill (1995) used the Penn Treebank (Marcus et al. 1993). Table 7.5 lists the five most productive rules that the algorithm learned from *The Wall Street Journal* annotated section of the corpus (Brill 1995).

Table 7.5 The five first transformations learned from *The Wall Street Journal* corpus (Brill 1995), where NN is a singular noun; VB is a verb, base form; TO is the word *to*; VBP is a verb, non-third person singular present; MD is a modal; DT is a determiner; VBD is a verb, past tense; and VBZ is a verb, third-person singular present. These tags are defined by the Penn Treebank, and Sect. 7.5.2 details the complete tagset

	Change		
#	From	To	Condition
1	NN	VB	Previous tag is TO
2	VBP	VB	One of the previous three tags is MD
3	NN	VB	One of the previous two tags is MD
4	VB	NN	One of the previous two tags is DT
5	VBD	VBN	One of the previous three tags is VBZ

Table 7.6 A confusion matrix. The first column corresponds to the correct tags, and for each tag, the rows give the assigned tags. Excerpt from Franz (1996, p. 124). IN is a preposition, RB is an adverb, JJ is an adjective, RP is a particle, VBG is a verb, gerund (complete tagset in Sect. 7.5.2)

	Tagger →									
↓Correct	DT	IN	JJ	NN	RB	RP	VB	VBD	VBG	VBN
DT	99.4	0.3	–	–	0.3	–	–	–	–	–
IN	0.4	97.5	–	–	1.5	0.5	–	–	–	–
JJ	–	0.1	93.9	1.8	0.9	–	0.1	0.1	0.4	1.5
NN	–	–	2.2	95.5	–	–	0.2	–	0.4	–
RB	0.2	2.4	2.2	0.6	93.2	1.2	–	–	–	–
RP	–	24.7	–	1.1	12.6	61.5	–	–	–	–
VB	–	–	0.3	1.4	–	–	96.0	–	–	0.2
VBD	–	–	0.3	–	–	–	–	94.6	–	4.8
VBG	–	–	2.5	4.4	–	–	–	–	93.0	–
VBN	–	–	4.6	–	–	–	–	4.3	–	90.6

7.3.4 Confusion Matrices

At each iteration of TBL algorithm, we can derive a confusion matrix that shows for each tag how many times a word has been wrongly labeled. Table 7.6 shows an example of it (Franz 1996), which enables us to understand and track errors. Again, parts of speech use the Penn Treebank tagset described in Sect. 7.5.2. The diagonal shows the breakdown of the tags correctly assigned, for example, 99.4 % for determiners (DT). The rest of the table shows the tags wrongly assigned, i.e., for determiners: 0.3 % to prepositions (IN) and 0.3 % to adverbs (RB). This table is only an excerpt, therefore the sum of rows is not equal to 100.

Table 7.7 The first five transformations for unknown words (Brill 1995), where NN is a noun, singular; NNS a noun, plural; CD cardinal number; JJ an adjective; VBN a verb, past participle; VBG a verb, gerund (complete tagset in Sect. 7.5.2)

	Change		
#	From	To	Condition
1	NN	NNS	Has suffix **s**
2	NN	CD	Has character **.**
3	NN	JJ	Has character **-**
4	NN	VBN	Has suffix **ed**
5	NN	VBG	Has suffix **ing**

7.4 Unknown Words

We have made the assumption of a finite vocabulary. This is never the case in practice. Many words will likely be absent from the dictionary: proper and common nouns, verbs, adjectives, or adverbs.

There is no standard technique to deal with the unknown words. The baseline is to tag unknown words as nouns since it is the most frequent part of speech. Another technique is to use suffixes. Brill (1995) proposes a combination of both to extend the transformation-based algorithm. The initial step tags unknown words as proper nouns for capitalized words and as common nouns for the rest. Then it applies transformations from a set of predefined templates: change the tag of an unknown word from X to Y if:

1. Deleting the prefix (suffix) x, $|x| \leq 4$, results in a word (x is any string of length 1 to 4).
2. The first (last) (1, 2, 3, 4) characters of the word are x.
3. Adding the character string x as a prefix (suffix) results in a word.
4. Word w ever appears immediately to the left (right) of the word.
5. Character z appears in the word.

These templates are specific to English, but they can easily be modified to accommodate other European languages. Table 7.7 shows the first five transformations learned from *The Wall Street Journal* corpus.

7.5 Standardized Part-of-Speech Tagsets

While basic parts of speech are relatively well defined: determiners, nouns, pronouns, adjectives, verbs, auxiliaries, adverbs, conjunctions, and prepositions, there is a debate on how to standardize them for a computational analysis. One issue is the level of detail. Some tagsets feature a dozen tags, some over a hundred. Another issue that is linked to the latter is that of subcategories. How many classes

Table 7.8 Parts of speech and grammatical features

Main parts of speech	Features (subcategories)
Adjective, noun, pronoun	Regular base comparative superlative interrogative person number case
Adverb	Regular base comparative superlative interrogative
Article, determiner, preposition	Person case number
Verb	Tense voice mood person number case

for verbs? Only one, or should we create auxiliaries, modal, gerund, intransitive, transitive verbs, etc.?

The debate becomes even more complicated when we consider multiple languages. In French and German, the main parts of speech can be divided into subclasses depending on their gender, case, and number. In English, these divisions are useless. Although it is sometimes possible to map tagsets from one language to another, there is no universal scheme, even within the same language.

A few years ago, many computational linguists had a personal tagset. There are now standards, but the discussion is not over. We will examine here two multilingual part-of-speech schemes, a widely accepted tagset for English (the Penn Treebank), and a tagset for Swedish.

7.5.1 Multilingual Part-of-Speech Tags

Building a multilingual tagset imposes the condition of having a set of common classes, which enables a comparison between languages. These classes correspond to traditional parts of speech and gather a relatively large consensus among European languages. However, they are not sufficiently accurate for any language in particular. Dermatas and Kokkinakis (1995) retained the traditional parts of speech to tag texts in seven European languages using statistical methods. They also added features (subcategories) specific to each language (Table 7.8).

MULTEXT (Ide and Véronis 1995; Monachini and Calzolari 1996), is a multinational initiative that aims at providing an annotation scheme for all the Western and Eastern European languages. For the parts of speech, MULTEXT merely perpetuates the traditional categories and assigns them a code. The universal part-of-speech tagset (Petrov et al. 2012) is more recent and almost identical, but includes a mapping with other tagsets used in annotated corpora of 22 different languages. Table 7.9 shows both tag sets.

MULTEXT complements the parts of speech with a set of grammatical features, which they call attributes. Attributes enable us to subcategorize words and reconcile specific features of different European languages. Attributes for nouns and verbs are shown in Tables 7.10 and 7.11.

Table 7.9 Part-of-speech codes from MULTEXT and the universal POS tagset

Part of speech	MULTEXT	Universal POS tagset
Nouns	N	NOUN
Verbs	V	VERB
Adjectives	A	ADJ
Pronouns	P	PRON
Determiners and articles	D	DET
Adverbs	R	ADV
Adposition (Prepositions and postpositions)	S	ADP
Conjunctions	C	CONJ
Numerals	M	NUM
Interjections	I	–
Residuals (abbreviations, foreign words, etc.)	X	X
Particles	–	PRT
Punctuation marks	–	.

Table 7.10 Features (attributes) and values for nouns

Position	Attribute	Value	Code
1	Type	Common	c
		Proper	p
2	Gender	Masculine	m
		Feminine	f
		Neuter	n
3	Number	Singular	s
		Plural	p
4	Case	Nominative	n
		Genitive	g
		Dative	d
		Accusative	a

MULTEXT attributes concern only the morpho-syntactic layer and represent a superset of what is needed by all the languages. Some attributes may not be relevant for a specific language. For instance, English nouns have no gender, and French ones have no case. In addition, applications may not make use of some of the attributes even if they are part of the language. Tense, for instance, may be useless for some applications.

A part-of-speech tag is a string where the first character is the main class of the word to annotate and then a sequence of attribute values. Attribute positions correspond to their rank in the table, such as those defined in Tables 7.10 and 7.11 for nouns and verbs. When an attribute is not applicable, it is replaced by a dash (-). An English noun could receive the tag:

```
N[type=common number=singular] Nc-s-
```

a French one:

```
N[type=common gender=masculine number=singular] Ncms-
```

Table 7.11 Attributes (features) and values for verbs

Position	Attribute	Value	Code
1	Type	Main	m
		Auxiliary	a
		Modal	o
2	Mood/form	Indicative	i
		Subjunctive	s
		Imperative	m
		Conditional	c
		Infinitive	i
		Participle	p
		Gerund	g
		Supine	s
3	Tense	Base	b
		Present	p
		Imperfect	i
		Future	f
		Past	s
4	Person	First	1
		Second	2
		Third	3
5	Number	Singular	s
		Plural	p
6	Gender	Masculine	m
		Feminine	f
		Neuter	n

and a German one:

```
N[type=common gender=neuter number=singular
    case=nominative] Ncnsn
```

A user can extend the coding scheme and add attributes if the application requires it. A noun could be tagged with some semantic features such as country names, currencies, etc.

Finally, although they are not encoded the same way, the parts of speech and grammatical features in Tables 6.33 and 6.34 are roughly equivalent to the MULTEXT annotation.

7.5.2 Parts of Speech for English

The Penn Treebank is a large corpus of texts annotated with part-of-speech and syntactic tags (Marcus et al. 1993). Table 7.12 shows its part-of-speech tagset consisting of 48 tags and Table 7.13 shows an annotation example with the sentence:

Table 7.12 The Penn Treebank tagset

1.	CC	Coordinating conjunction	25.	TO	*to*
2.	CD	Cardinal number	26.	UH	Interjection
3.	DT	Determiner	27.	VB	Verb, base form
4.	EX	Existential *there*	28.	VBD	Verb, past tense
5.	FW	Foreign word	29.	VBG	Verb, gerund/present participle
6.	IN	Preposition/sub. conjunction	30.	VBN	Verb, past participle
7.	JJ	Adjective	31.	VBP	Verb, non-third pers. sing. pres.
8	JJR	Adjective, comparative	32.	VBZ	Verb, third pers. sing. present
9.	JJS	Adjective, superlative	33.	WDT	*wh*-determiner
10.	LS	List item marker	34.	WP	*wh*-pronoun
11.	MD	Modal	35.	WP$	Possessive *wh*-pronoun
12.	NN	Noun, singular or mass	36.	WRB	*wh*-adverb
13.	NNS	Noun, plural	37.	#	Pound sign
14.	NNP	Proper noun, singular	38.	$	Dollar sign
15.	NNPS	Proper noun, plural	39.	.	Sentence final punctuation
16.	PDT	Predeterminer	40.	,	Comma
17.	POS	Possessive ending	41.	:	Colon, semicolon
18.	PRP	Personal pronoun	42.	(	Left bracket character
19.	PRP$	Possessive pronoun	43.	)	Right bracket character
20.	RB	Adverb	44.	"	Straight double quote
21.	RBR	Adverb, comparative	45.	‘	Left open single quote
22.	RBS	Adverb, superlative	46.	“	Left open double quote
23.	RP	Particle	47.	’	Right close single quote
24.	SYM	Symbol	48.	”	Right close double quote

Battle-tested Japanese industrial managers here always buck up nervous newcomers with the tale of the first of their countrymen to visit Mexico, a boatload of samurai warriors blown ashore 375 years ago.

Unlike MULTEXT and the universal part-of-speech tagset, the Penn Treebank tagset concerns only English and shows little possibility of being adapted to another language. However, it is now widely established in the North American language processing community and in industry.

The Penn Treebank team proceeded in two steps to annotate their corpus. They first tagged the texts with an automatic stochastic tagger. They then reviewed and manually corrected the annotation. Table 7.13 follows the CoNLL 2008 and 2009 format (Surdeanu et al. 2008), which is slightly different from the CoNLL 2006 format presented in Sect. 6.6:

- The lemmas in the LEMMA and PLEMMA columns are generated automatically from a dictionary lookup with the WordNet lexical database. We will describe this database in Chap. 15.
- The POS column corresponds to the parts of speech manually assigned by the Penn Treebank team, while the PPOS tags – the predicted parts of speech – are generated automatically by a POS tagger (Giménez and Màrquez 2004). In the table, we can see that the tagger made only one mistake in the annotation of *buck*.

Table 7.13 Sample of annotated text from the Penn Treebank using the CoNLL 2008 format (After Marcus et al. (1993) and Surdeanu et al. (2008))

ID	FORM	LEMMA	PLEMMA	POS	PPOS	FEAT	PFEAT
1	Battle	battle	battle	NN	NN	_	_
2	-	-	-	HYPH	HYPH	_	_
3	tested	tested	tested	NN	NN	_	_
4	Japanese	japanese	japanese	JJ	JJ	_	_
5	industrial	industrial	industrial	JJ	JJ	_	_
6	managers	manager	manager	NNS	NNS	_	_
7	here	here	here	RB	RB	_	_
8	always	always	always	RB	RB	_	_
9	buck	buck	buck	VBP	VB	_	_
10	up	up	up	RP	RP	_	_
11	nervous	nervous	nervous	JJ	JJ	_	_
12	newcomers	newcomer	newcomer	NNS	NNS	_	_
13	with	with	with	IN	IN	_	_
14	the	the	the	DT	DT	_	_
15	tale	tale	tale	NN	NN	_	_
16	of	of	of	IN	IN	_	_
17	the	the	the	DT	DT	_	_
18	first	first	first	JJ	JJ	_	_
19	of	of	of	IN	IN	_	_
20	their	their	their	PRP$	PRP$	_	_
21	countrymen	countryman	countryman	NNS	NNS	_	_
22	to	to	to	TO	TO	_	_
23	visit	visit	visit	VB	VB	_	_
24	Mexico	mexico	mexico	NNP	NNP	_	_
25	,	,	,	,	,	_	_
26	a	a	a	DT	DT	_	_
27	boatload	boatload	boatload	NN	NN	_	_
28	of	of	of	IN	IN	_	_
29	samurai	samurai	samurai	NN	NN	_	_
30	warriors	warrior	warrior	NNS	NNS	_	_
31	blown	blow	blow	VBN	VBN	_	_
32	ashore	ashore	ashore	RB	RB	_	_
33	375	375	375	CD	CD	_	_
34	years	years	years	NNS	NNS	_	_
35	ago	ago	ago	RB	RB	_	_
36	.	.	.	.	.	_	_

- FEAT is the set of grammatical features as in Sect. 6.6, while PFEAT is automatically predicted. The Penn Treebank team did not annotate these features; they are replaced with an underscore in the table.

Fig. 7.1 Token annotation, where the identifier `id` corresponds to the word position

```
<tokens>
  <token id="1">Bilen</token>
  <token id="2">framför</token>
  <token id="3">justitieministern</token>
  <token id="4">svängde</token>
  <token id="5">fram</token>
  <token id="6">och</token>
  <token id="7">tillbaka</token>
  <token id="8">över</token>
  <token id="9">vägen</token>
  <token id="10">så</token>
  <token id="11">att</token>
  <token id="12">hon</token>
  <token id="13">blev</token>
  <token id="14">rädd</token>
  <token id="15">.</token>
</tokens>
```

7.5.3 An Annotation Scheme for Swedish

Current annotation schemes often use XML to encode data. This enables a stricter definition of codes through a DTD and makes it easier to use and share data. The annotation is often split into levels that reflect the processing stages. We describe here an example drawn from the Granska and CrossCheck projects to process Swedish (Carlberger et al. 2004) from the Kungliga Tekniska Högskolan in Stockholm. The annotation scheme uses the reference tagset for Swedish defined by the Stockholm–Umeå Corpus (Ejerhed et al. 1992).

The annotation has four levels, and we will describe two of them. The first one corresponds to tokenization. Figure 7.1 shows the token annotation of sentence:

Bilen framför justitieministern svängde fram och tillbaka över vägen så att hon blev rädd.
'The car in front of the Justice Minister swung back and forth and she was frightened.'

The second level contains the part-of-speech information, either with lemmas (Fig. 7.2) or without (Fig. 7.3). In both annotations, the tokens have been replaced by their positions. The `tag` attribute gives the part of speech and its features as a list separated by dots. The first item of the list the main category; for example, `nn` is a noun. The rest describes the features: `utr` is the utrum gender, `sin` is the singular number, `def` means definite, and `nom` is the nominative case.

7.6 Further Reading

Part-of-speech tagging has a long history in language processing, although many researchers in computational linguistics neglected it in the beginning. Early works include Harris (1962) and Klein and Simmons (1963). Harris' TDAP system was reconstructed and described by Joshi and Hopely (1999).

```
<taglemmas>
  <taglemma id="1" tag="nn.utr.sin.def.nom" lemma="bil"/>
  <taglemma id="2" tag="pp" lemma="framför"/>
  <taglemma id="3" tag="nn.utr.sin.def.nom" lemma=
    "justitieminister"/>
  <taglemma id="4" tag="vb.prt.akt" lemma="svänga"/>
  <taglemma id="5" tag="ab" lemma="fram"/>
  <taglemma id="6" tag="kn" lemma="och"/>
  <taglemma id="7" tag="ab" lemma="tillbaka"/>
  <taglemma id="8" tag="pp" lemma="över"/>
  <taglemma id="9" tag="nn.utr.sin.def.nom" lemma="väg"/>
  <taglemma id="10" tag="ab" lemma="så"/>
  <taglemma id="11" tag="sn" lemma="att"/>
  <taglemma id="12" tag="pn.utr.sin.def.sub" lemma="hon"/>
  <taglemma id="13" tag="vb.prt.akt.kop" lemma="bli"/>
  <taglemma id="14" tag="jj.pos.utr.sin.ind.nom" lemma="rädd"/>
  <taglemma id="15" tag="mad" lemma="."/>
</taglemmas>
```

Fig. 7.2 Tokens annotated with their part of speech and lemma. Tokens are indicated by their position. The tag specifies the part of speech and its features

```
<tags>
  <tag id="1" name="nn.utr.sin.def.nom"/>
  <tag id="2" name="pp"/>
  <tag id="3" name="nn.utr.sin.def.nom"/>
  <tag id="4" name="vb.prt.akt"/>
  <tag id="5" name="ab"/>
  <tag id="6" name="kn"/>
  <tag id="7" name="ab"/>
  <tag id="8" name="pp"/>
  <tag id="9" name="nn.utr.sin.def.nom"/>
  <tag id="10" name="ab"/>
  <tag id="11" name="sn"/>
  <tag id="12" name="pn.utr.sin.def.sub"/>
  <tag id="13" name="vb.prt.akt.kop"/>
  <tag id="14" name="jj.pos.utr.sin.ind.nom"/>
  <tag id="15" name="mad"/>
</tags>
```

Fig. 7.3 Tokens annotated with their part of speech only. Tokens are indicated by their position

Brill's tagging program marked a breakthrough in tagging with symbolic techniques. It is available from the Internet for English. Roche and Schabes (1995) proposed a dramatic optimization of it that proved ten times faster than and one third the size of stochastic methods. Constant (1991) and Vergne (1998, 1999) give examples of efficient symbolic taggers that use manually crafted rules.

Exercises

7.1. Complement Brill's tagging algorithm in Prolog with rules `alter(A, B, nexttag(C))` and `alter(A, B, surroundingtag(C, D))`.

7.2. Implement Brill's learning algorithm in Prolog or Perl with all the rule templates.

Chapter 8
Part-of-Speech Tagging Using Statistical Techniques

Like transformation-based tagging, statistical part-of-speech (POS) tagging assumes that each word is known and has a finite set of possible tags. These tags can be drawn from a dictionary or a morphological analysis. Statistical methods enable us to determine a sequence of part-of-speech tags $T = t_1, t_2, t_3, \ldots, t_n$, given a sequence of words $W = w_1, w_2, w_3, \ldots, w_n$. They use an annotated corpus to train a model and predict the correct tag when a word has more than one possible tag.

8.1 Part-of-Speech Tagging with Linear Classifiers

Linear classifiers, such as logistic regression, perceptrons, or support vector machines, which we saw in Sect. 4.4, are an efficient set of numerical techniques we can use to carry out part-of-speech tagging. As input, the tagger reads the sentence's words sequentially from left to right and, using a model it has trained beforehand, predicts the part of speech of the current word.

To train and apply the model, the tagger extracts a set of features from the surrounding words, typically a sliding window spanning five words and centered on the current word. Core features are the lexical values of the words inside this window, called the context, as well as the parts of speech to the left of the current word:

1. The lexical values are the input data to the tagger. They are produced by a tokenizer, possibly followed by a morphological parser.
2. The parts of speech are assigned from left to right by the tagger. They are reused by the tagger to predict the POS of the current word. The part-of-speech features are often called dynamic because they are created at run-time.

We then associate the feature vector $(w_{i-2}, w_{i-1}, w_i, w_{i+1}, w_{i+2}, t_{i-2}, t_{i-1})$ with the part-of-speech tag t_i at index i. Using the sentence in Table 7.13, Fig. 8.1 shows an example of it centered on the word *visit*.

P.M. Nugues, *Language Processing with Perl and Prolog*, Cognitive Technologies,
DOI 10.1007/978-3-642-41464-0_8, © Springer-Verlag Berlin Heidelberg 2014

ID	FORM	PPOS	
	BOS	BOS	*Padding*
	BOS	BOS	
1	Battle	NN	
2	-	HYPH	
3	tested	NN	
...	...	...	
17	the	DT	
18	first	JJ	
19	of	IN	
20	their	PRP$	
21	countrymen	NNS	*Input features*
22	to	TO	
23	**visit**	VB	*Predicted tag*
24	Mexico		↓
25	,		
26	a		
27	boatload		
...	...	...	
34	years		
35	ago		
36	.		
	EOS		*Padding*
	EOS		

Fig. 8.1 Extracting features from the Penn Treebank (Marcus et al. 1993) to predict a part of speech. The features are extracted from a window of five words surrounding the word *visit*: the five lexical values and the two preceding POS tags

- The part of speech to predict is $t_{23} = \text{VB}$;
- The surrounding words are $w_{21} = \textit{countrymen}$, $w_{22} = \textit{to}$, $w_{23} = \textit{visit}$, $w_{24} = \textit{Mexico}$, and $w_{25} = ,$;
- The preceding parts of speech are $t_{21} = \text{NNS}$ and $t_{22} = \text{TO}$.

Table 8.1 shows more feature vectors from this sentence. They are used first to train a model. This model is then applied sequentially to assign the tags. If we use logistic regression, the tagger outputs a probability, $P(t_i | w_{i-2}, w_{i-1}, w_i, w_{i+1}, w_{i+2}, t_{i-2}, t_{i-1})$, that we can associate with each tag of the sequence.

At the beginning and end of the sentence, the window will extend beyond the sentence boundaries. A practical way to get rid of this is to pad the sentence – the words and parts of speech – with dummy symbols such as BOS (beginning of sentence) and EOS (end of sentence) or `<s>` and `</s>`. If the window has a size of five, we will pad the sentence with two BOS symbols in the beginning and two EOS symbols in the end.

We extract the features from POS-annotated corpora, and we can train the models using machine-learning libraries such as LIBLINEAR (Fan et al. 2008) for logistic regression or LIBSVM (Chang and Lin 2011) for support vector machines. Real systems would use more features than those from the core feature set such as the word prefixes and suffixes, part-of-speech bigrams, word bigrams, etc. It is thus easy to extend the set we presented.

Linear classifiers are efficient tools to implement a POS tagger. Giménez and Màrquez (2004) describe a tagger using support vector machines as well as a

Table 8.1 The feature vectors and the parts of speech to predict

	Feature vectors							
ID	w_{i-2}	w_{i-1}	w_i	w_{i+1}	w_{i+2}	t_{i-2}	t_{i-1}	**PPOS**
1	BOS	BOS	Battle	–	tested	BOS	BOS	NN
2	BOS	Battle	–	tested	Japanese	BOS	NN	HYPH
3	Battle	–	tested	Japanese	industrial	NN	HYPH	JJ
…	…	…	…	…	…	…	…	…
19	the	first	of	their	countrymen	DT	JJ	IN
20	first	of	their	countrymen	to	JJ	IN	PRP$
21	of	their	countrymen	to	visit	IN	PRP$	NNS
22	their	countrymen	to	visit	Mexico	PRP$	NNS	TO
23	countrymen	to	visit	Mexico	,	NNS	TO	VB
24	to	visit	Mexico	,	a	TO	VB	NNP
25	visit	Mexico	,	a	boatload	VB	NNP	,
…	…	…	…	…	…	…	…	…
34	ashore	375	years	ago	.	RB	CD	NNS
35	375	years	ago	.	EOS	CD	NNS	RB
36	years	ago	.	EOS	EOS	NNS	RB	.

complete feature set. Ratnaparkhi (1996) is an early example with logistic regression. In addition to the local classification introduced in this section, Ratnaparkhi optimized the complete part-of-speech sequence. He multiplied the output probabilities from each tagging operation and searched the tag sequence so that the product:

$$\hat{T} = \underset{t_1,t_2,t_3,\ldots,t_n}{\arg\max} \prod_{i=1}^{n} P(t_i | w_{i-2}, w_{i-1}, w_i, w_{i+1}, w_{i+2}, t_{i-2}, t_{i-1}).$$

reaches a maximum. Ratnaparkhi applied a beam search to find this optimal sequence. The Viterbi algorithm is an alternative solution to this problem that we will study in Sect. 8.3.6.

8.2 The Noisy Channel Model

8.2.1 Presentation

The noisy channel is a second technique to find an optimal part-of-speech sequence. It refers to Shannon's (1948) model, where a sequence of symbols is transmitted over a noisy channel and received in the form of a sequence of signals. Here, we suppose that part-of-speech tags are transmitted and come out in the form of words:

$$t_1, t_2, t_3, \ldots, t_n \rightarrow \text{noisy channel} \rightarrow w_1, w_2, w_3, \ldots, w_n.$$

The optimal part-of-speech sequence knowing the word sequence corresponds to the maximization of the conditional probability:

$$\hat{T} = \underset{t_1,t_2,t_3,\ldots,t_n}{\arg\max}\ P(t_1, t_2, t_3, \ldots, t_n | w_1, w_2, w_3, \ldots, w_n),$$

where we use the function $\underset{x}{\arg\max}\ f(x)$ to denote the value of x for which $f(x)$ reaches its maximum value.

Bayes' theorem on conditional probabilities of events A and B states that:

$$P(A|B)P(B) = P(B|A)P(A).$$

We denote $P(W) = P(w_1, w_2, w_3, \ldots, w_n)$ and $P(T) = P(t_1, t_2, t_3, \ldots, t_n)$. Using Bayes' theorem, the most probable estimate of the part-of-speech sequence is given by:

$$\hat{T} = \underset{T}{\arg\max}\ \frac{P(T)P(W|T)}{P(W)}.$$

For a given word sequence, $w_1, w_2, w_3, \ldots, w_n$, $P(W)$ is constant and we can leave it out. We can rewrite the formula as:

$$\hat{T} = \underset{T}{\arg\max}\ P(T)P(W|T).$$

Such a model is called a *generative model*. It means that to find the most likely part-of-speech sequence, we need to generate all the possible sequences and search the one with the maximal probability.

However, using this brute-force technique yields an astronomic number of sequences. In most cases, it is intractable unless we reduce the sequences to n-gram approximations and implement an efficient search called the Viterbi algorithm.

8.2.2 The N-Gram Approximation

Statistics on sequences of any length are impossible to obtain, and at this point we need to make some approximations on $P(T)$ and $P(W|T)$ to make the estimation tractable. A product of trigrams usually approximates the complete part-of-speech sequence:

$$P(T) = P(t_1, t_2, t_3, \ldots, t_n) \approx P(t_1)P(t_2|t_1)\prod_{i=3}^{n} P(t_i|t_{i-2}, t_{i-1}).$$

If we use a start-of-sentence delimiter <s>, the two first terms of the product, $P(t_1)P(t_2|t_1)$, are rewritten as $P(<s>)P(t_1|<s>)P(t_2|<s>, t_1)$, where $P(<s>) = 1$.

We estimate the probabilities with the maximum likelihood, P_{MLE}:

$$P_{\text{MLE}}(t_i|t_{i-2}, t_{i-1}) = \frac{C(t_{i-2}, t_{i-1}, t_i)}{C(t_{i-2}, t_{i-1})}.$$

Probabilities on trigrams $P(t_i|t_{i-2}, t_{i-1})$ require an estimate for any sequence of three parts-of-speech tags. This is obtained from hand-annotated corpora. If N_p is the number of the different parts-of-speech tags, there are $N_p \times N_p \times N_p$ values to estimate. Most of the time, annotated data is not sufficient and some sequences are missing. Few corpora are likely to contain a reliable number of the article–article–article sequence, for instance. We already encountered this problem of sparse data in Chap. 5. We can solve it using a back-off strategy or a linear interpolation.

If data are missing, we can back-off to bigrams:

$$P(T) = P(t_1, t_2, t_3, \ldots, t_n) \approx P(t_1) \prod_{i=2}^{n} P(t_i|t_{i-1}).$$

We can further approximate the part-of-speech sequence as the product of part-of-speech probabilities:

$$P(T) = P(t_1, t_2, t_3, \ldots, t_n) \approx \prod_{i=1}^{n} P(t_i).$$

And finally, we can combine linearly these approximations:

$$P_{\text{LinearInter}}(t_i|t_{i-2}, t_{i-1}) = \lambda_1 P(t_i|t_{i-2}, t_{i-1}) + \lambda_2 P(t_i|t_{i-1}) + \lambda_3 P(t_i),$$

with $\lambda_1 + \lambda_2 + \lambda_3 = 1$, for example, $\lambda_1 = 0.6$, $\lambda_2 = 0.3$, $\lambda_3 = 0.1$.

Using the maximum likelihood estimate, this yields:

$$P_{\text{LinearInter}}(t_i|t_{i-2}, t_{i-1}) = \lambda_1 \frac{C(t_{i-2}, t_{i-1}, t_i)}{C(t_{i-2}, t_{i-1})} + \lambda_2 \frac{C(t_{i-1}, t_i)}{C(t_{i-1})} + \lambda_3 \frac{C(t_i)}{N},$$

where N is the count of words in the corpus.

We can obtain optimal λ values by using a development set: a part of the hand-annotated corpus distinct from the training set and the test set dedicated to the fine-tuning of parameters. After learning the probabilities from the training set, we will run the part-of-speech tagger on the development set. We will vary the λ values until we find the triple that yields the best accuracy. We will finally apply the POS tagger to the test set to know its real accuracy.

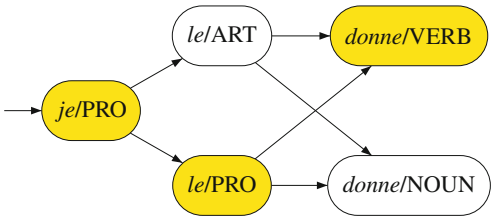


Fig. 8.2 Possible sequences of part-of-speech tags, where PRO denotes a pronoun, ART, an article, NOUN, a noun, and VERB, a verb

The complete word sequence knowing the part-of-speech sequence is usually approximated as:

$$P(W|T) = P(w_1, w_2, w_3, \ldots, w_n|t_1, t_2, t_3, \ldots, t_n) \approx \prod_{i=1}^{n} P(w_i|t_i).$$

Like the previous probabilities, $P(w_i|t_i)$ is estimated from hand-annotated corpora using the maximum likelihood:

$$P_{\text{MLE}}(w_i|t_i) = \frac{C(w_i, t_i)}{C(t_i)}.$$

For N_w different words, there are $N_p \times N_w$ values to obtain. Many of the estimates will be 0, however. This can reflect the true parts of speech of a word; nonetheless, it is very likely that many words will not appear with all their parts of speech in the training corpus. To get more accurate estimates, we can smooth them with a dictionary (Church 1988). We then extract of all the possible parts of speech of the words in the corpus and use Laplace's rule to smooth the values (Sect. 5.7.2).

8.2.3 Tagging a Sentence

We will now give an example of sentence tagging in French with *Je le donne* 'I give it'. Word *Je* is an unambiguous pronoun. Word *le* is either an article or a pronoun, and *donne* can be a noun (*deal*) or a verb (*donner*). Probabilistic tagging consists in finding the optimal path from the four possible in Fig. 8.2.

Using the formulas given before, we associate each transition with a probability product: $P(w_i|t_i) \times P(t_i|t_{i-2}, t_{i-1})$. We compute the estimate of part-of-speech sequences along the four paths by multiplying the probabilities. The optimal tagging corresponds to the maximum of these four values:

1. $P(\text{PRO}|<\text{s}>) \times P(\text{ART}|<\text{s}>, \text{PRO}) \times P(\text{VERB}|\text{PRO}, \text{ART}) \times P(je|\text{PRO}) \times P(le|\text{ART}) \times P(donne|\text{VERB})$
2. $P(\text{PRO}|<\text{s}>) \times P(\text{ART}|<\text{s}>, \text{PRO}) \times P(\text{NOUN}|\text{PRO}, \text{ART}) \times P(je|\text{PRO}) \times P(le|\text{ART}) \times P(donne|\text{NOUN})$

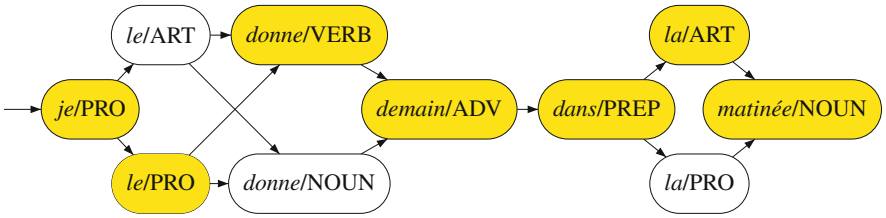


Fig. 8.3 The search space, where ADV denotes an adverb and PREP, a preposition; the other tags are as given in Fig. 8.2

3. $P(\text{PRO}|\texttt{<s>}) \times P(\text{PRO}|\texttt{<s>}, \text{PRO}) \times P(\text{VERB}|\text{PRO}, \text{PRO}) \times P(je|\text{PRO}) \times P(le|\text{PRO}) \times P(donne|\text{VERB})$
4. $P(\text{PRO}|\texttt{<s>}) \times P(\text{PRO}|\texttt{<s>}, \text{PRO}) \times P(\text{NOUN}|\text{PRO}, \text{PRO}) \times P(je|\text{PRO}) \times P(le|\text{PRO}) \times P(donne|\text{NOUN})$

This method is very simple. However, it is very costly for long sequences. The computation with a sentence of n words and a tagset of N_p tags will have an upper bound complexity of $(N_p)^n$, which means it is exponential.

8.2.4 The Viterbi Algorithm: An Intuitive Presentation

Using the noisy channel model as we described it is not efficient in terms of speed and memory. This is because the algorithm has to maintain nonoptimal paths for all the intermediate nodes in the automaton. The Viterbi algorithm is a common way to optimize the search.

In the naïve implementation, we traversed all the paths and we computed the most probable POS sequence at the final node of the automaton, i.e., at the final word of the sentence. The Viterbi algorithm (1967) determines the optimal subpaths for each node in the automaton while it traverses the automaton and discards the others. We shall extend the example of the previous section to

Je le donne demain dans la matinée.
'I give it tomorrow morning.'

and let us consider bigrams instead of trigrams to simplify the presentation.

Figure 8.3 shows the possible POS tags and the number of possible paths, which is $1 \times 2 \times 2 \times 1 \times 1 \times 2 \times 1 = 8$. Let us traverse the automaton from *Je* to *dans*.

The words *demain* and *dans* are not ambiguous, and we saw in the last section that there are four possible paths at this point. Up to *demain*, the most likely sequence will correspond to the most probable path out of the four we saw before:

1. $P(\text{PRO}|\texttt{<s>}) \times P(\text{ART}|\text{PRO}) \times P(\text{VERB}|\text{ART}) \times P(\text{ADV}|\text{VERB}) P(je|\text{PRO}) \times P(le|\text{ART}) \times P(donne|\text{VERB}) \times P(demain|\text{ADV})$
2. $P(\text{PRO}|\texttt{<s>}) \times P(\text{ART}|\text{PRO}) \times P(\text{NOUN}|\text{ART}) \times P(\text{ADV}|\text{NOUN}) P(je|\text{PRO}) \times P(le|\text{ART}) \times P(donne|\text{NOUN}) \times P(demain|\text{ADV})$

3. $P(\text{PRO}|<\text{s}>) \times P(\text{PRO}|\text{PRO}) \times P(\text{VERB}|\text{PRO}) \times P(\text{ADV}|\text{VERB}) P(je|\text{PRO}) \times P(le|\text{PRO}) \times P(donne|\text{VERB}) \times P(demain|\text{ADV})$
4. $P(\text{PRO}|<\text{s}>) \times P(\text{PRO}|\text{PRO}) \times P(\text{NOUN}|\text{PRO}) \times P(\text{ADV}|\text{NOUN}) P(je|\text{PRO}) \times P(le|\text{PRO}) \times P(donne|\text{NOUN}) \times P(demain|\text{ADV})$

Demain has still the memory of the ambiguity of *donne*: $P(\text{ADV}|\text{VERB})$ and $P(\text{ADV}|\text{NOUN})$. This is no longer the case with *dans*. According to the noisy channel model and the bigram assumption, the term brought by the word *dans* is $P(dans|\text{PREP}) \times P(\text{PREP}|\text{ADV})$. It does not show the ambiguity of *le* and *donne*. The subsequent terms will ignore it as well.

This means that the optimal POS tag sequence of words before *dans* is already determined even if we have not yet reached the end of the sentence. It corresponds to the highest value of the four paths. It is then sufficient to keep it with the corresponding path. We can forget the others. This is the idea of the Viterbi optimization. We will describe the algorithm rigorously in the next section.

8.3 Markov Models

When we tagged words with a stochastic technique, we assumed that the current word's part of speech depended only on a couple of words before it. This limited history is a frequent property of many linguistic phenomena. It has been studied extensively since the end of the nineteenth century, starting with Andrei Markov. Markov processes form the theoretical background to stochastic tagging and can be applied to many problems. We introduce them now.

8.3.1 Markov Chains

A Markov chain or process is a sequence $\{X_1, X_2, \ldots, X_n\}$, where X_t denotes a random variable at time t. Variables have their values in a finite set of states $\{q_1, \ldots, q_{N_p}\}$, called the state space. Following Rabiner (1989), processes are Markovian if they have the following properties:

- A limited history. The current state depends only on a constant number of previous states: one in first-order processes, $P(X_t = q_j|X_1, \ldots, X_{t-1}) = P(X_t = q_j|X_{t-1})$, and two in second-order processes $P(X_t = q_j|X_1, \ldots, X_{t-1}) = P(X_t = q_j|X_{t-2}, X_{t-1})$.
- Independent of time t. For first-order processes, this means that they can be represented as a transition matrix with coefficients $P(X_t = q_j|X_{t-1} = q_i) = a_{ij}$, $1 \leq i, j \leq N_p$, with ordinary probability constraints $\sum_{j=1}^{N_p} a_{ij} = 1$, and $a_{ij} \geq 0$.

Markov chains define random transitions from one state to another one. We can represent them as **probabilistic** or **weighted automata**. We just need to augment transitions of automata we used in Chap. 2 with a probability. Unlike ordinary

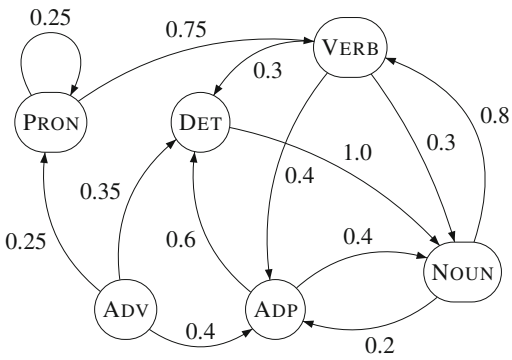


Fig. 8.4 A Markov chain representing bigram probabilities as part-of-speech transitions (numbers are fictitious and transitions are not complete). It uses the universal POS tagset (Petrov et al. 2012) shown in Table 7.9

automata, the initial state can be any state in the set and will be modeled by a probability at time 1. The probability of initial states is $\pi_i = P(X_1 = q_i)$, with $\sum_{i=1}^{N_p} \pi_i = 1$.

In the case of natural language processing, "time sequence" is not the most relevant term to describe the chain. More appropriately, the sequence corresponds to the word flow from left to right and t to the word position in the sequence. It is easy then to see that first-order processes reflect part-of-speech bigrams, while second-order processes correspond to trigrams. Figure 8.4 shows partial bigram probabilities using a Markov chain (numbers are fictitious and transitions are not complete). For part-of-speech tagging, a_{ij} coefficients correspond to probabilities of part-of-speech bigrams computed over the tagset.

8.3.2 Trellis Representation

Instead of using an automaton, we can represent a Markov process as a trellis, where states are a function of the time (the word's indexes, here). In part-of-speech tagging, the vertical axis corresponds to the different part-of-speech values (the states) and the horizontal axis corresponds to the part-of-speech sequence (Fig. 8.5). All the possible bigram combinations are represented as arrows from states at time $t-1$ to states at time t and n is the sentence length.

A trellis is a compact graphical representation of all the possible paths of length n with all the possible POS sequences. The bold lines on Fig. 8.5 is one of these paths that corresponds to the sequence: DET, ADV, ADJ, NOUN.

8.3.3 Hidden Markov Models

Markov chains provide a model to the part-of-speech sequence. However, this sequence is not directly accessible since we usually only have the word sequence.

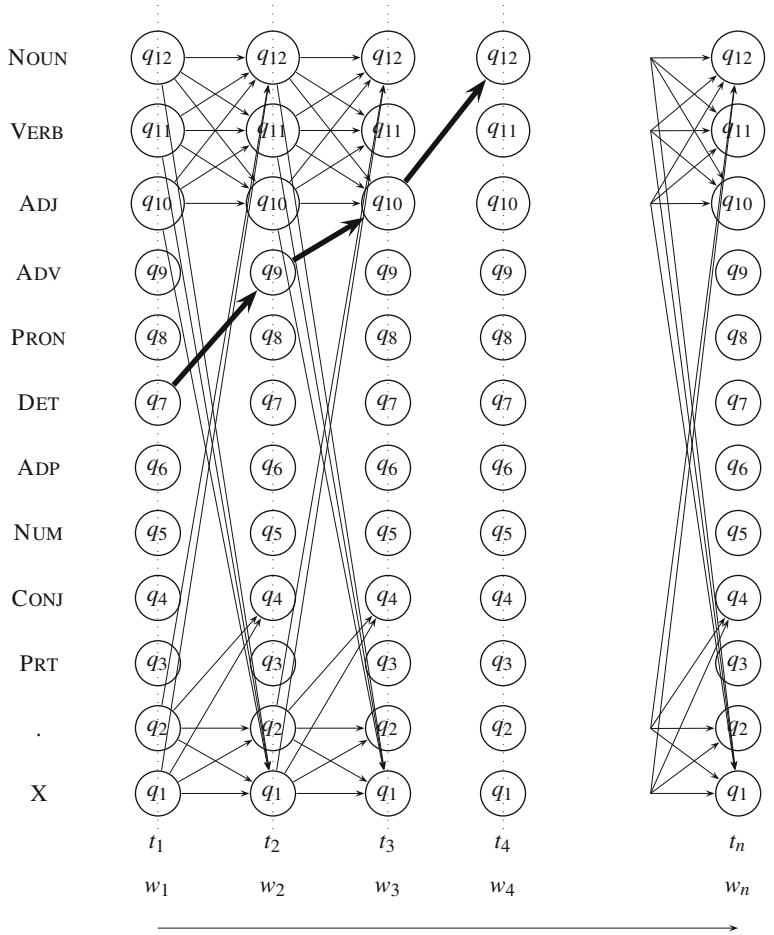


Fig. 8.5 A trellis that represents the states as the *vertical axis* and the time as the *horizontal axis*. The states $q_1, q_2, \ldots q_{12}$ represent the possible part-of-speech values, here from the universal POS tagset (Petrov et al. 2012). The discrete time values are the indices in the part-of-speech sequence: $t_1, t_2, t_3, \ldots, t_n$ corresponding to the word sequence $w_1, w_2, w_3, \ldots, w_n$, where n is the sentence length. The *bold lines* correspond to the part-of-speech sequence: DET, ADV, ADJ, NOUN

Hidden Markov models (HMM) are an extension to the Markov chains that make it possible to include the words in the form of observed symbols. Each state of an HMM emits a symbol taken from an output set along with an emission probability. HMMs are then a stochastic representation of an observable output generated by a hidden sequence of states. They enable us to compute the probability of a state sequence (the parts of speech) given an output or observation sequence (the words).

We saw that part-of-speech tagging uses a stochastic formula that comprises two terms: $P(T)$ and $P(W|T)$. The first one, $P(T)$, corresponds to a Markov chain where transition probabilities between states represent the part-of-speech bigrams.

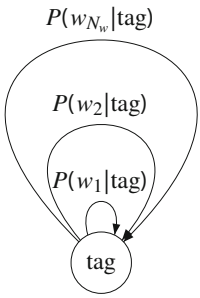


Fig. 8.6 Each state in the trellis is augmented with word emission probabilities, here for a given tag, where N_w is the total number of different words in the vocabulary

Table 8.2 The hidden Markov model notation and its application to part-of-speech tagging

HMM notation	Application to POS tagging
$S = \{q_1, q_2, q_3, \ldots, q_{N_p}\}$ is a finite set of states.	The set of parts of speech.
$V = \{v_1, v_2, v_3, \ldots, v_{N_w}\}$ is an output alphabet: a finite set of symbols.	The set of words, the vocabulary.
$O = \{o_1, o_2, o_3, \ldots, o_n\}$ is the output or observation sequence, with $o_i \in V$ obtained from a sequence of states.	Each part of speech emits one word taken in the vocabulary. This is what we observe.
$A = \{a_{ij}\}$ is a state transition matrix.	The bigram probabilities $P(t_k = q_j \mid t_{k-1} = q_i)$.
$B = \{b_j(v_k)\}$ are the emission probabilities of symbol v_k in state j.	The conditional probability to observe a word given a part of speech $P(w_k \mid t_j)$.
$\Pi = \{\pi_i\}$ are the initial state probabilities.	The probability of the first part of speech.

The second term, $P(W|T)$, is an HMM superimposed on the chain. It augments each state with the capacity to emit a word using a probability function $P(w_i|t_i)$ that measures the association between the parts of speech and the words (Fig. 8.6). Although a state – a part of speech – can emit any word in the model, most probabilities will be 0 in reality. This is because words have a finite number of possible parts of speech, most of the time, as we saw, only one or two.

The formal definition of HMMs is based on the Markov chains where we add the emission properties. Table 8.2 shows the notation and its application in part-of-speech tagging.

8.3.4 Three Fundamental Algorithms to Solve Problems with HMMs

Hidden Markov models are able to represent associations between word and parts-of-speech sequences. However, they do not tell how to solve the annotation problem. We need complementary algorithms for them to be useful. More generally, problems

to solve fall into three categories that correspond to three fundamental algorithms (Rabiner 1989):

- Estimate the probability of an observed sequence. This corresponds to the sum of all the paths producing the observation. It is solved using the forward procedure. In the specific case of POS tagging, it will determine the probability of the word sequence. Although the forward procedure is not of primary importance here, it is fundamental and has many other applications.
- Determine the most likely path of an observed sequence. This is a decoding problem that is solved using the Viterbi algorithm.
- Determine (learn) the parameters given a set of observations. This algorithm is used to build models when we do not know the parameters. It is solved using the forward–backward algorithm.

We now present the algorithms where we follow Rabiner (1989).

8.3.5 The Forward Procedure

The first problem to solve is to compute the probability of an observation sequence $O = \{o_1, o_2, o_3, \ldots, o_n\}$, given a HMM model $\lambda = (A, B, \pi)$.

Let us start with only one sequence of states $Q = \{s_1, s_2, s_3, \ldots, s_n\}$, with $s_i \in S$. The observation probability is the probability of the state sequence we consider:

$$\begin{aligned} P(Q|\lambda) &= \pi_{s_1} \prod_{t=2}^{n} P(s_t|s_{t-1}), \\ &= \pi_{s_1} a_{s_1 s_2} a_{s_2 s_3} \ldots a_{s_{n-1} s_n}, \end{aligned}$$

multiplied by the product of each observation probability given the state it is emitted from in the sequence:

$$\begin{aligned} P(O|Q,\lambda) &= \prod_{t=1}^{n} P(o_t|s_t,\lambda), \\ &= b_{s_1}(o_1) b_{s_2}(o_2) b_{s_3}(o_3) \ldots b_{s_n}(o_n). \end{aligned}$$

In HMMs, any sequence of states can produce the observation. This means that the observation probability is the sum of observation probabilities for all the possible state combinations:

$$\begin{aligned} P(O|\lambda) &= \sum_{\text{All} Q} P(O|Q,\lambda) P(Q|\lambda), \\ &= \sum_{\text{All} s_1, s_2, \ldots, s_n} \pi_{s_1} b_{s_1}(o_1) a_{s_1 s_2} b_{s_2}(o_2) a_{s_2 s_3} b_{s_3}(o_3) \ldots a_{s_{n-1} s_n} b(o_n). \end{aligned}$$

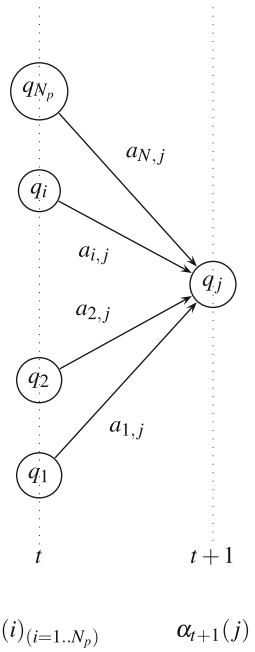


Fig. 8.7 Transitions from states $q_1, q_2, q_3, \ldots, q_{N_p}$ at time t to state q_j at time $t+1$. We have $\alpha_{t+1}(j) = b_j(o_{t+1}) \times \sum_{i=1}^{N_p} \alpha_t(i) a_{ij}$

This method, however, is intractable for long sequences because of its complexity, $(N_p)^n$.

The forward procedure simplifies the brute-force method by factoring all paths incoming into a state at time t. This means that at each instant of time of the observation sequence, we maintain exactly N_p paths: the number of different states.

Let us denote $\alpha_t(j)$ the probability of an observation $o_1, o_2, o_3, \ldots, o_t$, with the condition that we are in state q_j at time t: $P(o_1, o_2, o_3, \ldots, o_t, s_t = q_j|\lambda)$. We compute $\alpha_{t+1}(j)$ by induction with transitions from all states at time t to state i at time $t+1$. Figure 8.7 shows how $\alpha_t(i)$ values with i ranging from 1 to N_p are summed to obtain $\alpha_{t+1}(j)$.

We can compute an observation probability with a matrix reproducing the structure of the trellis in Fig. 8.5. The algorithm iteratively fills the trellis columns from left to right. Each column is an array of length N_p corresponding to the number of states where we store the probabilities of the observation so far. The element of index i in the tth column contains the $\alpha(i)$ value at time t.

The first step of the algorithm fills the first column with the initial probabilities. The induction loop updates the values from t to $t+1$ by summing all the incoming transitions for each element in the $(t+1)$th column from the tth column (Table 8.3). Finally, we obtain the observation probability by summing all the elements of the last column in the matrix. The complexity of this algorithm is $O((N_p)^2 \cdot n)$.

Table 8.3 The forward procedure: N_p is the number of states and n is the length of the sequence

Steps	Operations
1. Initialization	$\alpha_1(i) = \pi_i b_i(o_1), 1 \leq i \leq N_p$
2. Induction	$\alpha_{t+1}(j) = b_j(o_{t+1}) \times \sum_{i=1}^{N_p} \alpha_t(i) a_{ij}, 1 \leq j \leq N_p$, and $1 \leq t \leq n-1$
3. Termination	$P(O\|\lambda) = \sum_{i=1}^{N_p} \alpha_n(i)$

Table 8.4 The Viterbi algorithm: N_p is the number of states and n is the length of the sequence

Steps	Operations
1. Initialization	$\delta_1(i) = \pi_i b_i(o_1), 1 \leq i \leq N_p$
	$\psi_1(i) = null$
2. Induction	$\delta_{t+1}(j) = b_j(o_{t+1}) \times \max_{1 \leq i \leq N_p} \delta_t(i) a_{ij}, 1 \leq j \leq N_p$, and $1 \leq t \leq n-1$
	$\psi_{t+1}(j) = \arg \max_{1 \leq i \leq N_p} \delta_t(i) a_{ij}$
3. Termination	$P* = \max_{1 \leq i \leq N_p} \delta_n(i)$
	$s_n* = \arg \max_{1 \leq i \leq N_p} \delta_n(i)$
	The optimal path sequence is given by the backtracking: $s_n^*, s_{n-1}^* = \psi_n(s_n^*), s_{n-2}^* = \psi_{n-2}(s_{n-1}^*), \ldots$

8.3.6 Viterbi Algorithm

The Viterbi algorithm is an efficient method to find the optimal sequence of states given an observation. As with the forward procedure, it iterates from $t = 1$ to $t = n$ and searches the optimal path leading to each state in the trellis at time t.

Let us denote $\delta_t(j)$ the maximal probability of an observation $o_1, o_2, o_3, \ldots, o_t$ with the condition that we are in state q_j at time t:

$$\max_{s_1, s_2, \ldots, s_{t-1}} P(s_1, s_2, \ldots, s_{t-1}, o_1, o_2, o_3, \ldots, o_t, s_t = q_j | \lambda),$$

and $\psi_t(j)$ the corresponding optimal path.

The Viterbi algorithm resembles the forward procedure. It moves from left to right iteratively to fill the columns in the trellis. Each column element contains the most probable path, $\psi(j)$, to reach this element and its probability $\delta(j)$. In fact, $\psi(j)$ just needs to store the preceding state in the optimal path.

The first step of the algorithm fills the first column with the initial probabilities. The induction loop updates the values from t to $t + 1$ by taking the maximum of all the incoming transitions for each element in the $(t + 1)$th column and the node that led to it. Finally, we determine the most probable path from the maximum of all the elements of the last column in the matrix. We backtrack in the matrix to find the state sequence that led to it, for instance, using back pointers (Table 8.4).

$i \backslash \delta$	δ_1	δ_2	δ_3	δ_4	δ_5	δ_6	δ_7	δ_8
PREP	0							
ADV	0							
PRO	0							
VERB	0							
NOUN	0							
ART	0							
<s>	1.0	0	0	0	0	0	0	0
	<s>	*Je*	*le*	*donne*	*demain*	*dans*	*la*	*matinée*

Fig. 8.8 The Viterbi algorithm applied to the sentence <s> *Je le donne demain dans la matinée*

The Viterbi algorithm is a dynamic programming technique comparable to the computation of the min-edit distance. Its implementation also uses a table. Figure 8.8 shows how to fill the three first columns with the sentence <s> *Je le donne demain dans la matinée*.

We start the sentence with $\delta_1(<s>) = 1.0$ and $\delta_1(i) = 0$ for the rest of the indices $i \neq <s>$. This means that in the first column, all the cells equal 0, except for one. The computation of the second column is easy. Each cell i is filled with the term $P(i|<s>) \times P(Je|i)$, with $i \in \{\text{PREP}, \text{ADV}, \text{PRO}, \text{VERB}, \text{NOUN}, \text{ART}, <s>\}$. The algorithm really starts with the third column. For each cell j, we compute

$$\max_i \; P(j|i) \times P(le|j) \times \delta_2(i).$$

The pronoun cell, for instance, is filled with

$$\max_i \; P(\text{PRO}|i) \times P(le|\text{PRO}) \times \delta_2(i).$$

This process is iterated for each column to the end of the matrix.

8.3.7 The Backward Procedure

We have computed the estimation of an observation from left to right. Although less natural, we can also compute it from right to left. We now present this backward procedure to introduce the forward–backward algorithm in the next section.

The backward variable $\beta_t(j) = P(o_{t+1}, o_{t+2}, o_{t+3}, \ldots, o_n|s_t = q_j, \lambda)$ is the probability of an observation $o_{t+1}, o_{t+2}, o_{t+3}, \ldots, o_n$ with the condition that we are in state q_j at time t. We compute $\beta_t(i)$ by induction with transitions from state i at time t to all states at time $t+1$. Figure 8.9 shows how $\beta_{t+1}(i)$ values are summed to obtain β_t, and Table 8.5 shows the procedure.

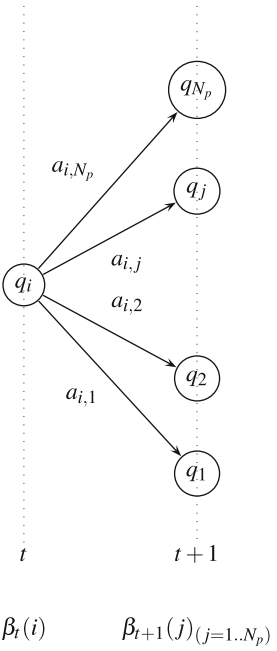


Fig. 8.9 Transitions from state q_i at time t to states $q_1, q_2, q_3, \ldots, q_{N_p}$ at time $t+1$. We have $\beta_t(i) = \sum_{j=1}^{N_p} a_{ij} b_j(o_{t+1}) \beta_{t+1}(i)$

Table 8.5 The backward procedure: N_p is the number of states and n is the length of the sequence

Steps	Operations
1. Initialization	$\beta_n(i) = 1, 1 \leq i \leq N_p$
2. Induction	$\beta_t(i) = \sum_{j=1}^{N_p} a_{ij} b_j(o_{t+1}) \beta_{t+1}(i), 1 \leq j \leq N_p$, and for $t = n-1$ to $t = 1$.
3. Termination	$P(O\|\lambda) = \sum_{i=1}^{N_p} \pi_i b_i(o_1) \beta_1(i)$

8.3.8 The Forward–Backward Algorithm

The forward–backward algorithm will enable us to derive the a_{ij} and $b_j(o_t)$ coefficients, here $P(w_i|t_i)$ and $P(t_i|t_{i-1})$, from raw, unannotated texts. Although this yields results inferior to those obtained from a hand-annotated corpus, it makes it possible to build a part-of-speech tagger when no annotation is available.

The forward–backward algorithm is referred to as an **unsupervised learning** method, because no additional information is available except the text. This is opposed to **supervised learning**, when the algorithm has access to some sort of reference annotation.

Table 8.6 Iterative estimation of $P(t_i|t_{i-1})$ (figures are fictitious)

Steps	Estimates used to tag the corpus	Estimates derived from the tagged corpus
Initial estimates	$P(\text{PRO}\|\text{PRO}) = 0.2$ $P(\text{ART}\|\text{PRO}) = 0.2$ $P(\text{VERB}\|\text{PRO}) = 0.6$	
We tag the corpus and we derive new estimates		$P(\text{PRO}\|\text{PRO}) = 0.15$ $P(\text{ART}\|\text{PRO}) = 0.05$ $P(\text{VERB}\|\text{PRO}) = 0.8$
Second estimates	$P(\text{PRO}\|\text{PRO}) = 0.15$ $P(\text{ART}\|\text{PRO}) = 0.05$ $P(\text{VERB}\|\text{PRO}) = 0.8$	
We retag the corpus and we derive estimates		$P(\text{PRO}\|\text{PRO}) = 0.18$ $P(\text{ART}\|\text{PRO}) = 0.02$ $P(\text{VERB}\|\text{PRO}) = 0.9$
Third estimates	$P(\text{PRO}\|\text{PRO}) = 0.18$ $P(\text{ART}\|\text{PRO}) = 0.02$ $P(\text{VERB}\|\text{PRO}) = 0.9$	

Informal Presentation

The idea of the forward–backward algorithm is to guess initial estimates to $P(t_i|t_{i-1})$ and $P(w_i|t_i)$ and tag the corpus. Once we have a tagged corpus, we can derive new estimates of $P(w_i|t_i)$ and $P(t_i|t_{i-1})$ that we will use to retag the corpus. We repeat the process until it converges (Table 8.6).

However, we have no guarantee that the algorithm converges, and when it converges, we can also hit a local maximum. In the latter case, the learning procedure will stop without finding correct figures. This is the drawback of this method. For this reason, some quantity of hand-annotated data is always preferable to a raw corpus (Merialdo 1994).

The Algorithm

In the presentation above, we had to tag the text before we could derive new estimates of probabilities $P(t_i|t_{i-1})$ and $P(w_i|t_i)$, or more generally a_{ij} and $b_j(o_t)$. In fact, we can avoid the tagging stage. The coefficients can be computed directly using the forward procedure. We will reestimate $\hat{a}_{ij}$ at step k of the estimation process from estimates a_{ij} at step $k-1$.

The algorithm idea is to consider one observation – one word – and then to average it on all the other observations – the whole sentence. For one specific observation $b_j(o_{t+1})$ at time $t+1$, corresponding here to the word of index $t+1$, the transition probability from state $s_t = q_i$ to state $s_{t+1} = q_j$ corresponds to

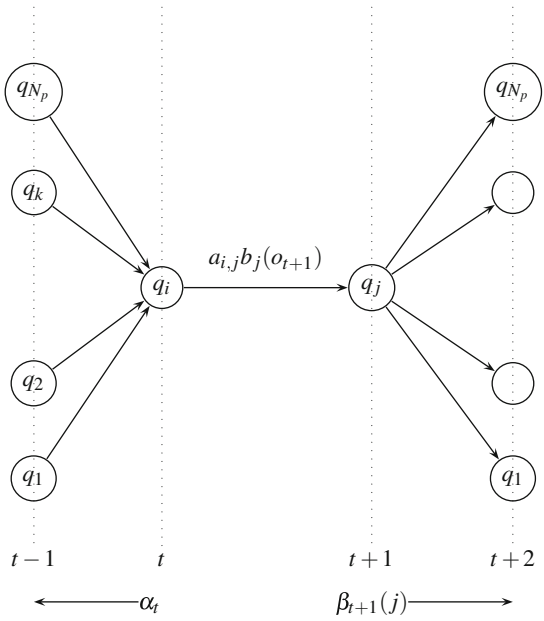


Fig. 8.10 Transition from state q_i at time t to state q_j at time $t+1$ with observation o_{t+1} (After Rabiner (1989))

$$\begin{aligned}\xi_t(i,j) &= P(s_t = q_i, s_{t+1} = q_j | O, \lambda),\\ &= \frac{P(s_t = q_i, s_{t+1} = q_j, O|\lambda)}{P(O|\lambda)},\\ &= \frac{P(s_t = q_i, s_{t+1} = q_j, O|\lambda)}{\sum\limits_{1\le i\le N_p}\sum\limits_{1\le j\le N_p} P(s_t = q_i, s_{t+1} = q_j, O|\lambda)}.\end{aligned}$$

We can use the forward and backward probabilities to determine the estimate. Figure 8.10 shows how to introduce them in the equation.

We have:

$$\xi_t(i,j) = \frac{\alpha_t(i)a_{ij}b_j(o_{t+1})\beta_{t+1}(j)}{\sum\limits_{i=1}^{N_p}\sum\limits_{j=1}^{N_p}\alpha_t(i)a_{ij}b_j(o_{t+1})\beta_{t+1}(j)}.$$

We denote $\gamma_t(i) = \sum\limits_{j=1}^{N_p}\xi_t(i,j)$ the probability to be in state q_i at time t.

To consider all the observations, we sum $\xi_t(i,j)$ from $t = 1$ to $t = n-1$. The expected number of transitions from state q_i to state q_j is $\sum\limits_{t=1}^{n-1}\xi_t(i,j)$, and

the expected number of transitions from state q_i is $\sum_{t=1}^{n-1} \gamma_t(i)$. The last sum also corresponds to the number of times we are in state q_i. We derive:

- The new estimate of a_{ij}:

$$\begin{aligned}\hat{a}_{ij} &= \frac{\text{expected number of transitions from state } q_i \text{ to state } q_j}{\text{expected number of transitions from state } q_i}, \\ &= \frac{\sum_{t=1}^{n-1} \xi_t(i,j)}{\sum_{t=1}^{n-1} \gamma_t(i)}.\end{aligned}$$

- The initial state estimates $\pi_i = \gamma_1(i)$.
- The observation estimates:

$$\begin{aligned}\hat{b}_i(v_k) &= \frac{\text{expected number of times in state } q_i \text{ and observing symbol } v_k}{\text{expected number of times in state } q_i}, \\ &= \frac{^{o_t=v_k, 1\le t\le T}\sum_{t=1}^{n} \gamma_t(i)}{\sum_{t=1}^{n} \gamma_t(i)}.\end{aligned}$$

8.4 POS Tagging with the Perceptron

Hidden Markov models use maximum likelihood estimates of $P(w_i|t_i)$ and $P(t_i|t_{i-2}, t_{i-1})$ to find the optimal part-of-speech sequence. Instead of computing these values from a corpus, Collins (2002) applied the perceptron algorithm to derive equivalent parameters iteratively. This method yields state-of-the-art results and we describe it here. For a description of the perceptron, see Sect. 4.7.

Let us rewrite the optimal part-of-speech sequence from Sect. 8.2 using a trigram approximation and logarithms. We have:

$$\begin{aligned}\hat{T} &= \arg\max_T \left(\log P(T) + \log P(W|T)\right), \\ &= \arg\max_T \Bigg(\log P(t_1) + \log P(t_2|t_1) + \sum_{i=3}^{n} \log P(t_i|t_{i-2}, t_{i-1}) \\ &\qquad + \sum_{i=1}^{n} \log P(w_i|t_i)\Bigg).\end{aligned}$$

Let us denote $\alpha_{t_{i-2},t_{t-1},t_i}$ and α_{t_i,w_i} the parameters equivalent to $P(t_i|t_{i-2},t_{i-1})$ and $P(w_i|t_i)$, respectively. The optimal sequence corresponds to the maximal score:

$$\hat{T} = \arg\max_{T} \left(\sum_{i=1}^{n} \alpha_{t_{i-2},t_{t-1},t_i} + \sum_{i=1}^{n} \alpha_{t_i,w_i} \right),$$

where t_{-2} and t_{-1} are start-of-sentence symbols.

The idea in Collins (2002) is simple. We first create α parameters for each possible trigram sequence x, y, z and tag-word pair t, w and we initialize them:

- $\alpha_{x,y,x} \leftarrow 0$ and $\alpha_{t,w} \leftarrow 0$.

We then apply the perceptron to learn these α parameters. For each sentence of the training corpus, we compute the maximal score using the Viterbi algorithm and we assign the corresponding POS tags to the sentence words. We then update the α parameters with the number of tagging errors in this sentence. Given a sentence of the corpus $w_1, \ldots, w_n$, its hand-annotated part-of-speech sequence $t_1, \ldots, t_n$, and the tagger output $t'_1, \ldots t'_n$, the update rules are:

- $\alpha_{x,y,x} \leftarrow \alpha_{x,y,x} + c_1 - c_2$, where c_1 and c_2 are the respective counts of x, y, z trigrams in the $t_1, \ldots, t_n$ and $t'_1, \ldots t'_n$ sequences.
- $\alpha_{t,w} \leftarrow \alpha_{t,w} + c_1 - c_2$, where c_1 and c_2 are the respective counts of w, t pairs in the $t_1, \ldots, t_n$ and $t'_1, \ldots t'_n$ sequences.

We repeat this procedure N times over the training corpus, N being the number of epochs.

Let us exemplify the update rule with a slightly modified sequence from Collins (2002). If the training corpus has the sentence:

the/DT boy/NN hit/VBD the/DT ball/NN

and the tagger outputs:

the/DT boy/NN hit/NN the/DT ball/NN

The update rule will add one to the parameters:

$$\alpha_{\text{DT,NN,VBD}}, \alpha_{\text{NN,VBD,DT}}, \alpha_{\text{VBD,DT,NN}}, \text{and} \alpha_{\text{VBD},hit}$$

and subtract one to:

$$\alpha_{\text{DT,NN,NN}}, \alpha_{\text{NN,NN,DT}}, \alpha_{\text{NN,DT,NN}}, \text{and} \alpha_{\text{NN},hit}.$$

In his original perceptron, Collins (2002) used more features than the POS trigrams and tag-word pairs described here. For the complete list of features, see Ratnaparkhi (1996).

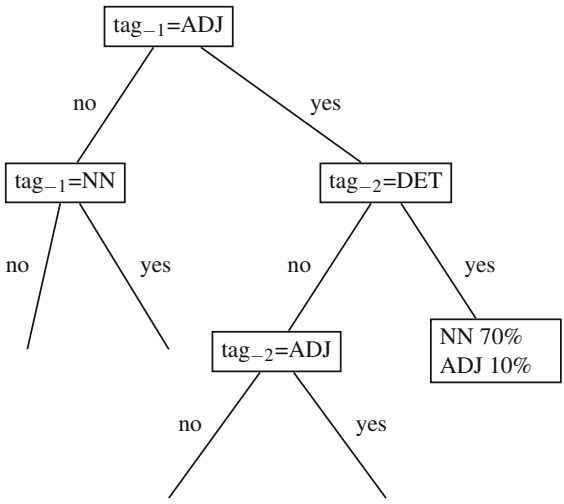


Fig. 8.11 A decision tree to estimate POS frequencies where NN is a noun, DET, a determiner, and ADJ, an adjective (After Schmid (1994))

8.5 Tagging with Decision Trees

As another alternative to hidden Markov models, we can replace the maximum likelihood with decision trees induced from an annotated corpus. The tagging performance could be superior when the training set is small.

TreeTagger (Schmid 1994, 1995) is a stochastic tagger that replaces the maximum likelihood estimate with a binary decision tree to estimate $P(t_i|t_{i-2}, t_{i-1})$. Figure 8.11 shows an example of an imaginary tree where the conditional probability $P(\text{NN}|\text{DET}, \text{ADJ})$ is read from the tree by examining t_{-1} and t_{-2}, here ADJ and DET, respectively. The probability estimate is 0.70.

The decision tree is built from a training set of POS trigrams t_{-2}, t_{-1}, t_0 extracted from an annotated corpus. The condition set is $t_{-i} = v$, with $i \in \{1, 2\}$ and $v \in S$, where S is the tagset.

The idea is to use the entropy of the POS trigams where the random variable is t_0. The entropy is then defined as:

$$-\sum_{t_0 \in S} P(t_0) \log_2 P(t_0).$$

If the total number of tokens is N, the entropy is estimated as:

$$-\sum_{t_0 \in S} \frac{C(t_0)}{N} \log_2 \frac{C(t_0)}{N}.$$

The decision tree minimizes the information it needs to identify the third tag, t_0, given the two preceding tags, t_{-2} and t_{-1}. This reflects the minimal amount of information brought by the third tag of a trigram.

To find the root node, the algorithm creates all the possible partitions of the training set according to the values of t_{-2} and t_{-1}. It computes the weighted average of the entropy of the positive and negative examples. The root condition corresponds to the values i and v with $i \in \{1, 2\}$ and $v \in S$ that minimize

$$-\frac{p}{p+n}\sum_{t_0 \in S}\frac{C(t_0, t_{-i} = v)}{p}\log_2\frac{C(t_0, t_{-i} = v)}{p} - \frac{n}{p+n}\sum_{t_0 \in S}\frac{C(t_0, t_{-i} \neq v)}{n} \times \log_2\frac{C(t_0, t_{-i} \neq v)}{n},$$

where p is the count of the trigrams that pass the test to be the root condition, and n is the count of trigrams that do not pass the test. $C(t_0, t_{-1} = v)$ is the count of trigrams t_{-2}, t_{-1}, t_0 that pass the test and where the third tag is t_0, and $C(t_0, t_{-1} \neq v)$ the count of trigrams that do not pass the test and where the third tag is t_0.

The algorithm stops expanding the tree and creates a leaf when the next node would gather a number of positive or negative trigrams below a certain threshold, 2, for example.

8.6 Unknown Words

For stochastic taggers, the main issue to tag unknown words is to estimate $P(w|t)$. Carlberger and Kann (1999) proposed to use suffixes or, more precisely, word endings to compute the estimate. They counted the number of word types with common word endings of length i, $C(w_{end-i}, t)$, for each tag t in the tagset, with i ranging from 0 to L. The estimate $P(w|t)$ for an unknown word is then

$$P_{est}(w|t) = \sum_{i=0}^{L}\alpha_i\frac{C(w_{end-i}, t)}{\sum_{\tau \in \text{tagset}} C(w_{end-i}, \tau)}.$$

where α_i are parameters optimized on the training set. They tried their formula with increasing values of L, and they found that tagging accuracy did not improve for $L > 5$.

If $L = 0$, $P_{est}(w|t) = \frac{C(t)}{\sum_{\tau \in \text{tagset}} C(\tau)}$ corresponds to the proportion of part of speech t among the word types.

We can also use word prefixes and suffixes, this time in the form of features, with taggers based on linear classifiers (Sect. 8.1) or Collins' perceptron (Sect. 8.4). Ratnaparkhi (1996), for example, used prefixes and suffixes ranging from 1 to 4 letters, to represent rare or unknown words.

8.7 An Application of the Noisy Channel Model: Spell Checking

An interesting application of the noisy channel model is to help a spell-checker rank candidate corrections (Kernighan et al. 1990). In this case, the source sequence is a correct string c that produces an incorrect one called the typo t through the noisy channel. The most likely correction is modeled as

$$\hat{c} = \arg\max P(c)P(t|c).$$

Possible typos are deletion, insertion, substitutions, and transpositions. In their original paper, Kernighan et al. (1990) allowed only one typo per word. Typo frequencies are estimated from a corpus where:

- $del(xy)$ is the number of times the characters xy in the correct word were typed x in the training set.
- $ins(xy)$ is the number of times x was typed as xy in the training set.
- $sub(xy)$ is the number of times the character y was typed as x.
- $trans(xy)$ is the number of times xy was typed as yx in the training set.

$P(t|c)$ is estimated as:

$$P(t|c) = \begin{cases} \dfrac{del(c_{i-1}, c_i)}{C(c_{i-1}, c_i)} & \text{if deletion,} \\ \dfrac{ins(c_{i-1}, t_i)}{C(c_{i-1})} & \text{if insertion,} \\ \dfrac{sub(t_i, c_i)}{C(c_i)} & \text{if substitution,} \\ \dfrac{trans(c_i, c_{i+1})}{C(c_i, c_{i+1})} & \text{if transposition.} \end{cases}$$

where c_i is the ith character of c, and t_i the ith of t.

The algorithm needs four confusion matrices, of size 26×26 for English, that contain the frequencies of deletions, insertions, substitutions, and transpositions. The *del* matrix will give the counts $del(xy)$, how many times y was deleted after x for all the letter pairs, for instance, $del(ab)$.

The matrices can be obtained through hand-annotation or automatically. Hand-annotation is expensive, and Kernighan et al. (1990) described an algorithm to automatically train the matrices. It resembles the forward–backward procedure introduced in Sect. 8.3.8.

The training phase initializes the matrices with equal values and applies the spelling algorithm to generate a correct word for each typo in the text. The pairs typo/corrected word are used to update the matrices. The algorithm is repeated on the original text to obtain new pairs and is iterated until the matrices converge.

8.8 A Second Application: Language Models for Machine Translation

Natural language processing was born with machine translation, which was one of its first applications. Facing competition from Russia after the Second World War, the government of the United States decided to fund large-scale translation programs to have quick access to documents written in Russian. It started the field and resulted in programs like SYSTRAN, which are still in use today.

Given the relatively long history of machine translation, a variety of methods have been experimented on and applied. In this section, we outline how language models and statistical techniques can be used to translate a text from one language into another one. IBM teams pioneered statistical models for machine translation in the early 1990s (Brown et al. 1993). Their work is still the standard reference.

8.8.1 Parallel Corpora

Parallel corpora are the main resource of statistical language translation. Administrative or parliamentary texts of multilingual countries are widely used because they are easy to obtain and are often free. The Canadian Hansard or the European Parliament proceedings are examples of them. Table 8.7 shows an excerpt of the Swiss federal law in German, French, and Italian on the quality of milk production.

The idea of machine translation with parallel texts is simple: given a sentence, a phrase, or a word in a **source language**, find its equivalent in the **target language**. The translation procedure splits the text to translate into fragments, finds a correspondence for each source fragment in the parallel corpora, and composes the resulting target pieces to form a translated text. Using the titles in Table 8.7, we can build pairs from the phrases *transport du lait* 'milk transportation' in French, *Milchtransport* in German, and *trasporto del latte* in Italian.

The idea of translating with the help of parallel texts is not new and has been applied by many people. A notable example is the Egyptologist and linguist Jean-François Champollion, who used the famous Rosetta Stone, an early parallel text, to decipher Egyptian hieroglyphs from Greek.

8.8.2 Alignment

The parallel texts must be aligned before using them in machine translation. This corresponds to a preliminary segmentation and mark-up that determines the corresponding paragraphs, sentences, phrases, and words across the texts. Inside sentences, aligned fragments are called **beads**. Alignment of texts in Table 8.7 is made easier because paragraphs are numbered and have the same number of

Table 8.7 Parallel texts from the Swiss federal law on milk transportation

German	French	Italian
Art. 35 Milchtransport	**Art. 35 Transport du lait**	**Art. 35 Trasporto del latte**
1 Die Milch ist schonend und hygienisch in den Verarbeitungsbetrieb zu transportieren. Das Transportfahrzeug ist stets sauber zu halten. Zusammen mit der Milch dürfen keine Tiere und milchfremde Gegenstände transportiert werden, welche die Qualität der Milch beeinträchtigen können.	1 Le lait doit être transporté jusqu'à l'entreprise de transformation avec ménagement et conformément aux normes d'hygiène. Le véhicule de transport doit être toujours propre. Il ne doit transporter avec le lait aucun animal ou objet susceptible d'en altérer la qualité.	1 Il latte va trasportato verso l'azienda di trasformazione in modo accurato e igienico. Il veicolo adibito al trasporto va mantenuto pulito. Con il latte non possono essere trasportati animali e oggetti estranei, che potrebbero pregiudicarne la qualità.
2 Wird Milch ausserhalb des Hofes zum Abtransport bereitgestellt, so ist sie zu beaufsichtigen.	2 Si le lait destiné à être transporté est déposé hors de la ferme, il doit être placé sous surveillance.	2 Se viene collocato fuori dall'azienda in vista del trasporto, il latte deve essere sorvegliato.
3 Milchpipelines sind nach den Anweisungen des Herstellers zu reinigen und zu unterhalten.	3 Les lactoducs des exploitations d'estivage doivent être nettoyés et entretenus conformément aux instructions du fabricant.	3 I lattodotti vanno puliti e sottoposti a manutenzione secondo le indicazioni del fabbricante.

sentences in each language. This is not always the case, however, and some texts show a significantly different sentence structure.

Gale and Church (1993) describe a simple and effective method based on the idea that

> longer sentences in one language tend to be translated into longer sentences in the other language, and that shorter sentences tend to be translated into shorter sentences.

Their method generates pairs of sentences from the target and source texts, assigns them a score, which corresponds to the difference of lengths in characters of the aligned pairs, and uses dynamic programming to find the maximum likelihood alignment of sentences.

The sentences in the source language are denoted $s_i, 1 \leq i \leq I$, and the sentences in the target language $t_i, 1 \leq i \leq J$. $D(i, j)$ is the minimum distance between sentences $s_1, s_2, \ldots, s_i$ and $t_1, t_2, \ldots, t_j$, and $d(source_1, target_1; source_2, target_2)$ is the distance function between sentences. The algorithm identifies six possible cases of alignment through insertion, deletion, substitution, expansion, contraction, or merger. They are expressed by the formula below:

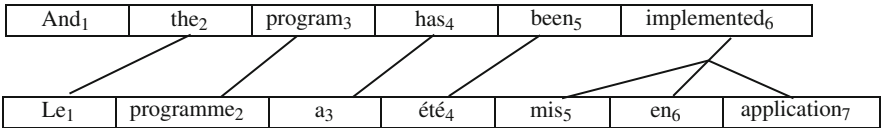


Fig. 8.12 Alignment (After Brown et al. (1993))

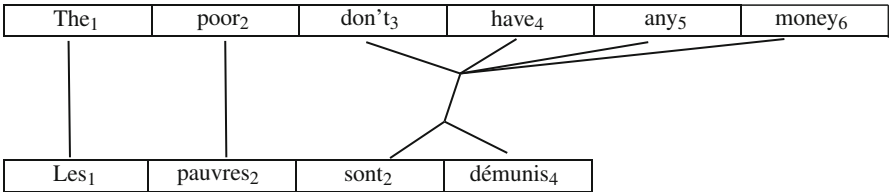


Fig. 8.13 A general alignment (After Brown et al. (1993))

$$D(i,j) = \min \begin{pmatrix} D(i,j-1) + d(0,t_j;0,0) \\ D(i-1,j) + d(s_i,0;0,0) \\ D(i-1,j-1) + d(s_i,t_j;0,0) \\ D(i-1,j-2) + d(s_i,t_j;0,t_{j-1}) \\ D(i-2,j-1) + d(s_i,t_j;s_{i-1},0) \\ D(i-2,j-2) + d(s_i,t_j;s_{i-1},t_{j-1}) \end{pmatrix}.$$

The distance function is defined as $-\log P(alignment|\delta)$, with $\delta = (l_2 - l_1 c)/\sqrt{l_1 s^2}$, and where l_1 and l_2 are the lengths of the sentences under consideration, c the average number of characters in the source language L_2 per character in the target language L_1, and s^2 its variance. Gale and Church (1993) found a value of c of 1.06 for the pair French–English and 1.1 for German–English. This means that French and German texts are longer than their English counterparts: 6 % longer for French and 10 % for German. They found $s^2 = 7.3$ for German–English and $s^2 = 5.6$ for French–English.

Using Bayes' theorem, we can derive a new distance function:

$$-\log P(\delta|alignment) - \log P(alignment).$$

Gale and Church (1993) estimated the probability $P(alignment)$ of their six possible alignments with these figures: substitution 1–1: 0.89, deletion and substitution 0–1 or 1–0: 0.0099, expansion and contraction 2–1 or 1–2: 0.089, and merger 2–2: 0.011. They rewrote $P(\delta|alignment)$ as $2(1 - P(|\delta|))$, which can be computed from statistical tables. See Gale and Church's original article.

Alignment of words and phrases uses similar techniques, however, it is more complex. Figures 8.12 and 8.13 show examples of alignment from Brown et al. (1993).

8.8.3 Translation

Using a statistical formulation, given a source text, S, the most probable target text, T, corresponds to $\arg\max_T P(T|S)$, which can be rewritten as $\arg\max_T P(T)P(S|T)$. The first term, $P(T)$, is a language model, for instance, a trigram model, and the second one, $P(S|T)$, is the translation model. In their original article, Brown et al. (1993) used French as the source language and English as the target language with the notations F and E. They modeled the correspondence between a French string $f = f_1, f_2, \ldots, f_m$ and an English string, $e = e_1, e_2, \ldots, e_l$.

The first step is to rewrite the translation model as

$$P(f|e) = \sum_a P(f, a|e),$$

where a is the alignment between the source and target sentences and where each source word has one single corresponding target word. The target word can be the empty string. The alignment is represented by the string $a = a_1, a_2, \ldots, a_m$, where a_j is the position of the corresponding word in the English string as $a_j = i$, which denotes that word j in the French string is connected to word i in the English string. When there is no connection $a_j = 0$. In the example of Fig. 8.12, we have the alignment $a = (2, 3, 4, 5, 6, 6, 6)$.

Brown et al. (1993) proposed five models ranging from relatively simple to pretty elaborate to work out concretely the formula. In their simplest model 1, they introduce the simplification:

$$P(f, a|e) = \frac{\varepsilon}{(l+1)^m} \prod_{j=1}^{m} t(f_j|e_{a_j}),$$

where $t(f_j|e_{a_j})$ is the translation probability of f_j given e_{a_j} and ε a small, fixed number.

Using the example in Fig. 8.12, the product in

$$P(\text{Le programme a été mis en application}, a|\text{And the program has been implemented})$$

for $a = (2, 3, 4, 5, 6, 6, 6)$ corresponds to the terms:

$$t(\text{Le}|\text{the}) \times t(\text{programme}|\text{program}) \times t(\text{a}|\text{has}) \times t(\text{été}|\text{been}) \times$$
$$t(\text{mis}|\text{implemented}) \times t(\text{en}|\text{implemented}) \times t(\text{application}|\text{implemented})$$

where t values are derived from aligned corpora. Summing over all the possible alignments, we obtain the probability of the translation of *Le programme a été mis en application* into *And the program has been implemented.*

8.8.4 *Evaluating Translation*

The results of automatic translation are most frequently evaluated using the bilingual evaluation understudy (BLEU) algorithm (Papineni et al. 2002).

BLEU compares the machine translation of a text with corresponding human translations. It uses a test set, where each sentence is translated by one or more human beings, and computes a score for each sentence and an average on the test set. The most basic score is a word-for-word comparison. It corresponds to the number of machine-translated words that appear in the human translations divided by the total number of words in the machine-translated sentence. The final score on the test set ranges from 0 to 1.

Papineni et al. noted that sequences of repeated words, such as articles, could reach high scores even if they made no sense. They modified their first algorithm in consequence by setting a maximal count for each word. This maximal count is computed from the human translations.

BLEU extends the word-for-word comparison to n-grams with the counts of machine-translated n-grams matching the human translations and divided by the total number of n-grams in the machine-translated sentence.

8.9 Further Reading

There are plenty of techniques to carry out part-of-speech tagging. We reviewed the most popular ones in this chapter. Carlberger and Kann (1999) is a very readable and complete text to implement a HMM tagger, while HunPos (Halácsy et al. 2007) is a modern, compact, and open-source implementation (http://code.google.com/p/hunpos/).

Ratnaparkhi (1996) proposed a method similar to hidden Markov models, but he used probability estimates from logistic regression instead of the maximum likelihood. For a given word, the probability is conditioned on the sentence words and the parts of speech already assigned: $P_{\text{LogReg}}(t_i | w_{1..N}, t_{1..j})$ with $j < i$. Usually, the sequence is limited to a window of five words.

Conditional random fields (Lafferty et al. 2001) are an extension of Ratnaparkhi's method that is conditioned on the complete word and tag sequences except the current tag: $P_{\text{LogReg}}(t_i | w_{1..N}, t_{1..i-1,i+1..N})$. In practice, the sequence is a limited window too. Although appealing and very frequently cited, conditional random fields did not outperform other methods until now. They are also more difficult to train and to apply.

We briefly introduced machine translation in this chapter. Brown et al. (1993) started the field on statistical translation models. The original article is worth reading. Koehn (2010) is a recent and comprehensive overview. Statistical techniques have tremendously improved translation quality over the 10 last years. Google Translate is the most notable example of this trend. As notable software resources,

GIZA++ (Och and Ney 2003) is a program to train alignment models available from: http://code.google.com/p/giza-pp/ and Moses is a complete statistical machine translation system http://www.statmt.org/moses/.

Exercises

8.1. Implement a part-of-speech tagger using logistic regression or support vector machines. You can use LIBLINEAR or LIBSVM.

8.2. Implement the HMM part-of-speech tagging algorithm in Prolog or Perl using unigrams.

8.3. Implement the HMM part-of-speech tagging algorithm in Prolog or Perl using bigrams without the Viterbi algorithm.

8.4. Complement the previous program with the Viterbi search.

8.5. Implement a spell-checker in Prolog or Perl.

Chapter 9
Phrase-Structure Grammars in Prolog

ἔσται πᾶσα κατάφασις ἢ ἐξ ὀνόματος καὶ ῥήματος ἢ ἐξ ἀορίστου ὀνόματος καὶ ῥήματος.
'Every affirmation, then, and every denial, will consist of a noun and a verb, either definite or indefinite.'

Aristotle, *De Interpretatione*, Chap. 10. Translated by E. M. Edghill.

Simplicium vero enuntiationum partes sunt subjectum atque prædicatum.

'The parts of a simple proposition are the subject and the predicate.'

Boethius (470-525), *Introductio ad syllogismos categoricos*, In Patrologica Latina, 64, page 768 C.

Sentence $\rightarrow NP + VP$

Chomsky (1957, Chap. 4)

Three inventions, and teachings, in their time.

9.1 Using Prolog to Write Phrase-Structure Grammars

This chapter introduces parsing using phrase-structure rules and grammars. It uses the Definite Clause Grammar (DCG) notation (Pereira and Warren 1980), which is a feature of virtually all Prologs. The DCG notation enables us to transcribe a set of phrase-structure rules directly into a Prolog program.

Prolog was designed from the very beginning for language processing. It has built-in search and unification mechanisms that make it naturally suited to implement formal models of linguistics with elegance and concision. Parsing with DCG rules comes down to a search in Prolog. Prolog recognizes the rules at load

P.M. Nugues, *Language Processing with Perl and Prolog*, Cognitive Technologies,
DOI 10.1007/978-3-642-41464-0_9, © Springer-Verlag Berlin Heidelberg 2014

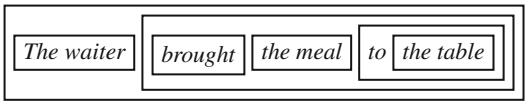


Fig. 9.1 The constituent structure of *The waiter brought the meal to the table*

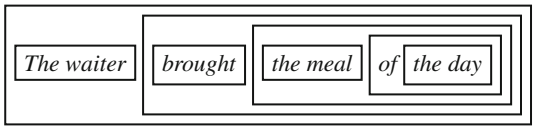


Fig. 9.2 The constituent structure of *The waiter brought the meal of the day*

time and translates them into clauses. Its engine automatically carries out the parse without the need for additional programming.

Many natural language processing systems, both in academia and in industry, have been written in Prolog. Other languages like Perl, Python, Java, or C++ are now widely used in language engineering applications. However, much programming is often necessary to implement an idea or a linguistic theory. Prolog gets to the heart of the problem in sometimes only a few lines of code. It thus enables us to capture fundamental concepts while setting aside coding chores.

9.2 Representing Chomsky's Syntactic Formalism in Prolog

9.2.1 Constituents

Chomsky's syntactic formalism (1957) is based on the concept of constituents. Constituents can be defined as groups of words that fit together and act as relatively independent syntactic units. We shall illustrate this idea with the sentences:

The waiter brought the meal.
The waiter brought the meal to the table.
The waiter brought the meal of the day.

Phrases such as *the waiter*, *the meal*, *of the day*, or *brought the meal of the day* are constituents because they sound natural. On the contrary, the groups of words *meal to* or *meal of the* sound odd or not complete and therefore are not constituents.

The set of constituents in a sentence includes all the phrases that meet this description. Simplest constituents are the sentence's words that combine with their neighbors to form larger constituents. Constituents combine again and extend up to the sentence itself. Constituents can be pictured by boxed groups of sentence chunks (Figs. 9.1 and 9.2).

In Fig. 9.2, the phrase *the meal of the day* fits in a box, while in Fig. 9.1, *the meal* and *to the table* are separated. The reason is semantic. *The meal of the day* can be considered as a single entity, and so *of the day* is attached to *the meal*. Both can merge in a single constituent and hence fit in the same box. *To the table* is related to the sentence verb rather than to *the meal*: this phrase specifies where the waiter

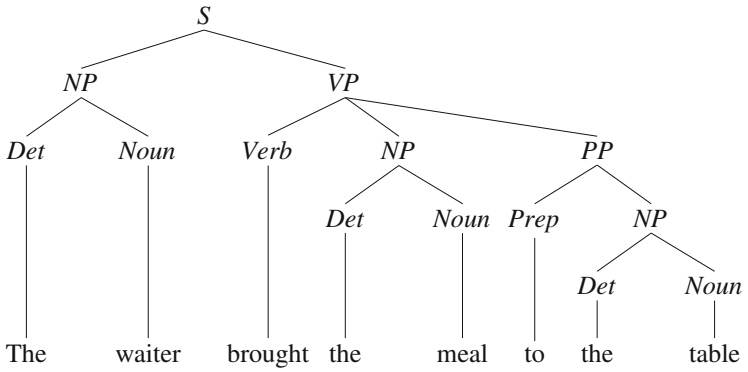


Fig. 9.3 Tree structure of *The waiter brought the meal to the table*

brought something. That is why the next enclosing box frames the phrase *brought the meal to the table* and not *the meal to the table.*

Constituents are organized around a headword that usually has the most significant semantic content. The constituent category takes its name from the headword part of speech. So, *the waiter, the meal, the day*, and *the meal of the day* are noun phrases (*NP*s), and *brought the meal of the day* is a verb phrase (*VP*). Prepositional phrases (*PP*s) are noun phrases beginning with prepositions such as *to the table* and *of the day*.

9.2.2 Tree Structures

Tree structures are an alternate representation to boxes where constituent names annotate the tree nodes. The symbol *S* denotes the whole sentence and corresponds to the top node. This node divides into two branches that lead to the *NP* and *VP* nodes, and so on. Figure 9.3 shows the structure of *The waiter brought the meal to the table*, and Fig. 9.4 the structure of *The waiter brought the meal of the day*.

9.2.3 Phrase-Structure Rules

Phrase-structure rules (PS rules) are a device to model constituent structures. PS rules rewrite the sentence or phrases into a sequence of simpler phrases that describe the composition of the tree nodes. More precisely, a PS rule has a left-hand side that is the parent symbol and a right-hand side made of one, two, or more symbols labeling the downward-connected nodes. For instance, rule

$S \rightarrow NP\ VP$

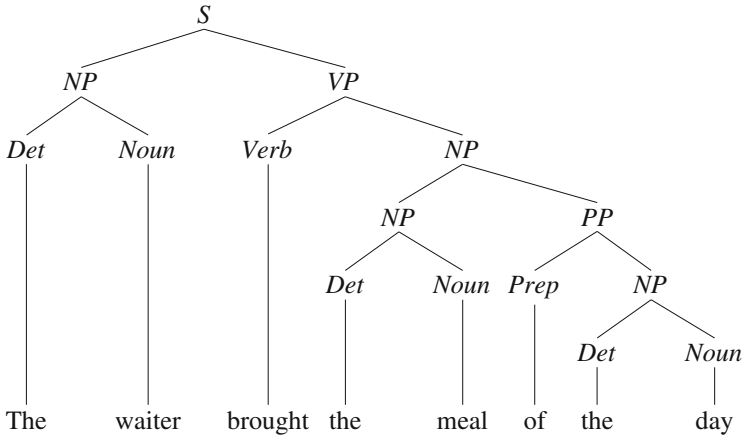


Fig. 9.4 Tree structure of *The waiter brought the meal of the day*

Table 9.1 A phrase-structure grammar

Phrases	**Lexicon**	
S → *NP VP*	*Determiner* → *the*	*Noun* → *day*
NP → *Determiner Noun*	*Noun* → *waiter*	*Verb* → *brought*
NP → *NP PP*	*Noun* → *meal*	*Preposition* → *to*
VP → *Verb NP*	*Noun* → *table*	*Preposition* → *of*
VP → *Verb NP PP*		
PP → *Preposition NP*		

describes the root node of the tree: a sentence can consist of a noun phrase and a verb phrase.

A phrase-structure grammar is a set of PS rules that can decompose sentences and phrases down to the words and describe complete trees. The phrase categories occurring in Figs. 9.3 and 9.4 are sentence, noun phrase, verb phrase, and prepositional phrase. In the phrase-structure formalism, these categories are called the **nonterminal symbols**. Parts of speech or lexical categories here are determiners (or articles), nouns, verbs, and prepositions. PS rules link up categories to rewrite the sentence and the phrases until they reach the words – the **terminal symbols**. Table 9.1 shows a grammar to parse the sentences in Figs. 9.1 and 9.2.

The first rule in Table 9.1 means that the sentence consists of a noun phrase followed by a verb phrase. The second and third rules mean that a noun phrase can consist either of a determiner and a noun, or a noun phrase followed by a prepositional phrase, and so on. The left constituent is called the **mother** of the rule, and the right constituents are its **expansion** or its **daughters**. The sequence of grammar rules applied from the sentence node to get to the words is called a **derivation**.

9.2.4 The Definite Clause Grammar (DCG) Notation

The translation of PS rules into DCG rules is straightforward. The DCG notation uses the `-->/2` built-in operator to denote that a constituent can consist of a sequence of simpler constituents. DCG rules look like ordinary Prolog clauses except that the operator `-->/2` separates the head and body instead of `:-/2`. Let us use the symbols `s`, `np`, `vp`, and `pp` to represent phrases. The grammar in Table 9.1 corresponds to DCG rules:

```
s --> np, vp.
np --> det, noun.
np --> np, pp.
vp --> verb, np.
vp --> verb, np, pp.
pp --> prep, np.
```

DCG rules encode the vocabulary similarly. The left-hand side of the rule is the part of speech, and the right-hand side is the word put inside a list – enclosed between brackets:

```
det --> [the].
det --> [a].
noun --> [waiter].
noun --> [meal].
noun --> [table].
noun --> [day].
verb --> [brought].
prep --> [to].
prep --> [of].
```

The Prolog search mechanism checks whether a fact is true or generates all the solutions. Applied to parsing, the search checks whether a sentence is acceptable to the grammar or generates all the sentences accepted by this grammar.

Once the Prolog interpreter has consulted the DCG rules, we can query it using the input word list as a first parameter and the empty list as a second. Both queries:

```
?- s([the, waiter, brought, the, meal, to, the,
table], []). true

?- s([the, waiter, brought, the, meal, of, the, day],
[]). true
```

succeed because the grammar accepts the sentences.

In addition to accepting sentences, the interpreter finds all the sentences generated by the grammar. It corresponds to the so-called syntactically correct sentences:

```
?-s(L, []).
L = [the, waiter, brought, the, waiter] ;
```

```
L = [the, waiter, brought, the, meal] ;
L = [the, waiter, brought, the, table] ;
...
```

In the grammar above, the two first lexical rules mean that a determiner can be either *the* or *a*. This rule could have been compacted in a single one using Prolog's disjunction operator `;/2` as:

```
det --> [the] ; [a].
```

However, like for Prolog programs, using the semicolon operator sometimes impairs the readability and is not advisable.

In our grammar, nonterminal symbols of lexical rules are limited to a single word. They can also be a list of two or more words as in:

```
prep --> [in, front, of].
```

which means that the word sequence *in front of* corresponds to a preposition.

DCG rules can mix terminal and nonterminal symbols in their expansion as in:

```
np --> noun, [and], noun.
```

Moreover, Prolog programs can mix Prolog clauses with DCG rules, and DCG rules can include Prolog goals in the expansion. These goals are enclosed in braces:

```
np --> noun, [and], noun, {prolog_code}.
```

as, for example:

```
np -->
  noun, [and], noun,
  {write('I found two nouns'), nl}.
```

9.3 Parsing with DCGs

9.3.1 Translating DCGs into Prolog Clauses

Prolog translates DCG rules into Prolog clauses when the file is consulted. The translation is nearly a mapping because DCG rules are merely a notational variant of Prolog rules and facts. In this section, we will first consider a naïve conversion method. We will then outline how most common interpreters adhering to Edinburgh Prolog (Pereira 1984) tradition carry out the translation.

A tentative translation of DCG rules in Prolog clauses would add a variable to each predicate. The rule

```
s --> np, vp.
```

would then be converted into the clause

```
s(L) :- np(L1), vp(L2) ...
```

so that each variable unifies with the word list corresponding to the predicate name. With this kind of translation and the input sentence *The waiter brought the meal*, variable

- `L` would match the input list `[the, waiter, brought, the, meal]`;
- `L1` would match the noun phrase list `[the, waiter]`; and
- `L2` would match the verb phrase `[brought, the, meal]`.

To be complete, the Prolog clause requires an `append/3` predicate at the end to link `L1` and `L2` to `L`:

```
s(L) :- np(L1), vp(L2), append(L1, L2, L).
```

Although this clause might seem easy to understand, it would not gracefully scale up. If there were three daughters, the rule would require two `append`s, and if there were four daughters, the rule would then need three `append`s, and so on.

In most Prologs, the translation predicate adds two variables to each DCG symbol to the left-hand side and the right-hand side of the rule. The DCG rule

```
s --> np, vp.
```

is actually translated into the Prolog clause

```
s(L1, L) :- np(L1, L2), vp(L2, L).
```

where `L1`, `L2`, and `L` are lists of words. As with the naïve translation, the clause expresses that a constituent matching the head of the rule is split into subconstituents matching the goals in the body. However, constituent values correspond to the difference of each pair of arguments.

- *The waiter brought the meal* corresponds to the `s` symbol and unifies with `L1\L`, where `L1\L` denotes `L1` minus `L`.
- *The waiter* corresponds to the `np` symbol and unifies with `L1\L2`.
- *brought the meal* corresponds to the `vp` symbol and unifies with `L2\L`.

In terms of lists, `L1\L` corresponds to `[the, waiter, brought, the, meal]`; `L1\2` corresponds to the first noun phrase `[the, waiter]`; and `L2\L` corresponds to the verb phrase and `[brought, the, meal]`.

`L1` is generally set to the input sentence and `L` to the empty list, `[]`, when querying the Prolog interpreter, as in:

```
?- s([the, waiter, brought, the, meal], []).
true
```

So the variables `L1` and `L2` unify respectively with `[the, waiter, brought, the, meal]` and `[brought, the, meal]`.

The lexical rules are translated the same way. The rule

```
det --> [the].
```

is mapped onto the fact:

```
det([the | L], L).
```

Sometimes, terminal symbols are rewritten using the `'C'/3` (connects) built-in predicate. In this case, the previous rule could be rewritten into:

```
det(L1, L) :- 'C'(L1, the, L).
```

The `'C'/3` predicate links `L1` and `L` so that the second parameter is the head of `L1` and `L`, its tail. `'C'/3` is defined as:

```
'C'([X | Y], X, Y).
```

In many Prologs, the translation of DCG rules into Prolog clauses is carried out by a predicate named `expand_term/2`.

9.3.2 Parsing and Generation

DCG parsing corresponds to Prolog's top-down search that starts from the **start symbol**, `s`. Prolog's search mechanism rewrites `s` into subgoals, here `np` and `vp`. Then it rewrites the leftmost symbols starting with `np` and goes down until it matches the words of the input list with the words of the vocabulary. If Prolog finds no solution with a set of rules, it backtracks and tries other rules.

Let us illustrate a search tracing the parser with the sentence *The waiter brought the meal* in Table 9.2. The interpreter is launched with the query

```
?- s([the, waiter, brought, the, meal], []).
```

The Prolog clause

```
s(L1, L) :- np(L1, L2), vp(L2, L).
```

is called first (Table 9.2, line 1). The leftmost predicate of the body of the rule, `np`, is then tried. Rules are examined in the order they occur in the file, and

```
np(L1, L) :- det(L1, L2), noun(L2, L).
```

is then called (line 2). The search continues with `det` (line 3) that leads to the terminal rules. It succeeds with the fact

```
det([the | L], L).
```

and unifies `L` with `[waiter, brought, the, meal]` (line 4). The search skips from `det/2` to `noun/2` in the rule

```
np(L1, L) :- det(L1, L2), noun(L2, L).
```

`noun/2` is searched the same way (lines 5 and 6). `np` succeeds and returns with `L` unified with `[brought, the, meal]` (line 7). The rule

```
s(L1, L) :- np(L1, L2), vp(L2, L).
```

proceeds with `vp` (line 8) until `s` succeeds (line 18). The search is pictured in Fig. 9.5.

Table 9.2 Trace of *The waiter brought the meal*

```
1    Call:   s([the, waiter, brought, the, meal], [])
2    Call:   np([the, waiter, brought, the, meal], _2)
3    Call:   det([the, waiter, brought, the, meal], _6)
4    Exit:   det([the, waiter, brought, the, meal], [waiter,
               brought, the, meal])
5    Call:   noun([waiter, brought, the, meal], _2)
6    Exit:   noun([waiter, brought, the, meal], [brought, the,
               meal])
7    Exit:   np([the, waiter, brought, the, meal], [brought, the,
               meal])
8    Call:   vp([brought, the, meal], [])
9    Call:   verb([brought, the, meal], _10)
10   Exit:   verb([brought], [the, meal])
11   Call:   np([the, meal], [])
12   Call:   det([the, meal], _11)
13   Exit:   det([the, meal], [meal])
14   Call:   noun([meal], [])
15   Exit:   noun([meal], [])
16   Exit:   np([the, meal], [])
17   Exit:   vp([brought, the, meal], [])
18   Exit:   s([the, waiter, brought, the, meal], [])
```

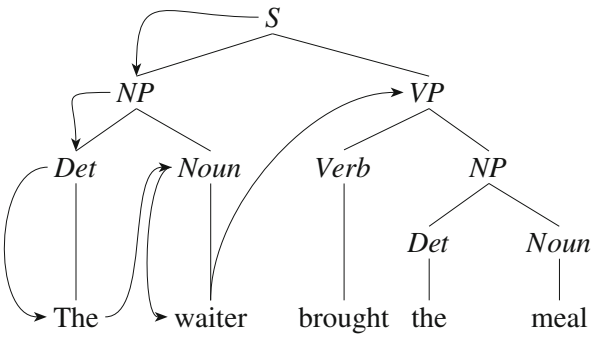


Fig. 9.5 The DCG parsing process

9.3.3 Left-Recursive Rules

We saw that the DCG grammar in Table 9.1 accepts and generates correct sentences, but what about incorrect ones? A first guess is that the grammar should reject them. In fact, querying this grammar with *The brought the meal* (*) never returns or even crashes Prolog. This is due to the left-recursive rule

```
np --> np, pp.
```

Incorrect strings, such as:

The brought the meal (*)

trap the parser into an infinite loop. Prolog first tries to match *The brought* to

```
np --> det, noun.
```

Since *brought* is not a noun, it fails and tries the next rule

```
np --> np, pp.
```

Prolog calls `np` again, and the first `np` rule is tried anew. The parser loops hopelessly.

The classical method to get rid of the left-recursion is to use an auxiliary rule with an auxiliary symbol (`ngroup`), which is not left-recursive, and to rewrite the noun phrase rules as:

```
ngroup --> det, noun.
np --> ngroup.
np --> ngroup, pp.
```

When a grammar does not contain left-recursive rules, or once left-recursion has been removed, any sentence not accepted by the grammar makes Prolog fail:

```
?- s([the, brought, the, meal, to, the, table], []).
false
```

9.4 Parsing Ambiguity

The tree structure of a sentence reflects the search path that Prolog is traversing. With the rule set we used, verb phrases containing a prepositional phrase can be parsed along to two different paths. The rules

```
vp --> verb, np.
np --> np, pp.
```

give a first possible path. Another path corresponds to the rule

```
vp --> verb, np, pp.
```

This alternative corresponds to a syntactic ambiguity.

Two parse trees reflect the result of a different syntactic analysis for each sentence. Parsing

The waiter brought the meal to the table

corresponds to the trees in Figs. 9.3 and 9.6. Parsing

The waiter brought the meal of the day

corresponds to the trees in Figs. 9.4 and 9.7.

In fact, only Figs. 9.3 and 9.4 can be viewed as correct because the prepositional phrases attach differently in the two sentences. In

The waiter brought the meal to the table

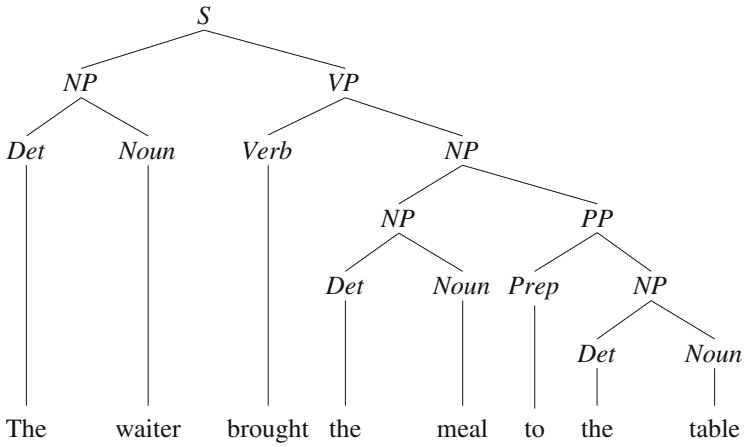


Fig. 9.6 A possible parse tree for *The waiter brought the meal to the table*

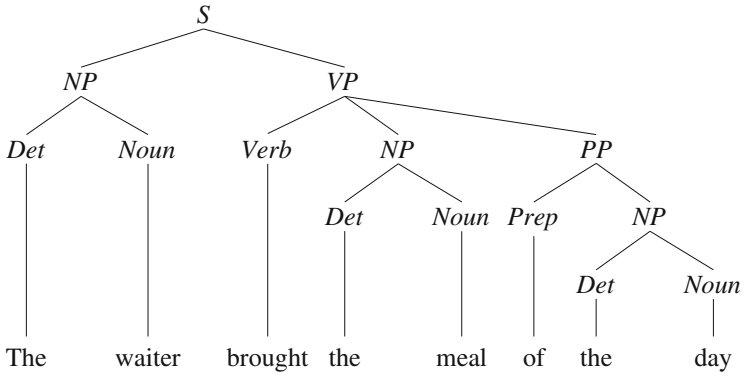


Fig. 9.7 A possible parse tree for *The waiter brought the meal of the day*

the object is *the meal* that the waiter brings to a specific location, *the table*. These are two distinct entities. In consequence, the phrase *to the table* is a verb adjunct and must be attached to the verb phrase node.

In the sentence

The waiter brought the meal of the day

the verb object is *the meal of the day*, which is an entity in itself. The phrase *of the day* is a **postmodifier** of the noun *meal* and must be attached to the noun phrase node.

When we hear such ambiguous sentences, we unconsciously retain the one that is acceptable from a pragmatic viewpoint. Prolog does not have this faculty, and the parser must be hinted. It can be resolved by considering verb, preposition, and noun

types using logical constraints or statistical methods. It naturally requires adding some more Prolog code. In addition, sentences such as

I saw a man with a telescope

remain ambiguous, even for humans.

9.5 Using Variables

Like Prolog, DCG symbols can have variables. These variables can be used to implement a set of constraints that may act on words in a phrase. Such constraints govern, for instance, the number and gender agreement, the case, and the verb transitivity. Variables can also be used to get the result from a parse. They enable us to build the parse tree and the logical form while parsing a sentence.

DCG variables will be kept in their Prolog predicate counterpart after consulting. Variables of a DCG symbol appear in front of the two list variables that are added by `expand_term/2` while building the Prolog predicate. That is, the DCG rule

```
np(X, Y, Z) --> det(Y), noun(Z).
```

is translated into the Prolog clause

```
np(X, Y, Z, L1, L) :-
  det(Y, L1, L2),
  noun(Z, L2, L).
```

9.5.1 Gender and Number Agreement

French and German nouns have a gender and a number that must agree with that of the determiner and the adjective. Genders in French are masculine and feminine. German also has a neuter gender. Number is singular or plural. Let us use variables `Gender` and `Number` to represent them in the noun phrase rule and to impose the agreement:

```
np(Gender, Number) -->
  det(Gender, Number), noun(Gender, Number).
```

To keep the consistency along with all the rules of the grammar, lexical rules must also describe the gender and number of words (Table 9.3).

A Prolog query on `np` with the French vocabulary loaded generates two noun phrases whose determiner and noun agree in gender:

```
?- np(Gender, Number, L, []).
Gender = masc, Number = sing, L = [le, garçon];
Gender = fem, Number = sing, L = [la, serveuse];
No
```

Table 9.3 A vocabulary with gender and number

French	German
`det(masc, sing) --> [le].`	`det(masc, sing) --> [der].`
`det(fem, sing) --> [la].`	`det(fem, sing) --> [die].`
`det(_, plur) --> [les].`	`det(neut, sing) --> [das].`
`noun(masc, sing) --> [garçon].`	`det(_, plur) --> [die].`
`noun(fem,sing) --> [serveuse].`	`noun(masc, _) --> ['Ober'].`
	`noun(fem,sing) --> ['Speise'].`

In addition to number and gender, German nouns are marked with four cases: nominative, dative, genitive, and accusative. The determiner case must agree with that of the adjective and the noun. To implement the case agreement, let us mark the noun phrase rule with an extra variable `Case`.

```
np(Gender, Number, Case) -->
  det(Gender, Number, Case),
  adj(Gender, Number, Case),
  noun(Gender, Number, Case).
```

Let us also write a small vocabulary:

```
det(masc, sing, nominative) --> [der].
det(masc, sing, dative) --> [dem].
det(masc, sing, genitive) --> [des].
det(masc, sing, accusative) --> [den].

adj(masc, sing, nominative) --> [freundliche].
adj(masc, sing, dative) --> [freundlichen].
adj(masc, sing, genitive) --> [freundlichen].
adj(masc, sing, accusative) --> [freundlichen].

noun(masc, _, Case) -->
  ['Ober'],
  {Case \= genitive}.
noun(masc, _, genitive) --> ['Obers'].
```

Querying `np` with the German vocabulary

```
?- np(G, N, C, L, []).
```

generates four noun phrases whose determiner, adjective, and noun agree in gender and case:

```
G = masc, N = sing, C = nominative,
  L = [der, freundliche, 'Ober'];
G = masc, N = sing, C = dative,
  L = [dem, freundlichen, 'Ober'];
G = masc, N = sing, C = genitive,
```

```
    L = [des, freundlichen, 'Obers'];
G = masc, N = sing, C = accusative,
    L = [den, freundlichen, 'Ober'];
No
```

So far, we have seen agreement within the noun phrase. It can also be applied to categorize verbs. Some verbs such as *sleep*, *appear*, or *rushed* are never followed by a noun phrase. These verbs are called intransitive (`iv`). Transitive verbs such as *bring* require a noun phrase after them: the object (`tv`). We can rewrite two verb phrase rules to mark transitivity:

```
vp --> verb(iv).
vp --> verb(tv), np.

verb(tv) --> [brought].
verb(iv) --> [rushed].
```

9.5.2 Obtaining the Syntactic Structure

We used variables to implement constraints. Variables can also return the parse tree of a sentence. The idea is to unify variables with the syntactic structure of a constituent while it is being parsed. To exemplify this, let us use a simplified version of our grammar:

```
s --> np, vp.
np --> det, noun.
vp --> verb, np.
```

The parse tree of

The waiter brought the meal

is reflected by the Prolog term

```
T = s(np(det(the), noun(waiter)),
      vp(verb(brought), np(det(the), noun(meal))))
```

To get this result, the idea is to attach an argument to all the symbols of rules, where each argument represents the partial parse tree of its corresponding symbol. Each right-hand-side symbol will have a variable that corresponds to the structure it matches, and the argument of the left-hand-side symbol will unify with the structure it has parsed. Each rule carries out a part of the tree construction when it is involved in the derivation. Let us consider the rule:

```
s --> np, vp.
```

We add two variables to `np` and `vp`, respectively `NP` and `VP`, that reflect the partial structure they map. When the whole sentence has been parsed, `NP` and `VP` should be

```
NP = np(det(the), noun(waiter))
```

and

```
VP = vp(verb(brought), np(det(the), noun(meal)))
```

When `NP` and `VP` are unified, `s` combines them into a term to form the final structure. This term is `s(NP, VP)`. We obtain the construction of the parse tree by changing rule

```
s --> np, vp
```

into

```
s(s(NP, VP)) --> np(NP), vp(VP).
```

The rest of the rules are modified in the same way:

```
np(np(D, N)) --> det(D), noun(N).
vp(vp(V, NP)) --> verb(V), np(NP).

det(det(the)) --> [the].
det(det(a)) --> [a].

noun(noun(waiter)) --> [waiter].
noun(noun(meal)) --> [meal].
noun(noun(table)) --> [table].
noun(noun(tray)) --> [tray].

verb(verb(bring)) --> [brought].
```

The query:

```
?- s(Structure, L, []).
```

generates all the sentences together with their syntactic structure:

```
Structure = s(np(det(the), noun(waiter)),
   vp(verb(brought), np(det(the), noun(waiter)))),
L = [the, waiter, brought, the, waiter] ;

Structure = s(np(det(the), noun(waiter)),
   vp(verb(brought), np(det(the), noun(meal)))),
L = [the, waiter, brought, the, meal] ;

Structure = s(np(det(the), noun(waiter)),
   vp(verb(brought), np(det(the), noun(table)))),
L = [the, waiter, brought, the, table]
...
```

9.6 Application: Tokenizing Texts Using DCG Rules

We can use DCG rules for many applications other than sentence parsing, which we exemplify here with a tokenization grammar.

9.6.1 Word Breaking

The first part of a tokenizer takes a character list as an input and breaks it into tokens. Let us implement this with a DCG grammar. We start with rules describing a sequence of tokens (`tokens`) separated by blanks. Blank characters (`blank`) are white spaces, carriage returns, tabulations, or control codes. A token (`token`) is a sequence of alphanumeric characters (`alphanumerics`) or another symbol (`other`). Finally, alphanumerics are digits, uppercase letters, lowercase letters, or accented letters:

```
tokens(Tokens) --> blank, {!}, tokens(Tokens).
tokens([FirstT | Tokens]) -->
  token(FirstT), {!}, tokens(Tokens).
tokens([]) --> [].

% A blank is a white space or a control character
blank --> [B], {B =< 32, !}.

% A token is a sequence of alphanumeric characters
% or another symbol

token(Word) --> alphanumerics(List), {name(Word,
List), !}.
token(Symbol) --> other(CSymbol), {name(Symbol,
[CSymbol]), !}.

% A sequence of alphanumerics is an alphanumeric
% character followed by other alphanumerics
% or a single alphanumeric character.

alphanumerics([L | LS]) -->
  alphanumeric(L), alphanumerics(LS).
alphanumerics([L]) --> alphanumeric(L).

% Here comes the definition of alphanumeric
characters:
% digits, uppercase letters without accent, lowercase
% letters without accent, and accented characters.
```

```
% Here we only consider letters common in French,
German,
% and Swedish

% digits
alphanumeric(D) --> [D], { 48 =< D, D =< 57, !}.

% uppercase letters without accent
alphanumeric(L) --> [L], {65 =< L, L =< 90, !}.

% lowercase letters without accent
alphanumeric(L) --> [L], {97 =< L, L =< 122, !}.

% accented characters
alphanumeric(L) --> [L], {name(A, [L]), accented(A),
!}.

accented(L) :-
  member(L,
    ['à', 'â', 'ä', 'å', 'æ', 'ç', 'é', 'è', 'ê', 'ë',
    'î', 'ï', 'ô', 'ö', 'œ', 'ù', 'û', 'ü', 'ÿ',
    'À', 'Â', 'Ä', 'Å', 'Æ', 'Ç', 'É', 'È', 'Ê', 'Ë',
    'Î', 'Ï', 'Ô', 'Ö', 'Œ', 'Ù', 'Û', 'Ü', 'Ÿ']).

% All other symbols come here
other(Symbol) --> [Symbol], {!}.
```

Before applying the `tokens` rules, we need to read the file to tokenize and to build a character list. We do it with the `read_file/2` predicate. We launch the complete word-breaking program with

```
?- read_file(myFile, CharList),
   tokens(TokenList, CharList, []).
```

9.6.2 Recognition of Sentence Boundaries

The second role of tokenization is to delimit sentences. The corresponding grammar takes the token list as an input. The sentence list (`sentences`) is a list of words making a sentence (`words_of_a_sentence`) followed by the rest of the sentences. The last sentence can be a punctuated sentence or a string of words with no final punctuation (`words_without_punctuation`). We define a sentence as tokens terminated by an end punctuation: a period, a colon, a semicolon, an exclamation point, or a question mark.

```
sentences([S | RS]) --> words_of_a_sentence(S),
sentences(RS).
% The last sentence (punctuated)
sentences([S]) --> words_of_a_sentence(S).
% Last sentence (no final punctuation)
sentences([S]) --> words_without_punctuation(S).

words_of_a_sentence([P]) --> end_punctuation(P).
words_of_a_sentence([W | RS]) -->
  word(W),
  words_of_a_sentence(RS).

words_without_punctuation([W | RS]) -->
  word(W),
  words_without_punctuation(RS).
words_without_punctuation([W]) --> [W].

word(W) --> [W].

end_punctuation(P) --> [P], {end_punctuation(P), !}.

end_punctuation(P) :- member(P, ['.', ';', ':', '?',
'!']).
```

We launch the whole tokenization program with

```
?- read_file(myFile, CharacterList),
   tokens(TokenList, CharacterList, []),
   sentences(SentenceList, TokenList, []).
```

9.7 Semantic Representation

9.7.1 λ-Calculus

One of the goals of semantics is to map sentences onto logical forms. In many applications, this is a convenient way to represent meaning. It is also a preliminary step to further processing such as determining whether the meaning of a sentence is true or not.

In some cases, the logical form can be obtained simultaneously while parsing. This technique is based on the principle of compositionality, which states that it is possible to compose the meaning of a sentence from the meaning of its parts. We shall explain this with the sentence

Bill is a waiter

and its corresponding logical form

```
waiter('Bill').
```

If *Pierre* replaces *Bill* as the waiter, the semantic representation of the sentence is

```
waiter('Pierre').
```

This means that the constituent *is a waiter* retains the same meaning independently of the value of the subject. It acts as a property or a function that is applied to other constituents. This is the idea of compositional analysis: combine independent constituents to build the logical form of the sentence.

The λ-calculus (Church 1941) is a mathematical device that enables us to represent intermediate constituents and to compose them gracefully. It is a widely used tool in compositional semantics. The λ-calculus maps constituents onto abstract properties or functions, called λ-expressions. Using a λ-expression, the property *is a waiter* is represented as

$$\lambda x.waiter(x)$$

where λ is a right-associative operator. The transformation of a phrase into a property is called a λ-abstraction. The reverse operation is called a β-reduction. It is carried out by applying the property to a value and is denoted

$$\lambda x.waiter(x)(Bill)$$

which yields

$$waiter(Bill)$$

Since there is no λ character on most computer keyboards, the infix operator `^` classically replaces it in Prolog programs. So $\lambda x.waiter(x)$ is denoted `X^waiter(X)`. λ-expressions are also valid for adjectives, and *is fast* is mapped onto `X^fast(X)`. A combination of nouns and adjectives, such as *is a fast waiter*, is represented as: `X^(fast(X), waiter(X))`.

While compositionality is an elegant tool, there are also many sentences where it does not apply. *Kick* is a frequently cited example. It shows compositional properties in *kick the ball* or *kick the box*. A counter example is the idiom *kick the bucket*, which means to die, and where *kick* is not analyzable alone.

9.7.2 Embedding λ-Expressions into DCG Rules

It is possible to use DCG rules to carry out a compositional analysis. The idea is to embed λ-expressions into the rules. Each rule features a λ-expression corresponding

to the constituent it can parse. Parsing maps λ-expressions onto constituents rule-by-rule and builds the semantic representation of the sentence incrementally.

The sentence we have considered applies the property of being a waiter to a name: *Pierre* or *Bill*. In this sentence, the verb *is*, as other verbs of being, only links a name to the predicate `waiter(X)`. So the constituent *is a waiter* is roughly equivalent to *waiter*. Then, the semantic representation of common nouns or adjectives is that of a property: $\lambda x.waiter(x)$. Nouns incorporate their semantic representation as an argument in DCG rules, as in:

```
noun(X^waiter(X)) --> [waiter].
```

As we saw, verbs of being have no real semantic content. If we only consider these verbs, verb phrase rules only pass the semantics of the complement to the sentence. Therefore, the semantics of the verb phrase is simply that of its noun phrase:

```
vp(Semantics) --> verb, np(Semantics).
```

The `Semantics` variable is unified to `X^waiter(X)`, where `X` is to represent the sentence's subject. Let us write this in the sentence rule that carries out the β-reduction

```
s(Predicate) --> np(Subject),
vp(Subject^Predicate).
```

The semantic representation of a name is just this name:

```
np('Bill') --> ['Bill'].
np('Mark') --> ['Mark'].
```

We complement the grammar with an approximation: we consider that determiners have no meaning. It is obviously untrue. We do it on purpose to keep the program simple. We will get back to this later:

```
np(X) --> det, noun(X).
det --> [a].
verb --> [is].
```

Once the grammar is complete, querying it with a sentence results in a logical form:

```
?- s(S, ['Mark', is, a, waiter], []).
S = waiter('Mark').
```

The reverse operation generates a sentence from the logical form:

```
?- s(waiter('Bill'), L, []).
L = ['Bill, is, a, waiter].
```

9.7.3 Semantic Composition of Verbs

We saw that verbs of being played no role in the representation of a sentence. On the contrary, other types of verbs, as in

Bill rushed
Mr. Schmidt called Bill

are the core of the sentence representation. They correspond to the principal functor of the logical form:

```
rushed('Bill')
called('Mr. Schmidt', 'Bill')
```

Their representation is mapped onto a λ-expression that requires as many arguments as there are nouns involved in the logical form. *Rushed* in the sentence *Bill rushed* is intransitive. It has a subject and no object. It is represented as

```
X^rushed(X)
```

where `X` stands for the subject. This formula means that to be complete the sentence must supply `rushed(X)` with `X = 'Bill'` so that it reduces to `rushed('Bill')`.

Called in the sentence *Mr. Schmidt called Bill* is transitive: it has a subject and an object. We represent it as

```
Y^X^called(X, Y)
```

where `X` and `Y` stand, respectively, for the subject and the object. This expression means that it is complete when `X` and `Y` are reduced.

Let us now examine how the parsing process builds the logical form. When the parser considers the verb phrase

called Bill

it supplies an object to the verb's λ-expression. The λ-expression reduces to one argument, $\lambda x.called(x, Bill)$, which is represented in Prolog by

```
X^called(X, 'Bill')
```

When the subject is supplied, the expression reduces to

```
called('Mr. Schmidt', 'Bill').
```

Figure 9.8 shows graphically the composition.

Let us now write a complete grammar accepting both sentences. We add a variable or a constant to the left-hand-side symbol of each rule to represent the constituent's or the word's semantics. The verb's semantics is a λ-expression as described previously, and `np`'s value is a proper noun. The semantic representation is built compositionally – at each step of the constituent parsing – by unifying the argument of the left-hand-side symbol.

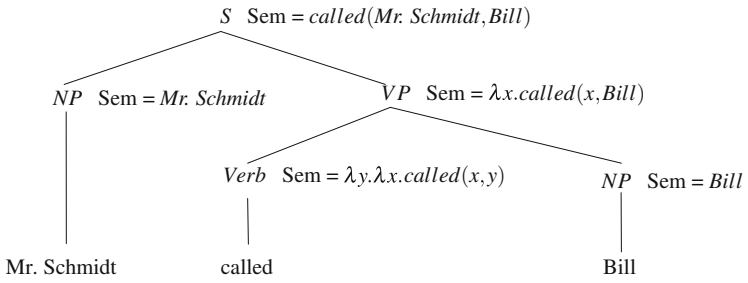


Fig. 9.8 Parse tree with a semantic composition

```
s(Semantics) --> np(Subject),
vp(Subject^Semantics).
vp(Subject^Semantics) --> verb(Subject^Semantics).
vp(Subject^Semantics) -->
  verb(Object^Subject^Semantics), np(Object).
np('Bill') --> ['Bill'].
np('Mr. Schmidt') --> ['Mr. Schmidt'].

verb(X^rushed(X)) --> [rushed].
verb(Y^X^called(X, Y)) --> [called].

?- s(Semantics, ['Mr. Schmidt', called, 'Bill'], []).
Semantics = called('Mr. Schmidt', 'Bill')
```

In this paragraph, proper nouns were the only noun phrases we considered. We have set aside common nouns and determiners to simplify the presentation. In addition, prepositions and prepositional phrases can also be mapped onto λ-expressions in the same way as verbs and verb phrases. We will examine the rest of the semantics in more detail in Chap. 14.

9.8 An Application of Phrase-Structure Grammars and a Worked Example

As we saw in Chap. 1, the Microsoft Persona agent uses a phrase-structure grammar module to parse sentences and gets a logical form from them. Ball et al. (1997) give an example of order:

> I'd like to hear something composed by Mozart.

that Persona transforms in the logical form:

```
like1 (+Modal +Past +Futr)
   Dsub: i1 (+Pers1 +Sing)
```

```
Dobj: hear1
  Dsub: i1
  Dobj: something1 (+Indef +Exis +Pers3 +Sing)
     Prop: compose1
        Dsub: mozart1 (+Sing)
        Dobj: something1
```

Although Persona uses a different method (Jensen et al. 1993), a small set of DCG rules can parse this sentence and derive a logical form using compositional techniques. To write the grammar, let us simplify the order and proceed incrementally. The core of the sentence means that the user would like something or some Mozart. It is easy to write a grammar to parse sentences such as:

I would like something
I would like some Mozart

The sentence and the noun phrase rules are close to those we saw earlier:

```
s(Sem) --> np(Sub), vp(Sub^Sem).
```

In anticipation of a possible left-recursion, we use an auxiliary `npx` symbol to describe a nonrecursive noun phrase:

```
npx(SemNP) --> pro(SemNP).
npx(SemNP) --> noun(SemNP).
npx(SemNP) --> det, noun(SemNP).

np(SemNP) --> npx(SemNP).
```

The verb phrase is slightly different from those of the previous sections because it contains an auxiliary verb. A possible expansion would consist of the auxiliary and a recursive verb phrase:

```
vp --> aux, vp.
```

Although some constituent grammars are written this way, the treatment of auxiliary *would* is disputable. In some languages – notably in Romance languages – the conditional auxiliary is rendered by the inflection of the main verb, as in French: *j'aimerais* 'I would like'. A better modeling of the verb phrase uses a verb group that corresponds either to a single verb or to a sequence, including an auxiliary to the left and the main verb, here

```
verb_group(SemVG) --> aux(SemAux), verb(SemVG).
verb_group(SemVG) --> verb(SemVG).

vp(SemVP) --> verb_group(SemVP).
vp(SemVP) --> verb_group(Obj^SemVP), np(Obj).
```

The vocabulary is also similar to what we saw previously:

```
verb(Obj^Sub^like(Sub, Obj)) --> [like].
```

```
verb(Obj^Sub^hear(Sub, Obj)) --> [hear].

aux(would) --> [would].

pro('I') --> ['I'].
pro(something) --> [something].

noun(N) --> proper_noun(N).

proper_noun('Mozart') --> ['Mozart'].

det --> [some].
```

This grammar answers queries such as:

```
?- s(Sem, ['I', would, like, some, 'Mozart'], []).
Sem = like('I', 'Mozart')
```

Now let us take a step further toward the original order, and let us add the infinitive verb phrase *to hear*:

I would like to hear something
I would like to hear some Mozart

The infinitive phrase has a structure similar to that of a finite verb phrase except that it is preceded by the infinitive marker *to*:

```
vp_inf(SemVP) --> [to], vp(SemVP).
```

We must add a new verb phrase rule to the grammar to account for it. Its object is the subordinate infinitive phrase:

```
vp(SemVP) --> verb_group(Obj^SemVP), vp_inf(Obj).
```

The new grammar accepts queries such as:

```
?- s(Sem, ['I', would, like, to, hear, some,
'Mozart'], []).
Sem = like('I', X^hear(X, 'Mozart'))
```

In the resulting logical form, the subject of *hear* is not reduced. In fact, this is because it is not explicitly indicated in the sentence. This corresponds to an anaphora within the sentence – an intrasentential anaphora – where both verbs *like* and *hear* implicitly share the same subject.

To solve the anaphora and to understand how Prolog composes the logical forms, instead of using the variable `Obj`, let us exhibit all the variables of the λ-expressions at the verb phrase level. The nonreduced λ-expression for *hear* is

```
ObjectHear^SubjectHear^hear(SubjectHear, ObjectHear).
```

When the infinitive verb phrase has been parsed, the `ObjectHear` is reduced and the remaining expression is

```
SubjectHear^hear(SubjectHear, 'Mozart').
```

The original λ-expression for *like* is

```
ObjectLike^SubjectLike^like(SubjectLike, ObjectLike)
```

where `ObjectLike` unifies with the λ-expression representing *hear*. Since both subjects are identical, λ-expressions can be rewritten so that they share a same variable in `Subject^SemInf` for *hear* and `SemInf^Subject^SemVP` for *like*. The verb phrase is then:

```
vp(Subject^SemVP) -->
  verb_group(SemInf^Subject^SemVP),
  vp_inf(Subject^SemInf).
```

and the new grammar now solves the anaphora:

```
?- s(Sem, ['I', would, like, to, hear, some,
'Mozart'], []).
Sem = like('I', hear('I', 'Mozart'))
```

Let us conclude with the complete order, where the track the user requests is *something composed by Mozart*. This is a noun phrase, which has a passive verb phrase after the main noun. We model it as:

```
np(SemNP) --> npx(SemVP^SemNP), vp_passive(SemVP).
```

We also need a model of the passive verb phrase:

```
vp_passive(SemVP) --> verb(Sub^SemVP), [by], np(Sub).
```

and of the verb:

```
verb(Sub^Obj^compose(Sub, Obj)) --> [composed].
```

Finally, we need to modify the pronoun *something* so that it features a property:

```
pro(Modifier^something(Modifier)) --> [something].
```

Parsing the order with the grammar yields the logical form:

```
?- s(Sem, ['I', would, like, to, hear, something,
composed, by, 'Mozart'], []).
Sem = like('I', hear('I', X^something(compose
('Mozart', X))))
```

which leaves variable `X` uninstantiated.[1] A postprocessor would then be necessary to associate `X` with *something* and reduce it.

[1] Prolog probably names it `_Gxxx` using an internal numbering scheme.

9.9 Further Reading

Colmerauer (1970, 1978) created Prolog to write language processing applications and, more specifically, parsers. Pereira and Warren (1980) designed the Definite Clause Grammar notation, although it is merely a variation on the Prolog syntax. Most Prolog environments now include a compiler that is based on the Warren Abstract Machine (WAM) (Warren 1983). This WAM has made Prolog's execution very efficient.

Textbooks on Prolog and natural language processing delve mostly into syntax and semantics. Pereira and Shieber (1987) provide a good description of phrase-structure grammars, parsing, and formal semantics. Other valuable books include Gazdar and Mellish (1989), Covington (1994b), and Gal et al. (1989).

SRI's Core Language Engine (Alshawi 1992) is an example of a comprehensive development environment based on Prolog. It is probably the most accomplished industrial system in the domain of syntax and formal semantics. Using it, Agnäs et al. (1994) built the Spoken Language Translator (SLT) to translate spoken English to spoken Swedish in the area of airplane reservations. The SLT has been adapted to other language pairs.

Exercises

9.1. Translate the sentences of Sect. 9.2.1 into French or German and write the DCG grammar accepting them.

9.2. Underline constituents of the sentence *The nice hedgehog ate the worm in its nest.*

9.3. Write a grammar accepting the sentence *The nice hedgehog ate the worm in its nest*. Draw the corresponding tree. Do the same in French or German.

9.4. The previous grammar contains a left-recursive rule. Transform it as indicated in this chapter.

9.5. Give a sentence generated by the previous grammar that is not semantically correct.

9.6. Verbs of being can be followed by adjective phrases or noun phrases. Imagine a new constituent category, `adjp`, describing adjective phrases. Write the corresponding rules. Write rules accepting the sentences *the waiter is tall*, *the waiter is very tall*, and *Bill is a waiter*.

9.7. How does Prolog translate the rule `lex --> [in, front]`?

9.8. How does Prolog translate the rule `lex --> [in], {prolog_code}, [front]`?

9.9. Write the `expand_term/2` predicate that converts DCG rules into Prolog clauses.

9.10. Write a grammar accepting the sentence *The nice hedgehog ate the worm in its nest* with variables building the parse tree.

9.11. Replace all nouns of the previous sentence by personal pronouns, and write the grammar.

9.12. Translate the sentence in Exercise 9.10 into French or German, and add variables to the rules to check number, gender, and case agreement.

9.13. Calculate the β-reductions of expressions $\lambda x.f(x)(y)$ and $\lambda x.f(x)(\lambda y.f(y))$.

9.14. Write a grammar that accepts the noun phrase *the nice hedgehog* and that builds a syntactic representation of it.

9.15. Persona's parser accepts orders like *Play before you accuse me*. Draw the corresponding logical form. Write grammar rules that parse the order *Play a song* and that build a logical form from it.

Chapter 10
Partial Parsing

10.1 Is Syntax Necessary?

The description of language in terms of layers – words, parts of speech, and syntax – could suggest that a parse tree is a necessary step to obtain the semantic representation of a sentence. Yet, many industrial applications do not rely on syntax as we presented it before. The reason is that a syntactic parser can be expensive in terms of resources and sometimes it is not worth the cost.

Some applications need only to detect key words, as in some telephone speech servers. There, the speech recognition module spots meaningful words and sets the others aside. It enables the system to deal with the noisy environment or the fragmented nature of speech by telephone. Other applications rely on the detection of word groups such as noun phrases. Although sentences are not fully parsed, the result is sufficient to make use of it. Information retrieval and extraction are typical applications relying on group detection techniques.

In this chapter, we will examine a collection of techniques to extract incomplete syntactic representations. These techniques are generally referred to as partial or shallow parsing and operate on groups of words, often called **chunks**. Some of them just carry out the detection of key words or specific word patterns. Others use phrase-structure rules describing groups such as noun groups or verb groups. Finally, some techniques are an extension of part-of-speech tagging and resort to similar methods.

10.2 Word Spotting and Template Matching

10.2.1 ELIZA

A first shallow technique consists in matching predefined templates. It appeared in the popular ELIZA program that mimics a dialogue between a psychotherapist

P.M. Nugues, *Language Processing with Perl and Prolog*, Cognitive Technologies,
DOI 10.1007/978-3-642-41464-0_10, © Springer-Verlag Berlin Heidelberg 2014

Table 10.1 Some ELIZA templates

User	Psychotherapist
...I like X...	*Why do you like X?*
...I am X...	*How long have you been X?*
...father...	*Tell me more about your father*

and his/her patient (Weizenbaum 1966). In fact, ELIZA understands nothing. She "spots" a handful of words or patterns such as *yes*, *no*, *why*, *I'm afraid of X*, *I like X*, etc., where *X* is a name or any group of words. When a template matches the user's sentence, ELIZA has a set of ready-made answers or questions mapped onto it. When no template matches, ELIZA tries to guess whether the sentence is a declaration, a negation, or an interrogation, and has repartees like *in what way*, *can you think of a specific example*, *go on*, etc. It enables the machine to follow the conversation with a semblance of realism. Table 10.1 shows some user/psychotherapist templates.

ELIZA's dialogue pays specific attention to words like *mother* and *father*. Whenever one of these words occurs, ELIZA asks for more details. We remind the reader that this program was created when Freudian theories were still influential. Although the approach is now considered simplistic, at best, the psychoanalytical settings secured ELIZA mainstream popularity.

10.2.2 Word Spotting in Prolog

A word spotting program can easily be written using DCG rules. Utterances are modeled as phrase-structure rules consisting of a beginning, the word or pattern to search, and an end. The translation into a DCG rule is straightforward:

```
utterance(U) --> beginning(B), [the_word], end(E).
```

Each predicate has a variable that unifies with the part of the utterance it represents. Variables `B` and `E` unify respectively with the beginning and the end of the utterance. The variable `U` is used to build the system answer as in the templates in Table 10.1.

Prolog translates the DCG rules into clauses when they are consulted. It adds two arguments to each predicate, and the previous rule expands into:

```
utterance(U, L1, L) :-
  beginning(B, L1, L2),
  c(L2, the_word, L3),
  end(E, L3, L).
```

We saw in Chap. 9 that each predicate in the rule covers a word sequence, and that it corresponds to the difference of the two new arguments: `L1` minus `L` corresponds to `utterance`; `L1` minus `L2` corresponds to `beginning`; `L3`

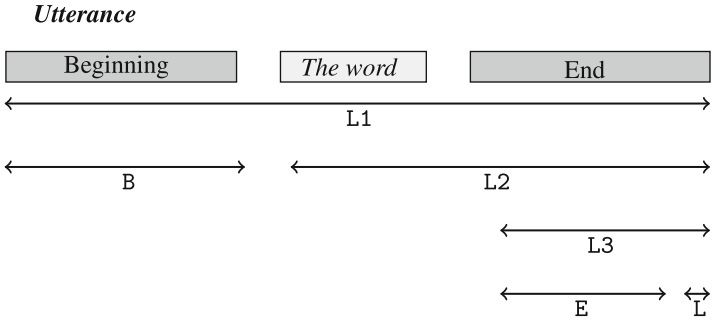


Fig. 10.1 The composition of `utterance`

minus `L` corresponds to `end`. Figure 10.1 shows the composition of the utterance with respect to the new lists.

To match `B` and `E`, the trick is to define `beginning/3` and `end/3` as `append`-like predicates:

```
beginning(X, Y, Z) :- append(X, Z, Y).
end(X, Y, Z) :- append(X, Z, Y).
```

ELIZA is then a loop that reads the user input, tries to find a matching utterance, and answers with the corresponding template. It stops when the user writes the word *bye*. The next program is a simplified version of ELIZA. It matches the user/psychoanalyst pairs in Table 10.1.

```
%% A simplified version of ELIZA in Prolog
%%

% The main loop reads the input and calls process/1
% It stops when the input is the word bye.

eliza :-
  write('Hello, I am ELIZA. How can I help you?'), nl,
  repeat,
  write('> '),
  tokenize(In),
  process(In).

% process/1 accepts the user's utterance. It either
% terminates or tries to find a template matching the
% user's utterance

process([bye | _]) :-
  write('ELIZA: bye'), nl, !.
process(In) :-
```

```
    utterance(Out, In, []),
    !,
    write('ELIZA: '), write_answer(Out),
    fail.

% utterance is a set of DCG rules describing
templates

utterance(['Why', aren, ''', t, you | Y]) -->
    ['I', am, not], end(Y).
utterance(['How', long, have, you, been | Y]) -->
    ['I', am], end(Y).
utterance(['Why', do, you, like | Y]) -->
    ['I', like], end(Y).
utterance(['Do', you, often, think, of | Y]) -->
    beginning(_), ['I', remember], end(Y).
utterance(['Please', tell, me, more, about, your,
X, .]) -->
    beginning(_), [X], end(_), {important(X)}.
utterance(['Why', are, so, negative, '?']) -->
    [no].
utterance(['Tell', me, more, .]) -->
    [_].
utterance(['Please', go, on, .]) -->
    beginning(_).

% The Prolog clauses beginning/3 and end/3
% link the pieces of the utterance

beginning(X, Y, Z) :- append(X, Z, Y).
end(X, Y, Z) :- append(X, Z, Y).

% The Freudian facts
important(father).
important(mother).
important(brother).
important(sister).

% write_answer/1 prints the content of the list
% containing Eliza's answer.

write_answer([Head | Tail]) :-
    write(Head), write(' '),
    write_answer(Tail).
write_answer([]) :- nl.
```

The next lines are a sample dialogue with ELIZA:

```
?- eliza.
Hello, I am ELIZA. How can I help you?
> yes
ELIZA: Tell me more .
> I like apples
ELIZA: Why do you like apples
>I went out with my father
ELIZA: Please tell me more about your father .
> no
ELIZA: Why are so negative ?
> I am cold
ELIZA: How long have you been cold
> bye
ELIZA: bye

true
?-
```

10.3 Named Entities and Multiwords

10.3.1 Named Entities

While ELIZA has no real application, the techniques we used in it can serve to detect specific patterns in texts or in speech. The recognition of named entities is an example of it. A named entity is an entity whose name in a text refers to a unique person, place, object, etc., as *William Shakespeare* or *Stratford-upon-Avon* in the phrase:

> William Shakespeare was born and brought up in Stratford-upon-Avon.

This opposes to phrases referring to entities with no name as *this person* or *a street* in:

> meeting with this person in a street nearby,

and reflects overall the distinction between common and proper nouns; see Fig. 10.2.

Names of people or organizations are frequent in the press and the media, where they surge and often disappear quickly. The first step to recognize them is to identify the phrases corresponding to names of persons, organizations, or locations (Table 10.2). Such phrases can be a single proper noun or a group of words.

Named entity recognition also commonly extends to temporal expressions describing times and dates, and numerical and quantity expressions, even if these are not entities.

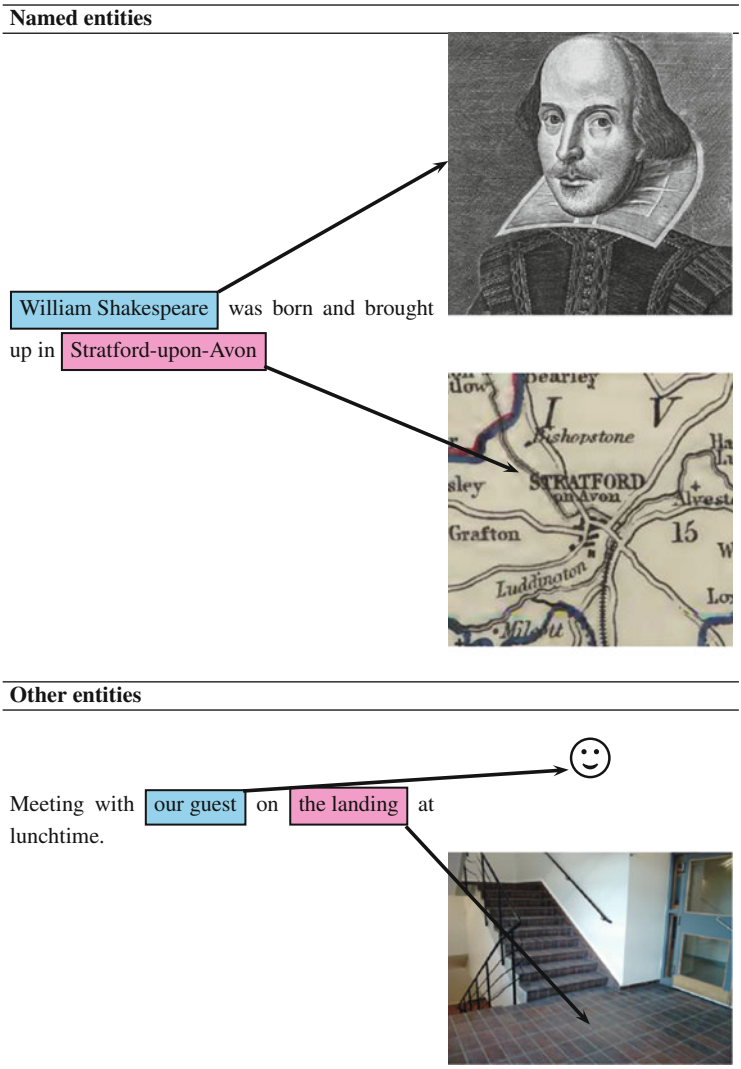


Fig. 10.2 Named entities: entities that we can identify by their names (Portrait: credits Wikipedia. Map: Samuel Lewis, *Atlas to the topographical dictionaries of England and Wales*, 1848, credits: archive.org)

10.3.2 Multiwords

Although conceptually different, named entities are often associated with multiwords – or multiword expressions (MWE) – i.e., sequences of two or more words that act as a single lexical unit. Multiwords include complex prepositions, adverbs, conjunctions, or phrasal verbs where each of the words taken separately cannot be

Table 10.2 Named entities in English and French

Type	English	French
Company names	*British Gas plc.*	*Compagnie générale d'électricité SA*
Person names	*Mr. Smith*	*M. Dupont*
Titles	*The President of the United States*	*Le président de la République*

Table 10.3 Multiwords in English and French

Type	English	French
Prepositions	*to the left-hand side*	*à gauche de*
Adverbs	*because of*	*à cause de*
Conjunctions		
Verbs	*give up*	*faire part* 'inform'
	go off	*rendre visite* 'pay a visit'

clearly understood (Table 10.3). The concept of what a multiword is may seem intuitive, but there are many tricky cases. In addition, people do not always agree on their exact definition.

The identification of named entities and multiwords uses roughly the same techniques.

10.3.3 A Standard Named Entity Annotation

In the 1990s, The US Department of Defense organized series of competitions to measure the performance of commercial and academic systems on multiword detection. It called them the Message Understanding Conferences (MUC). To help benchmarking the various systems, MUC-6 and MUC-7 defined an annotation scheme that was shared by all the participants. This annotation has subsequently been adopted by commercial applications. The definition of the named entity annotation can be read from the MUC-7 web page.[1]

The MUC annotation restricts the annotation to information useful for its main funding source: the US military. It considers named entities (persons, organizations, locations), time expressions, and quantities. The annotation scheme defines a corresponding XML element for each of these three classes: `<ENAMEX>`, `<TIMEX>`, and `<NUMEX>` (Chinchor 1997), with which it brackets the relevant phrases in a text. The phrases can consist of one, two, or more words.

The `<ENAMEX>` element identifies proper nouns and uses a `TYPE` attribute with three values to categorize them: `ORGANIZATION`, `PERSON`, and `LOCATION`, as in

```
the <ENAMEX TYPE="PERSON">Clinton</ENAMEX> government
<ENAMEX TYPE="ORGANIZATION">Bridgestone Sports Co.</ENAMEX>
<ENAMEX TYPE="ORGANIZATION">European Community</ENAMEX>
```

[1] http://www.itl.nist.gov/iaui/894.02/related_projects/muc/proceedings/muc_7_toc.html

```
<ENAMEX TYPE="ORGANIZATION">University of California</ENAMEX> in
<ENAMEX TYPE="LOCATION">Los Angeles</ENAMEX>
```

The `<TIMEX>` element identifies time expressions and uses a `TYPE` attribute to distinguish between `DATE` and `TIME`, as in

```
<TIMEX TYPE="TIME">twelve o'clock noon</TIMEX>
<TIMEX TYPE="TIME">5 p.m. EST</TIMEX>
<TIMEX TYPE="DATE">January 1990</TIMEX>
```

The `<NUMEX>` element is used to bracket quantities. It has also a `TYPE` attribute to categorize `MONEY` and `PERCENT`, as in

```
<NUMEX TYPE="MONEY">20 million New Pesos</NUMEX>
<NUMEX TYPE="MONEY">$42.1 million</NUMEX>
<NUMEX TYPE="MONEY">million-dollar</NUMEX> conferences
<NUMEX TYPE="PERCENT">15 pct</NUMEX>
```

10.4 Detecting Named Entities with Rules

The detection of named entities and multiwords with rules is an extension of word spotting. Just as for word spotting, we represent them using DCG rules. We use variables and Prolog code to extract them from the word stream and annotate them.

Compounded prepositions, conjunctions, and phrasal verbs are often listed in dictionaries and can be encoded as Prolog constants. Named entities raise more problems. Their identification generally requires specialized dictionaries of surnames, companies, countries, and trademarks. Some of these dictionaries, called **gazetteers**, are available on the Internet. They are built from the compilation of lexical sources such as economic and legal newspapers, directories, or internet web sites.

The extraction of named entities also relies on hints that vary according to the type of entity to detect. Locations may include words such as *Ocean*, *Range*, *River*, etc. Legal denominations will be followed by acronyms such as *Ltd.*, *Corp.*, *SA*, and *GmbH*. Persons might be preceded by titles such as *Mr.*, *Mme*, *Herr*, *Dr.*, by a surname, or have a capitalized initial. Currency phrases will include a sign such as €, $, £, etc., and a number. Such techniques can be applied to any measuring expression: length, time, etc.

Let us write rules to detect the phrasal verb *give up*, the French title *M. XXXX*, such as *M. Dupont*, and the European money worth *XXXX euros*, such as *200 euros*. As a result, the detector appends the multiword parts using an underscore character: `give_up`, or builds a list with surrounding XML tags `[<ENAMEX>, 'M.', 'Dupont', </ENAMEX>]`, and `[<NUMEX>, 200, euros, </NUMEX>]`. The corresponding rules are:

```
multiword(give_up) --> [give, up].
multiword(['<ENAMEX>', 'M.', Name, '</ENAMEX>']) -->
```

Table 10.4 Longer matches are preferred

	English	French
Competing multiwords	*in front of*	*en face de*
	in front	*en face*
Examples	*the car in front*	*la voiture en face*
	in front of me	*en face de moi*

```
    ['M.'], [Name],
    {
      name(Name, [Initial | _]),
      Initial >= 65, % must be an upper-case letter
      Initial =< 90
    }.
multiword(['<NUMEX>', Value, euros, '</NUMEX>']) -->
    [Value], [euros],
    {
      number(Value)
    }.
```

10.4.1 The Longest Match

Among the set of multiwords we want to detect, some may have a common suffix, as for the phrases *in front* and *in front of*. This corresponds to the rules:

```
multiword(in_front) --> [in, front].
multiword(in_front_of) --> [in, front, of].
```

With the sentence:

The car in front of the house

rules as they are ordered above yield two solutions. The first multiword to be matched is *in front*, and if Prolog backtracks, it will find *in front of*. A backtracking strategy is not acceptable in most cases. What we generally want is the longest possible match (Table 10.4).

Prolog interpreters consider rules sequentially and downwards (from the beginning to the end). We implement the longest match by ordering the DCG rules properly. When several multiwords compete, i.e., have the same beginning, the longest one must be searched first, as in the sequence:

```
multiword(in_front_of) --> [in, front, of].
multiword(in_front) --> [in, front].
```

10.4.2 Running the Program

Now we will write a rule to embed the multiword description. If the word stream contains a multiword, it should be modeled as a beginning, the multiword, and an end, as in ELIZA. Its transcription into a DCG rule is straightforward:

```
word_stream_multiword(Beginning, Multiword, End) -->
  beginning(Beginning),
  multiword(Multiword),
  end(End).
```

Extracting the list of multiwords means that the whole word stream must be matched against the rule set. The multiword detector scans the word stream from the beginning, and once a multiword has been found, it starts again with the remaining words.

`multiword_detector/2` is a Prolog predicate. It takes the word stream `In` as the input and the multiword list `Out` as the output. It searches a multiword within the word stream using the `word_stream_multiword` DCG rule.

Each `word_stream_multiword` rule is translated into a Prolog predicate when consulted and two new variables are added. Thus, `word_stream_multiword` is of arity 5 in the `multiword_detector` rule. The two last variables are unified respectively to the input list and to the empty list.

When `word_stream_multiword` reaches a multiword, `Beginning` is unified with the beginning of the word stream and `End` with the rest. The program is called recursively with `End` as the new input value.

```
multiword_detector(In, [Multiword | Out]) :-
  word_stream_multiword(Beginning, Multiword, End,
  In, []),
  multiword_detector(End, Out).
  multiword_detector(_, []).
```

Using the detector with the sentence *M. Dupont was given 500 euros in front of the casino* results in `['<ENAMEX>', 'M.', 'Dupont', '</ENAMEX>']`, `['<NUMEX>', 500, euros, '</NUMEX>']`, and `in_front_of`:

```
?- multiword_detector(['M.', 'Dupont', was, given,
500, euros, in, front, of, the, casino], Out).
Out = [[<ENAMEX>, M., Dupont, </ENAMEX>],
[<NUMEX>, 500, euros, </NUMEX>], in_front_of]
```

The result is a list containing sublists. The `flatten/2` predicate can replace recursively all the sublists by their elements and transform them into a flat list.

```
?- flatten([['<ENAMEX>', 'M. ', Dupont, '</ENAMEX>'],
['<NUMEX>', 500, 'DM', '</NUMEX>'], in_front_of], Out).
Out = [<ENAMEX>, M., Dupont, </ENAMEX>, <NUMEX>,
500, DM, </NUMEX>, in_front_of]
```

The multiword detector can be modified to output the whole stream. That is, the multiwords are tagged and other words remain unchanged. In this program, `Beginning` is appended to the multiword `Multiword` that has been detected to form the `Head` of the word stream. The `Head` and the result of the recursive call `Rest` form the `Output`. We must not forget the `End` in the termination fact.

```
multiword_detector(In, Out) :-
  word_stream_multiword(Beginning, Multiword, End, In,
  []), !,
  multiword_detector(End, Rest),
  append(Beginning, [Multiword], Head),
  append(Head, Rest, Out).
  multiword_detector(End, End).
```

Let us now execute a query with this new detector with `flatten/2`:

```
?- multiword_detector(['M.', 'Dupont', was, given,
500, euros, in, front, of, the, casino], Res),
flatten(Res, Out).
Out = [<ENAMEX>, M., Dupont, </ENAMEX>, was, given,
<NUMEX>, 500, euros, </NUMEX>, in_front_of, the,
casino]
```

10.5 Noun Groups and Verb Groups

The word detection techniques enabled us to search certain word segments, with no consideration of their category or part of speech. The detection can extend to syntactic patterns.

The two most interesting word groups are derived from the two major parts of speech: the noun and the verb. They are often called **noun groups** and **verb groups**, although **noun chunks** and **verb chunks** are also widely used. In a sentence, noun groups (Table 10.5) and verb groups (Table 10.6) correspond to verbs and nouns and their immediate depending words. This is often understood, although not always, as words extending from the beginning of the constituent to the head noun or the head verb. That is, the groups include the headword and its dependents to the left. They exclude the postmodifiers. For the noun groups, this means that modifying prepositional phrases or, in French, adjectives to the right of the nouns are not part of the groups.

The principles we exposed above are very general, and exact definitions of groups may vary in the literature. They reflect different linguistic viewpoints that may coexist or compete. However, when designing a parser, precise definitions are of primary importance. Like for part-of-speech tagging, hand-annotated corpora will solve the problem. Most corpora come with annotation guidelines. They are

Table 10.5 Noun groups

English	French	German
The waiter *is bringing* ***the very big dish*** *to* ***the table***	***Le serveur*** *apporte* ***le très grand plat*** *sur* ***la table***	***Der Ober*** *bringt* ***die sehr große Speise*** *an* ***den Tisch***
Charlotte *has eaten* ***the meal*** *of* ***the day***	***Charlotte*** *a mangé* ***le plat*** *du* ***jour***	***Charlotte*** *hat* ***die Tagesspeise*** *gegessen*

Table 10.6 Verb groups

English	French	German
The waiter ***is bringing*** *the very big dish to the table*	*Le serveur* ***apporte*** *le très grand plat sur la table*	*Der Ober* ***bringt*** *die sehr große Speise an den Tisch*
Charlotte ***has eaten*** *the meal of the day*	*Charlotte* ***a mangé*** *le plat du jour*	*Charlotte* ***hat*** *die Tagesspeise* ***gegessen***

usually written before the hand-annotation process. As definitions are often difficult to formulate the first time, they are frequently modified or complemented during the annotation process. Guidelines normally contain definitions of groups and examples of them. They should be precise enough to enable the annotators to bracket consistently the groups. The guidelines will provide the grammar writer with accurate definitions, or when using machine learning techniques, the annotated texts will encapsulate the linguistic knowledge about groups and make it accessible to the automatic analysis.

10.5.1 Groups Versus Recursive Phrases

The rationale behind word group detection is that a group structure is simpler and more tractable than that of a sentence. Group detection uses a local strategy that can accept errors without making subsequent analyses of the rest of the sentence fail. It also leaves less room for ambiguity because it sets aside the attachment of prepositional phrases. As a result, partial parsers are more precise. They can capture roughly 90 % of the groups successfully (Abney 1996).

Like for complete sentences, phrase-structure rules can describe group patterns. They are easier to write, however, because verb groups and noun groups have a relatively rigid and well-defined structure. In addition, local rules usually do not describe complex recursive linguistic structures. That is, there is no subgroup inside a group and, for instance, the noun group is limited to a unique head noun. This makes the parser very fast. Moreover, in addition to phrase-structure rules, finite-state automata or regular expressions can also describe group structures.

10.5.2 DCG Rules to Detect Noun Groups

A noun group consists of an optional determiner *the*, *a*, or determiner phrase such as *all of the*, one or more optional adjectives, and one or more nouns. It can also consist of a pronoun or a proper noun – a name. This definition is valid in English. In German, sequences of nouns usually form a single word through compounding. In French, noun groups also include adjectives to the right of the head noun that we set aside.

The core of the noun group is a sequence of nouns also called a nominal expression. A first possibility would be to write as many rules as we expect nouns. However, this would not be very elegant. A recursive definition is more concise: a nominal is then either a noun or a noun and a nominal. Symbols `noun` and `nominal` have variables that unify with the corresponding word. This corresponds to the rules:

```
nominal([NOUN | NOM]) --> noun(NOUN), nominal(NOM).
nominal([N]) --> noun(N).
```

Nouns usually divide into common and proper nouns although, depending on applications, it can be sometimes preferable to ignore the difference between both categories:

```
noun(N) --> common_noun(N).
noun(N) --> proper_noun(N).
```

The simplest noun groups consist of a determiner and a nominal. The determiners are the articles, the possessive pronouns, etc. They are sometimes more complex phrases that we set aside here. Determiners are optional, and the group definition must also represent their absence. A noun group can also be a single pronoun:

```
% noun_group(-NounGroup)
% detects a list of words making a noun group and
% unifies NounGroup with it

noun_group([D | N]) --> det(D), nominal(N).
noun_group(N) --> nominal(N).
noun_group([PRO]) --> pronoun(PRO).
```

The adjective group serves as an auxiliary in the description of noun group. It can feature one or more adjectives and be preceded by an adverb. If we set aside the commas, this corresponds to:

```
%adj_group(-AdjGroup)
%detects a list of words making an adjective
%group and unifies AdjGroup with it

adj_group_x([RB, A]) --> adv(RB), adj(A).
adj_group_x([A]) --> adj(A).
```

```
adj_group(AG) --> adj_group_x(AG).
adj_group(AG) -->
  adj_group_x(AGX), adj_group(AGR),
  {append(AGX, AGR, AG)}.
```

Past participles and gerunds can replace adjectives, as in *A flying object* or *The endangered species*:

```
adj(A) --> past_participle(A).
adj(A) --> gerund(A).
```

We must be aware that these rules may conflict with a subsequent detection of verb groups. Compare the ambiguous phrase *detected words* in *the detected words* and *The partial parser detected words*.

Adjectives can precede the noun. Using the adjective group, we add two rules to the noun group:

```
noun_group(NG) -->
  adj_group(AG), nominal(NOM),
  {append(AG, NOM, NG)}.
noun_group(NG) -->
  det(D), adj_group(AG), nominal(NOM),
  {append([D | AG], NOM, NG)}.
```

10.5.3 DCG Rules to Detect Verb Groups

Verb groups can be written in a similar way. In English, the simplest group consists of a single tensed verb:

```
verb_group([V]) --> tensed_verb(V).
```

Verb groups also include adverbs that may come before the verb:

```
verb_group([RB, V]) --> adv(RB), tensed_verb(V).
```

Verb groups can combine auxiliary and past participles, or auxiliary and gerund, or modal and infinitive, or *to* and infinitive, or be simply an auxiliary:

```
verb_group([AUX, V]) --> aux(AUX),past_participle(V).
verb_group([AUX, G]) --> aux(AUX), gerund(G).
verb_group([MOD, I]) --> modal(MOD), infinitive(I).
verb_group([to, I]) --> [to], infinitive(I).
verb_group([AUX]) --> aux(AUX).
```

Verb groups can include adverbs and have more auxiliaries:

```
verb_group([AUX, RB, V]) -->
  aux(AUX), adv(RB), past_participle(V).
```

```
verb_group([AUX1, AUX2, V]) -->
  aux(AUX1), aux(AUX2), past_participle(V).
verb_group([MOD, AUX, V]) -->
  modal(MOD), aux(AUX), past_participle(V).
```

Now let us write a rule that describes a group inside a word stream: `word_stream_group`. As for with the multiwords, such a stream consists of a beginning, the group, and an end. Its transcription into a DCG rule is:

```
word_stream_group(Beginning, Group, End) -->
  beginning(Beginning),
  group(Group),
  end(End).
```

Finally, a group can either be a noun group or a verb group. As for multiwords, noun groups and verb groups are annotated using the XML tags `<NG>` and `<VG>`:

```
group(NG) -->
  noun_group(Group),
  {append(['<NG>' | Group], ['</NG>'], NG)}.
group(VG) -->
  verb_group(Group),
  {append(['<VG>' | Group], ['</VG>'], VG)}.
```

10.5.4 Running the Rules

Let us write a Prolog program using an approximation of the longest match algorithm to run the rules. The program is similar to the multiword detector:

```
group_detector(In, Out) :-
  word_stream_group(Beginning, Group, End, In, []),
  group_detector(End, Rest),
  append(Beginning, [Group], Head),
  append(Head, Rest, Out).
group_detector(End, End).
```

Since these rules match the longest segments first, they must be written from the longest to the shortest.

Although the grammar is certainly not comprehensive, it can fare reasonably well for a first step. We shall apply it to a text from the *Los Angeles Times* "Flying Blind With the Titans", December 17, 1996:

> Critics question the ability of a relatively small group of big integrated prime contractors to maintain the intellectual diversity that formerly provided the Pentagon with innovative weapons. With fewer design staffs working on military problems, the solutions are likely to be less varied.

We complement the grammar with the lexical rules to identify the part of speech of each word:

```
% Determiners
det(a) --> [a].
det(the) --> [the].

% Common nouns
common_noun(ability) --> [ability].
common_noun(critics) --> [critics].
common_noun(contractors) --> [contractors].
common_noun(design) --> [design].
common_noun(diversity) --> [diversity].
common_noun(group) --> [group].
common_noun(problems) --> [problems].
common_noun(solutions) --> [solutions].
common_noun(staffs) --> [staffs].
common_noun(weapons) --> [weapons].

% Proper nouns
proper_noun(pentagon) --> [pentagon].

% Adverbs
adv(formerly) --> [formerly].
adv(less) --> [less].
adv(likely) --> [likely].
adv(relatively) --> [relatively].

% Adjectives
adj(big) --> [big].
adj(fewer) --> [fewer].
adj(innovative) --> [innovative].
adj(intellectual) --> [intellectual].
adj(military) --> [military].
adj(prime) --> [prime].
adj(small) --> [small].

% Infinitives
infinitive(be) --> [be].
infinitive(maintain) --> [maintain].

% Tensed verbs
tensed_verb(provided) --> [provided].
tensed_verb(question) --> [question].
```

```
% Past participles
past_participle(integrated) --> [integrated].
past_participle(varied) --> [varied].

% Auxiliaries
aux(are) --> [are].
```

And we run our detector, which results in:

```
?- group_detector([critics, question, the, ability,
of, a, relatively, small, group, of, big, integrated,
prime, contractors, to, maintain, the, intellectual,
diversity, that, formerly, provided, the, pentagon,
with, innovative, weapons, with, fewer, design,
staffs, working, on, military, problems, the,
solutions, are, likely, to, be, less, varied], L),
flatten(L, Out).

Out = [<NG>, critics, </NG>, <VG>, question, </VG>,
<NG>, the, ability, </NG>, of, <NG>, a, relatively,
small, group, </NG>, of, <NG>, big, integrated, prime,
contractors, </NG>, <VG>, to, maintain, </VG>, <NG>,
the, intellectual, diversity, </NG>, that, <VG>,
formerly, provided, </VG>, <NG>, the, pentagon, </NG>,
with, <NG>, innovative, weapons, </NG>, with, <NG>,
fewer, design, staffs, </NG>, working, on, <NG>,
military, problems, </NG>, <NG>, the, solutions,
</NG>, <VG>, are, </VG>, likely, <VG>, to, be, </VG>,
less, varied]
```

Though the grammar misses groups, we realize that a limited effort has rapidly produced results.

We must note that this example is slightly artificial because no word has an ambiguous part of speech. A more realistic example would have to deal with this. A way to solve it could be to write as many rules as there are possible parts of speech for a word, for instance:

```
common_noun(question) --> [question].
tensed_verb(question) --> [question].
```

Another way could be to use a part-of-speech tagger as a first processing step and to apply the rules on the part-of-speech sequence.

Table 10.7 Tagset to annotate noun groups

Beginning	End	Between	No bracket (outside)	No bracket (inside)
$[_{NG}$	$_{NG}]$	$_{NG}][_{NG}$	*Outside*	*Inside*

10.6 Group Annotation Using Tags

Group annotation results in bracketing a word sequence with opening and closing annotations. This can be recast as a tagging problem. However, the annotation inserts brackets between words instead of assigning tags to words. The most intuitive annotation is then probably to tag intervals. We can then use algorithms very similar to part-of-speech tagging to carry out the group detection. They give us an alternate method to DCG rules describing verb groups and noun groups.

For the sake of simplicity, we first present annotation schemes for noun groups. We will then generalize them to verb groups and other groups. We describe which tags to use to annotate the intervals, and we will see that we can equivalently tag the words instead of the gaps.

10.6.1 Tagging Gaps

Below are examples of noun group bracketing from Ramshaw and Marcus (1995). They insert brackets between the words where appropriate.

> $[_{NG}$ The government $_{NG}]$ has $[_{NG}$ other agencies and instruments $_{NG}]$ for pursuing $[_{NG}$ these other objectives $_{NG}]$.

> Even $[_{NG}$ Mao Tse-tung $_{NG}]$ $[_{NG}$'s China $_{NG}]$ began in $[_{NG}$ 1949 $_{NG}]$ with $[_{NG}$ a partnership $_{NG}]$ between $[_{NG}$ the communists $_{NG}]$ and $[_{NG}$ a number $_{NG}]$ of $[_{NG}$ smaller, non-communists parties $_{NG}]$.

If we only consider noun groups, the tagset must include opening and ending brackets. There must also be a tag to indicate a separation between two contiguous noun groups. The rest of the gaps are to be labeled with a "no bracket" tag.

As noun group detection usually considers nonrecursive sequences, we avoid nested brackets, as in this sequence: `[ . . .  [` or in this one: `]  . . . ]`. To check nesting while processing the stream, we must make a distinction between a "no bracket" inside a group and "no bracket" outside a group. The tagger can then prevent an inside "no bracket" to be followed by an opening bracket. We complement the tagset with "no bracket" tags denoting either we are within a group or outside (Table 10.7).

In addition to nested groups, other inconsistencies can occur, such as the sequences:

- `[ Outside`
- `] Inside`, or
- `Outside ]`.

The tagger must keep track of the preceding bracket to refuse illegal tag pairs.

10.6.2 Tagging Words

Instead of tagging the gaps, we can equivalently tag the words. Ramshaw and Marcus (1995) defined a tagset of three elements {`I`, `O`, `B`}, where `I` means that the word is inside a noun group, `O` means that the word is outside, and `B` (between) means that the word is at the beginning of a noun group that immediately follows another noun group. Using this tagging scheme, an equivalent annotation of the sentences in Sect. 10.6.1 is:

> The/`I` government/`I` has/`O` other/`I` agencies/`I` and/`I` instruments/`I` for/`O` pursuing/`O` these/`I` other/`I` objectives/`I` ./`O`

> Even/`O` Mao/`I` Tse-tung/`I` 's/`B` China/`I` began/`O` in/`O` 1949/`I` with/`O` a/`I` partnership/`I` between/`O` the/`I` communists/`I` and/`O` a/`I` number/`I` of/`O` smaller/`I` ,/`I` non-communists/`I` parties/`I` ./`O`

As in the case for gap tagging, some inconsistencies can occur, such as the sequence: `O B`. The tagger can refuse such sequences, mapping them to a plausible annotation. That is, in the example above, to change the `B` tag into an `I` tag.

10.6.3 Extending IOB to Two or More Groups

From its original definition, researchers modified the `IOB` scheme and extended it to annotate two or more group categories. The most widespread variant of `IOB` is called `IOB2`, where the first word in a group receives the `B` tag (begin), and the following words the `I` tag. As for `IOB`, words outside the groups are annotated with the `O` tag. Using `IOB2`, the two examples in Sect. 10.6.1 would be annotated as:

> The/`B` government/`I` has/`O` other/`B` agencies/`I` and/`I` instruments/`I` for/`O` pursuing/`O` these/`B` other/`I` objectives/`I` ./`O`

> Even/`O` Mao/`B` Tse-tung/`I` 's/`B` China/`I` began/`O` in/`O` 1949/`B` with/`O` a/`B` partnership/`I` between/`O` the/`B` communists/`I` and/`O` a/`B` number/`I` of/`O` smaller/`B` ,/`I` non-communists/`I` parties/`I` ./`O`

The `IOB2` annotation scheme gained acceptance from the conferences on Computational Natural Language Learning (CoNLL 2000, see Sect. 10.12) that adopted it and went popular enough so that many people now use the term "`IOB` scheme" when they actually mean `IOB2`.

Words	POS	Groups
He	PRP	B-NP
reckons	VBZ	B-VP
the	DT	B-NP
current	JJ	I-NP
account	NN	I-NP
deficit	NN	I-NP
will	MD	B-VP
narrow	VB	I-VP
to	TO	B-PP
only	RB	B-NP
£	#	I-NP
1.8	CD	I-NP
billion	CD	I-NP
in	IN	B-PP
September	NNP	B-NP
.	.	O

Fig. 10.3 An annotation example of syntactic groups using the CoNLL 2000 extended IOB scheme (IOB2). The parts of speech are predicted using Brill's tagger, while the groups are extracted from the Penn Treebank. The noun groups and verb groups are highlighted in *light blue* and *light green*, respectively (After data provided by Tjong Kim Sang and Buchholz (2000))

Extending `IOB` to annotate two or more group categories is straightforward. We just need to use tags with a type suffix as for instance the tagset `{I-Type1, B-Type1, I-Type2, B-Type2, O}` to markup two different group types, `Type1` and `Type2`. CoNLL 2000 (Tjong Kim Sang and Buchholz 2000) again is an example such an annotation extension. The organizers used 11 different group types: noun phrases (NP), verb phrases (VP), prepositional phrases (PP), adverb phrases (ADVP), subordinated clause (SBAR), adjective phrases (ADJP), particles (PRT), conjunction phrases (CONJ), interjections (INTJ), list markers (LST), and unlike coordinated phrases (UPC).[2] The noun phrases, verb phrases, and prepositional phrases making up more the 90 % of all the groups in the CoNLL 2000 corpus.

10.6.4 Annotation Examples from CoNLL 2000, 2002, and 2003

As we saw, the CoNLL shared task in 2000 used the IOB2 tag set to annotate syntactic groups. Figure 10.3 shows an example of it with the sentence:

> He reckons the current account deficit will narrow to only £1.8 billion in September.

whose annotation is:

> [$_{NG}$ He $_{NG}$] [$_{VG}$ reckons $_{VG}$] [$_{NG}$ the current account deficit $_{NG}$] [$_{VG}$ will narrow $_{VG}$] [$_{PG}$ to $_{PG}$] [$_{NG}$ only £1.8 billion $_{NG}$] [$_{PG}$ in $_{PG}$] [$_{NG}$ September $_{NG}$].

[2] We feel that the word "phrase" has a misleading sense here. Most people in the field would understand it differently. The CoNLL 2000 phrases correspond to what we call group or chunk in this book: nonrecursive syntactic groups.

and where the prepositional groups are limited to the preposition to avoid recursive groups. The corpus consists of three columns:

1. The first column contains the sentences with one word per line and a blank line after each sentence.
2. The second column contains the parts of speech of the words. The CoNLL 2000 organizers used Brill's tagger (Sect. 7.3) trained on the Penn Treebank to assign these parts of speech (Tjong Kim Sang and Buchholz 2000). The tagset shown in Table 7.12.
3. The third column contains the groups with the chunk annotation.

The topic of CoNLL 2002 and 2003 shared tasks was to annotate named entities. These tasks reused the ideas laid down in CoNLL 2001 with the IOB2 and IOB tag sets:

- The CoNLL 2002 annotation (Tjong Kim Sang 2002) consists two columns, the first one for the words and the second one for the named entities with four categories, persons (PER), organizations (ORG), locations (LOC), and miscellaneous (MISC). CoNLL 2002 uses IOB2. Figure 10.4, left part, shows the annotation of the sentence:

  [$_{PER}$ Wolff $_{PER}$], a journalist currently in [$_{LOC}$ Argentina $_{LOC}$], played with [$_{PER}$ Del Bosque $_{PER}$] in the final years of the seventies in [$_{ORG}$ Real Madrid $_{ORG}$].

- The CoNLL 2003 annotation (Tjong Kim Sang and De Meulder 2003) has four columns: the words, parts of speech, syntactic groups, and named entities. Both the syntactic groups and named entities use the original IOB scheme. Figure 10.4, right part, shows the annotation of the sentence:

  [$_{ORG}$ U.N. $_{ORG}$] official [$_{PER}$ Ekeus $_{PER}$] heads for [$_{LOC}$ Baghdad $_{LOC}$].

10.7 Machine Learning Methods to Detect Groups

As with part-of-speech tagging, we can use either statistical or symbolic methods to detect groups, where in both cases, statistical classifiers or rules are trained from hand-annotated corpora. Group detection methods usually consider the words and their part of speech in a window around the group tag to identify. This means that group detection generally involves a part-of-speech tagging step before starting to detect the groups.

10.7.1 Group Detection Using Symbolic Rules

The symbolic rules algorithm is very similar to that of Brill's part-of-speech tagging method. The initial tagging considers the part of speech of the word and assigns the

CoNLL 2002	
Words	**Named entities**
Wolff	B-PER
,	O
currently	O
a	O
journalist	O
in	O
Argentina	B-LOC
,	O
played	O
with	O
Del	B-PER
Bosque	I-PER
in	O
the	O
final	O
years	O
of	O
the	O
seventies	O
in	O
Real	B-ORG
Madrid	I-ORG
.	O

CoNLL 2003			
Words	**POS**	**Groups**	**Named entities**
U.N.	NNP	I-NP	I-ORG
official	NN	I-NP	O
Ekeus	NNP	I-NP	I-PER
heads	VBZ	I-VP	O
for	IN	I-PP	O
Baghdad	NNP	I-NP	I-LOC
.	.	O	O

Fig. 10.4 Annotation examples of named entities using the CoNLL 2002 (*left*) and CoNLL 2003 (*right*) IOB schemes. CoNLL 2002 has two columns and uses IOB2. CoNLL 2003 has four columns and uses IOB for the groups and named entities (After data provided by Tjong Kim Sang (2002) and Tjong Kim Sang and De Meulder (2003))

group annotation tag that is most frequently associated with it, that is, `I`, `O`, or `B`. Then, rules applied sequentially modify annotation tags.

Rules consider the immediate context of the tag to be modified, spanning a few words to the left and a few words to the right of the current word. More precisely, they take into account group annotation tags, parts-of-speech tags, and words around the current word. When the context of the current word matches that of the rule being applied, the current tag is altered.

Ramshaw and Marcus (1995) applied a set of 100 templates using a combination of 10 word contexts and 10 part-of-speech contexts, 20 templates in total, and 5 group annotation tag contexts spanning up to 3 words to the left and to the right:

- $\mathtt{W}_0$, $\mathtt{W}_{-1}$, $\mathtt{W}_1$ being, respectively, the current word, the first word to the left, and the first word to the right.
- $\mathtt{T}_0$, $\mathtt{T}_{-1}$, $\mathtt{T}_1$ being, respectively, the part of speech of the current word, of the first word to the left, and of the first word to the right.

Table 10.8 Patterns used in the templates

Word patterns		Noun group patterns	
Pattern	**Meaning**	**Pattern**	**Meaning**
W_0	Current word	G_0	Current noun group tag
W_{-1}	First word to left	G_{-1}, G_0	Tag bigram to left of current word
W_1	First word to right	G_0, G_1	Tag bigram to right of current word
W_{-1}, W_0	Bigram to left of current word	G_{-2}, G_{-1}	Tag bigram to left of current word
W_0, W_1	Bigram to right of current word	G_1, G_2	Tag bigram to right
W_{-1}, W_1	Surrounding words		
W_{-2}, W_{-1}	Bigram to left		
W_1, W_2	Bigram to right		
$W_{-1,-2,-3}$	Words 1 or 2 or 3 to left		
$W_{1,2,3}$	Words 1 or 2 or 3 to right		

Table 10.9 The five first rules from Ramshaw and Marcus (1995)

Pass	Old tag	Context	New tag
1	I	G_1 = O, T_0 = JJ	O
2	-	G_{-2} = I, G_{-1} = I, T_0 = DT	B
3	-	G_{-2} = O, G_{-1} = I, T_{-1} = DT	I
4	I	G_{-1} = I, T_0 = WDT	B
5	I	G_{-1} = I, T_0 = PRP	B

- G_0, G_{-1}, G_1 being, respectively, the group annotation tag of the current word, of the first word to the left, and of the first word to the right.

Table 10.8 shows the complete set of templates. Word and part-of-speech templates are the same.

After training the rules on the Penn Treebank using its part-of-speech tagset, they could retrieve more than 90 % of the noun groups. The five most productive rules are given in Table 10.9. The first rule means that an I tag is changed into an O tag when the current part of speech is an adjective (JJ) and the following word is tagged O. The second rule sets the tag to B if the two previous tags are I and the current word's part of speech is a determiner (DT).

10.7.2 Group Detection Using Stochastic Tagging

We can use the same methods as in Chap. 8 to determine a sequence of group tags. The maximum likelihood estimator determines the optimal sequence of group tags $G = g_1, g_2, g_3, \ldots, g_n$, given a sequence of words $W = w_1, w_2, w_3, \ldots, w_n$ and of part-of-speech tags $T = t_1, t_2, t_3, \ldots, t_n$.

As context of the current group tag g_i, we can use a window of three words w_{i-1}, w_i, and w_{i+1} centered on it as well the surrounding parts of speech: t_{i-1}, t_i,

and t_{i+1}. We estimate $P(g_i|w_{i-1}, w_i, w_{i+1}, t_{i-1}, t_i, t_{i+1})$ using logistic regression, for instance, and we maximize the equation:

$$P(G) = \prod_{i=1}^{n} P(g_i|w_{i-1}, w_i, w_{i+1}, t_{i-1}, t_i, t_{i+1}). \tag{10.1}$$

using the Viterbi algorithm (Sect. 8.2.4).

10.7.3 Using Classifiers

Other classifiers such decision trees, logistic regression, or support vector machines are alternatives to stochastic tagging. As with symbolic rules in Sect. 10.7.1, the baseline considers the current word's part of speech and assigns the group annotation tag that is most frequently associated with this part of speech. Considering a larger window of one or two adjacent words to the left or to the right of the current word improves the performance. For the sentence:

> He/PRP reckons/VBZ the/DT current/JJ account/NN **deficit**/NN will/MD narrow/VB to/TO only/RB £/# 1.8/CD billion/CD in/IN September/NNP ./.

the feature vector (t_{i-1}, t_i, t_{i+1}) associated with the word *deficit* has the value: (NN, NN, MD).

In addition to the parts of speech, a classifier can also use the word values. This corresponds to a **lexicalization** of the model, which usually improves the performance. However, as the training set is always finite, after being trained, the classifier will have to deal with unseen words, words that exist but are not present in the training set. This can be a problem for some learning algorithms like decision trees. A practical solution to this is to build a dictionary of the words seen in the training corpus, where all the words with a frequency lower than a certain threshold are mapped onto a unique token, `OTHER_WORD`. Once trained, instead of using the word value, the group detector will take this symbol when it encounters a word that is not in the dictionary.

The features consisting of parts of speech and word values are called static because they are determined before the program runs. We can build classifiers that use **dynamic features** or **dynamic attributes** that are computed at run time. In our case and provided that the analysis is carried from left to right, the classifier can use the group tag assigned to the word before the current word or to the two words preceding the current word.

Kudoh and Matsumoto (2000) obtained the best results in CoNLL 2000 with a feature set consisting of five words, five parts of speech centered on the current word as well as the two preceding group tags:

$$(w_{i-2}, w_{i-1}, w_i, w_{i+1}, w_{i+2}, t_{i-2}, t_{i-1}, t_i, t_{i+1}, t_{i+2}, g_{i-2}, g_{i-1}).$$

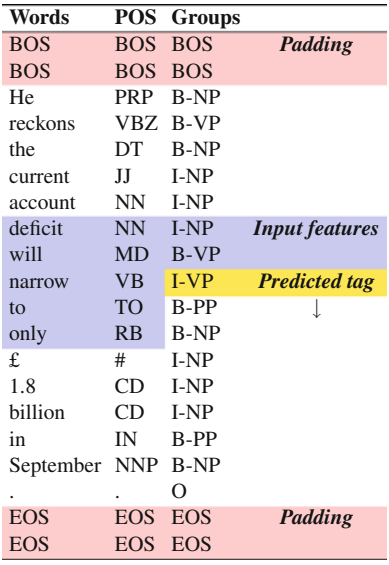


Fig. 10.5 Input features: $(w_{i-2}, w_{i-1}, w_i, w_{i+1}, w_{i+2}, t_{i-2}, t_{i-1}, t_i, t_{i+1}, t_{i+2}, g_{i-2}, g_{i-1})$ used by Kudoh and Matsumoto (2000) and the Yamcha system (After data provided by Tjong Kim Sang and Buchholz (2000))

Table 10.10 Input feature vectors $(w_{i-1}, w_i, w_{i+1}, t_{i-1}, t_i, t_{i+1}, g_{i-1})$ extracted from the sentence in Fig. 10.5. They are used by the classifier at index i to predict the group tag g_i. We used the padding symbols BOS, beginning of sentence, and EOS, end of sentence (After data provided by Tjong Kim Sang and Buchholz (2000))

Input feature vectors							**Output**
w_{i-1}	w_i	w_{i+1}	t_{i-1}	t_i	t_{i+1}	g_{i-1}	g_i
BOS	He	reckons	BOS	PRP	VBZ	BOS	B-NP
He	reckons	the	PRP	VBZ	DT	B-NP	B-VP
reckons	the	current	VBZ	DT	JJ	B-VP	B-NP
the	current	account	DT	JJ	NN	B-NP	I-NP
current	account	deficit	JJ	NN	NN	I-NP	I-NP
account	deficit	will	NN	NN	MD	I-NP	I-NP
deficit	will	narrow	NN	MD	VB	I-NP	B-VP
will	narrow	to	MD	VB	TO	B-VP	I-VP
narrow	to	only	VB	TO	RB	I-VP	B-PP
to	only	£	TO	RB	#	B-PP	B-NP
only	£	1.8	RB	#	CD	B-NP	I-NP
£	1.8	billion	#	CD	CD	I-NP	I-NP
1.8	billion	in	CD	CD	IN	I-NP	I-NP
billion	in	September	CD	IN	NNP	I-NP	B-PP
in	September	.	IN	NNP	.	B-PP	B-NP
September	.	EOS	NNP	.	EOS	B-NP	O

Figure 10.5 shows this feature set graphically, and Table 10.10 shows the features of a smaller set $(w_{i-1}, w_i, w_{i+1}, t_{i-1}, t_i, t_{i+1}, g_{i-1})$ extracted for all the words in the sentence.

Table 10.11 Group detection efficiency using different feature sets. The training data was extracted from the CoNLL 2000 training set, and the decision trees were trained using the C4.5 implementation available from the Weka environment. They were then applied to the CoNLL 2000 test set. In the lexicalized models, we used words that had a frequency greater than 100 in the training set. All the other words were mapped onto the `OTHER_WORD` symbol. The evaluation was carried out using the CoNLL 2000 script, which uses the F-measure, see Sect. 10.11.3 for a definition. The annotated data as well as the evaluation script are available from the CoNLL 2000 web page

Context					**Models**			
T_{-2}	T_{-1}	T_0	T_{+1}	T_{+2}	POS	POS	POS	POS
W_{-2}	W_{-1}	W_0	W_{+1}	W_{+2}	–	Words	–	Words
G_{-2}	G_{-1}				–	–	Groups	Groups
–	–	•	–	–	77.07	79.51	–	–
–	•	•	–	–	81.88	85.88	83.03	86.68
•	•	•	–	–	82.84	86.48	83.11	86.75
–	•	•	•	–	87.13	89.75	88.36	90.28
•	•	•	•	•	88.34	90.40	88.61	90.53

10.7.4 Group Detection Performance and Feature Engineering

The choice of a feature set is very significant for the performance of a classifier. Table 10.11 shows examples of with the C4.5 decision tree classifier using different sets. We used part-of-speech tags, lexical values, and group tags extracted from contexts of different sizes. Choosing and tuning an optimal feature set is a delicate operation that has to balance precision and computational requirements. It is often referred to as **feature engineering**.

As a general rule, the larger the contexts, the better the results, however, the result improvements are not proportional to the growth of the context size and tend quickly to reach a plateau. Some feature sets of are more efficient too. The classifier using features extracted from the two words surrounding the current word outperforms the one using the two words preceding it by 4.29 for nonlexicalized models (87.13 vs. 82.84), although the number of features is equal. We can see that lexicalization improves the figures by 2.88 on average. The gain brought by dynamic features – group tags – is more modest, 0.58 on average.

The best figure obtained by the classifier is for a window of five words with 12 parameters, $t_{-2}, t_{-1}, t_0, t_{+1}, t_{+2}, w_{-2}, w_{-1}, w_0, w_{+1}, w_{+2}, g_{-2}, g_{-1}$, not far off from the figure of 93.48 obtained by Kudoh and Matsumoto (2000) and the Yamcha system with support vector machines, a more efficient, but slower, machine learning algorithm. The figures obtained with the lexicalized models in Table 10.11 use a threshold value of 100, i.e., all the words with a frequency <100 in the training corpus are mapped onto the `OTHER_WORD` symbol.

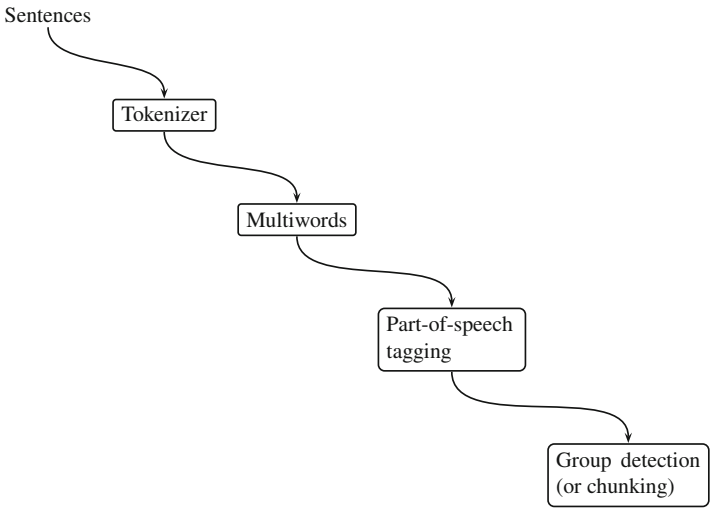


Fig. 10.6 A cascade of partial parsers

10.8 Cascading Partial Parsers

We saw that partial phrase-structure rules or statistical taggers could detect multiwords and groups. We can combine both detectors into a multilevel parser and add more layers. A tokenizer is necessary to read the text before it can be passed to the parsers. The applications generally use a part-of-speech tagger before the group detector (or **chunker**) and sometimes a morphological parser. The parser's structure is then a pipeline of analyzers, where each parsing level has a definite task to achieve. This technique is referred to as cascaded parsing.

With this approach, the exact number and nature of levels of cascaded parsers depends on the application and the expected result. In addition, some layers are generic, like tokenization, while others may be more specific and depend on the application goal. However, the principle is that one level uses the output of the lower level and passes on the result to the next layer (Fig. 10.6). This corresponds precisely to the format and organization of the CoNLL corpora.

10.9 Elementary Analysis of Grammatical Functions

10.9.1 Main Functions

In a previous section, we named groups according to the part of speech of their main word, that is, noun groups and verb groups. We can also consider their grammatical

function in the sentence. We already saw that main functions (or relations) are subject, direct object, and indirect object. An accurate detection of function is difficult, but we can write a simplified one using cascaded parsing and phrase-structure rules.

We can recognize grammatical functions using a layer above those we have already described and thus complement the cascade structure. In English, the subject is generally the first noun group of a sentence in the active voice. It is marked with the nominative case in German, while case inflection is limited to pronouns in English and French. The direct object is the noun group just after the verb if there is no preposition in-between. It is marked with the accusative case in German.

We will now write a small set of DCG rules to encode this simplified description. The structure of a simple sentence consists of a subject noun group, a verb group in the active voice, and an object noun group. It corresponds to the rules:

```
sentence(S, V, O) -->
  subject(S), verb(V, active), object(O), ['.'].

subject(S) --> noun_group(S).

object(O) --> noun_group(O).

verb(V, active) --> verb_group(V, active).
```

We must modify the description of verbs in the terminal symbols to add an active/passive feature.

10.9.2 Extracting Other Groups

The subject–verb–object relation is the core of most sentences. However, before extracting them, it is useful to skip some groups between them. Among the groups, there are prepositional phrases and embedded clauses, as in the two sequences: subject, prepositional groups, verb and subject, relative clause, verb.

A prepositional group can be defined as a preposition followed by a noun group. Using a DCG rule, this translates into:

```
prep_group([P | [NG]]) --> prep(P), ng(NG).
```

The detection of prepositional groups is a new layer in the cascade structure. A new rule describing `ng` as a terminal symbol is then necessary to be consistent with the noun groups detected before:

```
ng(['<NG>'| NG]) --> [['<NG>'| NG]].
```

Embedded clauses can be relative, infinitive, or subordinate. Here we will only consider relative and infinitive clauses that may modify a noun.

A relative clause is an embedded sentence whose subject or object has been replaced with a relative pronoun. The relative pronoun comes in front of the clause. For simple clauses, this translates into two rules:

```
%Relative clause: The relative pronoun is the subject
relative_clause(RC) -->
  relative_pronoun(R), vg(VG), ng(NG),
  {append([R | [VG]], [NG], RC)}.

% Relative clause: The relative pronoun is the object
relative_clause(RC) -->
  relative_pronoun(R), ng(NG), vg(VG),
  {append([R | [NG]], [VG], RC)}.
```

An infinitive clause is simply a verb phrase set in the infinitive. For simple examples, it translates into a verb group possibly followed by a noun group, where the verb group begins with *to*:

```
infinitive_clause([['<VG>', to | VG], NG]) -->
  vg(['<VG>', to | VG]), ng(NG).
infinitive_clause([['<VG>', to | VG]]) -->
  vg(['<VG>', to | VG]).
```

Like for noun groups, we must describe verb groups as a terminal symbol:

```
vg(['<VG>' | VG]) --> [['<VG>' | VG]].
```

Now let us write the rules to describe the modifiers and annotate them:

```
modifier(MOD) -->
  prep_group(PG),
  {append(['<PG>' | PG], ['</PG>'], MOD)}.
modifier(MOD) -->
  relative_clause(RC),
  {append(['<RC>' | RC], ['</RC>'], MOD)}.
modifier(MOD) -->
  infinitive_clause(IC),
  {append(['<IC>' | IC], ['</IC>'], MOD)}.
```

Finally, we write the detector to run the program:

```
modifier_detector(In, Out) :-
  word_stream_modifier(Beginning, Group, End, In,[]),
  modifier_detector(End, Rest),
  append(Beginning, [Group], Head),
  append(Head, Rest, Out).
modifier_detector(End, End).

word_stream_modifier(Beginning, Group, End) -->
  beginning(Beginning),
```

```
    modifier(Group),
    end(End).
```

Let us apply these rules on the first sentence of the *Los Angeles Times* excerpt. We must add prepositions and a relative pronoun to the vocabulary:

```
prep(of) --> [of].
prep(with) --> [with].

relative_pronoun(that) --> [that].
```

And the query yields:

```
?- modifier_detector([[<NG>, critics, </NG>], [<VG>,
question, </VG>], [<NG>, the, ability, </NG>], of,
[<NG>, a, relatively, small, group, </NG>], of, [<NG>,
big, integrated, prime, contractors, </NG>], [<VG>,
to, maintain, </VG>], [<NG>, the, intellectual,
diversity, </NG>], that, [<VG>, formerly, provided,
</VG>], [<NG>, the, pentagon, </NG>], with, [<NG>,
innovative, weapons, </NG>], with, [<NG>, fewer,
design, staffs, </NG>], working, on, [<NG>, military,
problems, </NG>], [<NG>, the, solutions, </NG>],
[<VG>, are, </VG>], likely, [<VG>, to, be, </VG>],
less, varied], O).

O = [[<NG>, critics, </NG>], [<VG>, question, </VG>],
[<NG>, the, ability, </NG>], [<PG>, of, [<NG>, a,
relatively, small, group, </NG>], </PG>], [<PG>, of,
[<NG>, big, integrated, prime, contractors, </NG>],
</PG>], [<IC>, [<VG>, to, maintain, </VG>], [<NG>,
the, intellectual, diversity, </NG>], </IC>], [<RC>,
that, [<VG>, formerly, provided, </VG>], [<NG>, the,
pentagon, </NG>], </RC>], [<PG>, with, [<NG>,
innovative, weapons, </NG>], </PG>], [<PG>, with,
[<NG>, fewer, design, staffs, </NG>], </PG>], working,
on, [<NG>, military, problems, </NG>], [<NG>, the,
solutions, </NG>], [<VG>, are, </VG>], likely, [<IC>,
[<VG>, to, be, </VG>], </IC>], less, varied]
```

Prepositional phrases and relative clauses are labeled with `<PG>`, `<IC>`, and `<RC>` tags. Remaining groups are `[<NG>, critics, </NG>]`, `[<VG>,question, </VG>]`, and `[<NG>, the, ability, </NG>]`, which correspond to heads of the subject, main verb, and the object of the sentence.

10.10 An Annotation Scheme for Groups in French

The PEAS initiative (Gendner et al. 2003) defines an XML annotation scheme for syntactic groups (chunks) and functional relations for French. It was created to reconcile different annotation practices and enable the evaluation of parsers. We present here the chunk annotation that applies to continuous, nonrecursive constituents.

The PEAS annotation identifies six types of chunks:

1. Verb groups (noyau verbal): <NV></NV>
2. Noun groups (groupe nominal): <GN></GN>
3. Prepositional groups: <GP></GP>
4. Adjective groups: <GA></GA>
5. Adverb groups: <GR></GR>
6. Verb groups starting with a preposition: <PV></PV>

The sentence *En quelle année a-t-on vraiment construit la première automobile?* 'Which year the first automobile was really built?' is bracketed as

<GP> En quelle année </GP> <NV> a –t-on </NV> <GR> vraiment </GR> <NV> construit </NV> <GN> la première automobile</GN> ?

The annotation first identifies the sentence in the corpus:

```
<E id="2"> En quelle année a -t-on vraiment
construit la première automobile ? </E>
```

The second step tokenizes the words:

```
<DOCUMENT fichier="Guide.1">
  <E id="E2">
    <F id="E2F1">En</F>
    <F id="E2F2">quelle</F>
    <F id="E2F3">année</F>
    <F id="E2F4">a</F>
    <F id="E2F5">-t-on</F>
    <F id="E2F6">vraiment</F>
    <F id="E2F7">construit</F>
    <F id="E2F8">la</F>
    <F id="E2F9">première</F>
    <F id="E2F10">automobile</F>
    <F id="E2F11">?</F>
  </E>
</DOCUMENT>
```

using the DTD

```
<!ELEMENT DOCUMENT ( E+ ) >
<!ATTLIST DOCUMENT fichier NMTOKEN #REQUIRED >
<!ELEMENT E ( F)+>
```

```
<!ATTLIST E id NMTOKEN #REQUIRED >
<!ELEMENT F ( #PCDATA ) >
<!ATTLIST F id ID #REQUIRED >
```

The third step brackets the groups:

```
<DOCUMENT fichier="Guide.1.ph1.IR.xml">
  <E id="E2">
    <Groupe type="GP" id="E2G1">
     <F id="E2F1">En</F>
     <F id="E2F2">quelle</F>
     <F id="E2F3">année</F>
    </Groupe>
    <Groupe type="NV" id="E2G2">
      <F id="E2F4">a</F>
      <F id="E2F5">-t-on</F>
    </Groupe>
    <Groupe type="GR" id="E2G3">
     <F id="E2F6">vraiment</F>
    </Groupe>
    <Groupe type="NV" id="E2G4">
      <F id="E2F7">construit</F>
    </Groupe>
    <Groupe type="GN" id="E2G5">
      <F id="E2F8">la</F>
      <F id="E2F9">première</F>
      <F id="E2F10">automobile</F>
    </Groupe>
      <F id="E2F11">?</F>
  </E>
</DOCUMENT>
```

using the DTD

```
<!ELEMENT DOCUMENT ( E+ ) >
<!ATTLIST DOCUMENT fichier NMTOKEN #REQUIRED >
<!ELEMENT E ( F | Groupe )+>
<!ATTLIST E id NMTOKEN #REQUIRED >
<!ELEMENT Groupe ( F+ ) >
<!ATTLIST Groupe id ID #REQUIRED >
<!ATTLIST Groupe type ( GA | GN | GP | GR | NV |
PV ) #REQUIRED >
<!ELEMENT F ( #PCDATA ) >
<!ATTLIST F id ID #REQUIRED >
```

10.11 Application: Information Extraction and the FASTUS System

Quis, quid, quando, ubi, cur, quem ad modum, quibus adminiculis
'Who, what, when, where, why, in what manner, by what means'
Hermagoras, *De rhetorica*, in K. Halm, *Rhetores latini minores*, p. 141.

10.11.1 The Message Understanding Conferences

The FASTUS system was designed at the Stanford Research Institute (SRI) to extract information from free-running text (Appelt et al. 1993; Hobbs et al. 1997). It was implemented within the course of the Message Understanding Conferences (MUCs) that we introduced in Sect. 10.3.3. MUCs were organized to measure the performance of news monitoring systems. They were held regularly until MUC-7 in 1997, under the auspices of DARPA, an agency of the US Department of Defense. The performances improved dramatically in the beginning and then stabilized. DARPA discontinued the competitions when it realized that the systems were no longer improving.

MUCs are divided into a set of tasks that have changed over time. The most basic task is to extract people and company names. The most challenging one is referred to as information extraction. It consists of the analysis of pieces of text ranging from one to two pages, the identification of entities or events of a specified type and their circumstances, and filling a predefined template with relevant information from the text. Information extraction then transforms free texts into tabulated information. Here is an example news wire cited by Hobbs et al. (1997) and its corresponding filled template drawn from MUC-3 (Table 10.12):

> San Salvador, 19 Apr 89 (ACAN-EFE) – [TEXT] Salvadoran President-elect Alfredo Cristiani condemned the terrorist killing of Attorney General Roberto Garcia Alvarado and accused the Farabundo Marti National Liberation Front (FMLN) of the crime.
> . . .
> Garcia Alvarado, 56, was killed when a bomb placed by urban guerrillas on his vehicle exploded as it came to a halt at an intersection in downtown San Salvador.
> . . .
> Vice President-elect Francisco Merino said that when the attorney general's car stopped at a light on a street in downtown San Salvador, an individual placed a bomb on the roof of the armored vehicle.
> . . .
> According to the police and Garcia Alvarado's driver, who escaped unscathed, the attorney general was traveling with two bodyguards. One of them was injured.

Table 10.12 A template derived from the previous text (After Hobbs et al. (1997))

Template slots	Information extracted from the text
Incident: Date	19 Apr 89
Incident: Location	El Salvador: San Salvador (city)
Incident: Type	Bombing
Perpetrator: Individual ID	*urban guerrillas*
Perpetrator: Organization ID	*FMLN*
Perpetrator: Organization confidence	Suspected or accused by authorities: *FMLN*
Physical target: Description	*vehicle*
Physical target: Effect	Some damage: *vehicle*
Human target: Name	*Roberto Garcia Alvarado*
Human target: Description	*Attorney general*: *Roberto Garcia Alvarado* *driver* *bodyguards*
Human target: Effect	Death: *Roberto Garcia Alvarado* No injury: *driver* Injury: *bodyguards*

10.11.2 The Syntactic Layers of the FASTUS System

FASTUS uses partial parsers that are organized as a cascade of finite-state automata. It includes a tokenizer, a multiword detector, and a group detector as first layers. The detector uses a kind of longest match algorithm. Verb groups are tagged with active, passive, gerund, and infinitive features. Then FASTUS combines some groups into more complex phrases. Complex groups include notably the combination of adjacent nouns groups (appositives):

The joint venture, Bridgestone Sports Taiwan Co.
First noun group Second noun group

of noun groups separated by prepositions *of* or *for* (noun postmodifiers):

The board of directors

and of noun group conjunctions:

a local concern and a Japanese trading house

Complex groups also include verb expressions such as:

plan to set up
announced a plan to form

Such complex groups can be found in French and German, where they have often a one-word counterpart in another language:

mettre une lettre à la poste 'mail a letter'
jemanden kennen lernen 'know somebody'

Table 10.13 Documents in a library returned from a catalog query and split into relevant and irrelevant books

	Relevant documents	**Irrelevant documents**
Retrieved	A	B
Not retrieved	C	D

They merely reduce to a single semantic entity that is formed differently from one language to another.

FASTUS' upper layers then deal with grammatical functions and semantics. FASTUS attempts to reduce sentences to a basic pattern consisting of a subject, a verb, and an object. Finally, FASTUS assigns a sense to some groups by annotating them with a semantic category such as company, product, joint venture, location, and so on.

SRI first used a full parser called TACITUS, and FASTUS as a front-end to offload it of some tasks. Seeing the excellent results and speed of FASTUS, SRI completely replaced TACITUS with FASTUS. It had a considerable influence on the present evolution of parsing techniques. FASTUS proved that the local and cascade approach was more efficient and much faster than other global analyses for information extraction. It had a considerable number of followers.

10.11.3 Evaluation of Information Extraction Systems

The MUCs introduced a metric to evaluate the performance of information extraction systems using three figures: recall, precision, and the F-measure. This latter metric, originally borrowed from library science, proved very generic to summarize the overall effectiveness of a system. It has been used in many other fields of language processing since then.

To explain these figures, let us stay in our library and imagine we want to retrieve all the documents on a specific topic, say *morphological parsing*. An automatic system to query the library catalog will, we hope, return some of them, but possibly not all. On the other hand, everyone who has searched a catalog knows that we will get irrelevant documents: *morphological pathology*, *cell morphology*, and so on. Table 10.13 summarizes the possible cases into which documents fall.

Recall measures how much relevant information the system has retrieved. It is defined as the number of relevant documents retrieved by the system divided by number of relevant documents in the library:

$$\text{Recall} = \frac{|A|}{|A \cup C|}.$$

Precision is the accuracy of what has been returned. It measures how much of the information is actually correct. It is defined as the number of correct documents returned divided by the total number of documents returned.

$$\text{Precision} = \frac{|A|}{|A \cup B|}.$$

Recall and precision are combined into the F**-measure**, which is defined as the harmonic mean of both numbers:

$$F = \frac{2PR}{P + R}.$$

The F-measure is a composite metric that reflects the general performance of a system. It does not privilege precision at the expense of recall, or vice versa. An arithmetic mean would have made it very easy to reach 50 % using, for example, very selective rules with a precision of 100 and a low recall.

Using a β-coefficient, it is possible to give an extra weight to either precision, $\beta > 1$, or recall, $\beta < 1$, however:

$$F = \frac{(\beta^2 + 1)PR}{\beta^2 P + R}.$$

Finally, a **fallout** figure is also sometimes used that measures the proportion of irrelevant documents that have been selected.

$$\text{Fallout} = \frac{|B|}{|B \cup D|}.$$

10.12 Further Reading

The development of partial parsing has been mainly driven by applications without concern for a specific linguistic framework. This is a notable difference from many other areas of language processing, where theories abound. Due to the simplicity of the methods involved, partial or shallow parsing attracted considerable interest in the 1990s and renewed the field. Its successes in information extraction competitions such as the MUCs, where it proved that it could outperform classical parsers, also contributed to its popularity. See, for instance, MUC-5 (1993).

One of the first partial parsing systems is due to Ejerhed (1988). Church (1988) first addressed group detection as a tagging problem and used statistical methods. He tagged the gaps with brackets. Ramshaw and Marcus (1995) used a symbolic strategy. They created the `IOB` tagset, and they adapted Brill's (1995) algorithm to learn rules to detect groups from annotated corpora. Kudoh and Matsumoto (2000) applied classifiers based on support vector machines that are to date the best-performing methods for group detection. Abney (1994) is a rather old but still valuable survey of partial parsing with much detail that provides a comprehensive bibliography of 200 papers! Roche and Schabes (1997) and Kornai (1999) are other

sources for partial parsing techniques. On the application side, Appelt et al. (1993) describe with eloquence the history and structure of the FASTUS system.

FASTUS started a long line of information extraction systems that are now ubiquitous. Techniques to find the circumstances of an event have a long history. Hermagoras of Temnos quoted in Sect. 10.11 is known to have formulated first the seven fundamental elements to answer. These elements were used throughout the Middle Ages by confessors for the examination of penitents, and they are now used by search engines. For a review of the early history, see Robertson (1946).

Partial parsing was the topic of a series of conferences on Computational Natural Language Learning (CoNLL). Each year, the CoNLL conference organizes a "shared task" where it provides an annotated training set. Participants can train their system on this set, evaluate it on a common test set, and report a description of their algorithms and results in the proceedings. In 1999, the shared task was dedicated to noun group chunking (http://www.cnts.ua.ac.be/conll99/npb/); in 2000 it was extended to other chunks (http://www.cnts.ua.ac.be/conll2000/chunking/); in 2001 the topic was the identification of clauses (http://www.cnts.ua.ac.be/conll2001/clauses/); and in 2002 and 2003 the task was multilingual named entity recognition (http://www.cnts.ua.ac.be/conll2002/ner/ and http://www.cnts.ua.ac.be/conll2003/ner/). The CoNLL sites and proceedings are extremely valuable as they provide data sets, annotation schemes, a good background literature, and an excellent idea of the state of the art.

Exercises

10.1. Complement the ELIZA program and add possible templates and answers.

10.2. Implement a multiword detector to detect dates in formats such as in English, 04/04/1997 or April 4, 1997, and in French, 20/04/1997 or 20 avril 1997.

10.3. Complement the noun group grammar from Sect. 10.5.2 and write down the vocabulary to recognize the noun groups of the text:

> The big tobacco firms are fighting back in the way that served them well for 40 victorious years, pouring their wealth into potent, relentless legal teams. But they are also starting to talk of striking deals – anathema for those 40 years, and a sure sign that, this time, victory is less certain.
> *The Economist*, no. 8004, 1997.

10.4. See Exercise 10.3; do the same for verb groups.

10.5. Write a noun group grammar to parse the French text:

> Les limites de la régulation de l'audiovisuel sont clairement définies aujourd'hui par la loi. C'est le principal handicap du CSA: son champ d'action est extrêmement limité. Alors que la télévision numérique prend son essor, le CSA, dont les compétences s'arrêtent au câble et à l'hertzien, n'a aucun pouvoir pour contrôler ou sanctionner la télévision de demain formée

par les chaînes satellitaires.
Le Monde, mercredi 3 septembre 1997.

10.6. See Exercise 10.5; do the same for verb groups.

10.7. Write a noun group grammar to parse the German text:

Die Freude über das neue große Europa wird also nur von kurzer Dauer sein. Die Probleme, die sich aus einer Union der 25 ergeben, dürften dagegen Regierungen und Völker über Jahre hinweg in Atem halten. Zunächst einmal wird es alles andere als leicht sein, die 10 neuen Mitgliedsstaaten zu integrieren. Die Migrationswellen, die von ihnen ausgehen, werden der „alten" EU reichlich Kopfschmerzen bereiten. Vor allem stellt sich der Entscheidungsprozess innerhalb der Union künftig noch weitaus schwieriger dar.
Die Zeit, 30 April 2004.

10.8. See Exercise 10.7; do the same for verb groups.

10.9. Write a Prolog program to convert the IOB2 tagging scheme explained in Sect. 10.6.2 into a bracketed notation such as the one described in Sect. 10.10. Apply it to the CoNLL-2000 corpora available from this page: http://www.cnts.ua.ac.be/conll2000/chunking/.

10.10. See Exercise 10.9 and write a reverse converter to translate a bracketed notation into an IOB2 tagging scheme.

10.11. Download the annotated corpora available from the CoNLL-2000 shared task as well as the evaluation script (http://www.cnts.ua.ac.be/conll2000/chunking/). Apply the noun group rules from Sect. 10.5.2 to detect them in the corpora and evaluate the rule efficiency using the CoNLL-2000 evaluation script. Use only the part-of-speech information. Complement the rules so that you reach a figure of 80 for the noun groups.

10.12. See Exercise 10.11; do the same for the verb groups.

10.13. Adapt the Prolog code of Brill's tagger from Chap. 7 so that it can detect noun groups.

10.14. Download the group annotated corpora from the CoNLL-2000 web page (http://www.cnts.ua.ac.be/conll2000/chunking/). From the training corpus, extract the feature vectors corresponding to different feature sets shown in Fig. 10.5 and Table 10.10 as described in Sect. 10.7.3. You can start from the feature set corresponding to the baseline – part of speech and group annotation tag of the current word – and then try one or two more models. Train the corresponding decision tree classifiers, apply them to bracket the test set, and evaluate the results using the CoNLL-2000 evaluation script. To build the classifiers, you can use the Weka implementation of C4.5 available from this site: http://www.cs.waikato.ac.nz/ml/weka/.

10.15. Write rules that detect some complex noun groups:

- Adjacent noun groups linked by the prepositions *of* or *for*
- Noun group conjunctions

10.16. Find press wires on football matches on the Web and implement a program to retrieve teams' names and final scores. Use a base of football team names, and adopt a cascaded architecture.

Chapter 11
Syntactic Formalisms

> A word is said to *govern* another word, which *depends* upon it in the sentence. Thus, when we say Charles loves study, the verb loves, *governs* the noun, study. So when we say, I am going to the church, the preposition *to*, governs the noun, *church*. Agreement and government are called *Syntax*.
> Chauncey Allen Goodrich, *Lessons in Latin Parsing*, Durrie & Peck, New Haven, 1838.

11.1 Introduction

Studies on syntax have been the core of linguistics for most of the twentieth century. While the goals of traditional grammars had been mostly to prescribe what the correct usage of a language is, the then-emerging syntactic theories aimed at an impartial description of language structures. These ideas revolutionized the field. Research activity was particularly intense in the years 1940–1970, and the focus on syntax was so great that, for a time, it nearly eclipsed phonetics, morphology, semantics, and other disciplines of linguistics.

Among all modern syntax researchers, Noam Chomsky has had a considerable and indisputable influence. Chomsky's seminal work, *Syntactic Structures* (1957), is still considered by many as a key reading in linguistics. In his book (in Sect. 6.1), Chomsky defined grammars as *essentially a theory of* [a language] that should be (1) adequate: whose correctness should be measurable using corpora; (2) general: extendible to a variety of languages; and, as far as possible, (3) simple. As goals, he assigned grammatical rules to describe syntactic structures:

> These rules express structural relations among the sentences of the corpus and the indefinite number of sentences generated by the grammar beyond the corpus (predictions).

P.M. Nugues, *Language Processing with Perl and Prolog*, Cognitive Technologies,
DOI 10.1007/978-3-642-41464-0_11, © Springer-Verlag Berlin Heidelberg 2014

More specifically (in Sect. 5.5), Chomsky outlined a formal model of syntax under the form of grammars that was precise enough to be programmable and verifiable.

Chomsky's ideas appealed to the linguistics community because they featured an underlying analogy between human languages and computer – or formal – languages together with a mathematical formalism that was already used for compilers. Chomsky came at a convergence point where advances in computer technology, mathematical logic, and programming languages made his theory possible and acceptable. Chomsky's theories on syntactic structures have originated much research in the domain and have an astounding number of followers, notably in the United States. In addition, his theories spurred a debate that went well beyond linguistic circles reaching psychology and philosophy.

In the meantime, linguists, mostly in Europe, developed other structural approaches and also tried to derive generic linguistic structures. But instead of using the computer operation as a model or to posit cognition universals, as Chomsky did, some of them tried to study and expose examples from a variety of languages to prove their theories. The most prominent figure of the European school is Lucien Tesnière. Although Tesnière's work (1959, 2nd edn., 1966, both posthumous) is less known, it is gaining recognition and it is used with success in implementations of grammars and parsers for English, French, German, and many other languages.

Many computational models of syntactic structures inherited from Chomskyan grammars use the notion of constituent – although Chomsky does not limit grammars to a constituent decomposition. Traditional approaches are based on the notion of connections between words, where each word of a sentence is linked to another one under a relation of subordination or dependence. For this reason, these syntactic models are also called dependency grammars. This chapter introduces both structural approaches – **constituency** and **dependency** – and associated formalisms.

11.2 Chomsky's Grammar in Syntactic Structures

Chomsky fractionates a grammar into three components. The first level consists of phrase-structure (PS) rules expressing constituency. The second one is made of transformation rules that complement PS rules. **Transformations** enable us to derive automatically new constructions from a given structure: a declarative form into an interrogative or a negative one; an active sentence into a passive one. Transformation rules apply to constituent structures or trees and describe systematic mappings onto new structures.

Initially, PS and transformation rules used a vocabulary made of morphemes, roots, and affixes, as well as complete words. The inflection of a verb with the past participle tense was denoted [*en* + *verb*] where *en* represented the past participle affix, for example, [*en* + *arrive*]. A third **morphophonemic** component handled the final word generation, mapping forms such as [*en* + *arrive*] onto *arrived*.

Sentence	0
NP + VP	1
T + N + VP	2
T + N + Verb + NP	3
the + N + Verb + NP	4
the + man + Verb + NP	5
the + man + hit + NP	6
the + man + hit + T + N	7
the + man + hit + the + N	8
the + man + hit + the + ball	9

Fig. 11.1 Generation of sentences

11.2.1 Constituency: A Formal Definition

Constituency is usually associated with context-free grammars. Formally, such grammars are defined by:

1. A set of designated start symbols, Σ, covering the sentences to parse. This set can be reduced to a single symbol, such as `sentence`, or divided into more symbols: `declarative_sentence`, `interrogative_sentence`.
2. A set of nonterminal symbols enabling the representation of the syntactic categories. This set includes the sentence and phrase categories.
3. A set of terminal symbols representing the vocabulary: words of the lexicon, possibly morphemes.
4. A set of rules, F, where the left-hand-side symbol of the rule is rewritten in the sequence of symbols of the right-hand side.

Chomsky (1957) portrayed PS rules with an example generating *the man hit the ball*. It has a straightforward equivalent in DCG:

```
sentence --> np, vp.
np --> t, n.
vp -- verb, np.
t --> [the].
n --> [man] ; [ball] ; etc.
verb --> [hit] ; [took] ; etc.
```

A set of such PS rules can generate sentences. Chomsky illustrated it using a mechanism that resembles the top-down algorithm of Prolog (Fig. 11.1).

Generation was the main goal of Chomsky's grammars: to produce all potential sentences – word and morpheme sequences – considered to be syntactically correct or acceptable by native speakers. Chomsky introduced recursion in grammars to give a finite set of rules an infinite capacity of generation.

From the initial goal of generation, computational linguists wrote and used grammars to carry out recognition – or parsing – of syntactically correct sentences. A sentence has then to be matched against the rules to check whether it falls within

the generative scope of the grammar. Parsing results in a parse tree – the sequence of grammar rules that were applied. The parsing process can be carried out using:

- A top-down mechanism, which starts from the initial symbol – the sentence – down to the words of the sentence to be parsed
- A bottom-up mechanism, which starts from the words of the sentence to be parsed up to the sentence symbol.

Some parsing algorithms run more efficiently with a restricted version of context-free grammars called the **Chomsky normal form** (CNF). Rules in the CNF have either two nonterminal symbols to their right-hand side or one nonempty terminal symbol:

```
lhs --> rhs1, rhs2.
lhs --> [a].
```

Any grammar can be converted into an equivalent CNF grammar using auxiliary symbols and rules as for

```
lhs --> rhs1, rhs2, rhs3.
```

which is equivalent to

```
lhs --> rhs1, lhs_aux.
lhs_aux --> rhs2, rhs3.
```

The equivalence is said to be weak because the resulting grammar generates the same sentences but does not yield exactly the same parse trees.

11.2.2 Transformations

The transformational level consists of the mechanical rearrangement of sentences according to some syntactic relations: active/passive, declarative/interrogative, etc. A transformation operates on a sentence with a given phrase structure and converts it into a new sentence with a new derived phrase structure. Transformations use rules – transformational rules or T-rules – to describe the conversion mechanism as:

```
T1:  np1, aux, v, np2 →
     np2, aux, [be], [en], v, [by], np1
```

which associates an active sentence to its passive counterpart. The active part of the rule matches sentences such as:

the man will hit the ball

and its passive part enables us to generate the equivalent passive sentence:

the ball will be (en hit) by the boy

where (*en hit*) corresponds to the past participle of verb *to hit*. An additional transformational rule permutes these two elements:

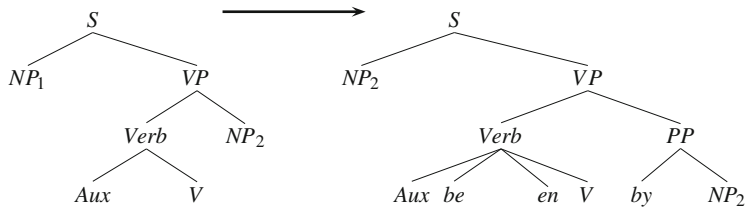


Fig. 11.2 A tree-to-tree mapping representing the active/passive transformational rule

```
T2:  affix, v →
     v, affix, #
```

where # marks a word boundary. Once applied, it yields:

the ball will be hit en # by the boy

hit en # is then rewritten into *hit* by morphophonemic rules. Finally, the transformational process yields:

the ball will be hit by the boy

A tree-to-tree mapping as shown in Fig. 11.2 can also reflect transformation rules. Other common transformations include (Chomsky 1957):

- Negations. *John comes → John doesn't come*.
- Yes/no questions. *they arrive → do they arrive*; *they have arrived → have they arrived*; *they can arrive → can they arrive*; *they arrived → did they arrive*
- Interrogatives. *John ate an apple → did John eat an apple*; *John ate an apple → what did John eat*; *John ate an apple → who ate an apple*
- Conjunction. (*the scene of the movie was in Chicago; the scene of the play was in Chicago*) *→ the scene of the movie and of the play was in Chicago*.
- Topicalization. that is, moving a constituent in front of a sentence to emphasize it. *the waiter brought the meal to the table → to the table, the waiter brought the meal*; *I don't like this meal → this meal, I don't like.*

In Chomsky's formalism, PS rules are written so that certain generated sentences require a transformation to be correct. Such transformations are said to be obligatory. An example is given by the affix permutation rule (T_2). Other rules are optional, such as the passive/active transformation (T_1). PS rules and obligatory transformations account for the "kernel of a language" and generate "kernel sentences". All other sentences can be unfolded and mapped onto this kernel using one or more transformations.

According to Chomsky, transformations simplify the description of a grammar, and make it more compact. Writing a grammar only requires the phrase structure of kernel sentences, and all others are derived from transformations. Later Chomsky related kernel sentences to a deep structure, while transformed sentences correspond to a surface structure. Transformations would then map the surface structure of

Table 11.1 Movements to obtain the passive of sentence *The man hit the ball*. Traces are represented by —. Original positions of traces are in *bold*

Movements	Traces	Passives
First movement	***The man*** *hit* ...	... *— is hit by the man*
Second movement	... *hit* ***the ball***	*The ball is hit —*

Table 11.2 Questions beginning with a *wh*-word and their traces (—)

Questions	Traces
Who ate an apple in the dining room?	*— ate an apple in the dining room*
What did John eat in the dining room?	*John ate — in the dining room*
Which apple did John eat in the dining room?	*John ate — in the dining room*
Where did John eat an apple?	*John ate an apple —*

a sentence onto its deep structure. The deep structure would consist of a set of obligatory transformations and a core phrase structure on which no transformation could be carried out.

11.2.3 Transformations and Movements

Transformation theory evolved into the concept of movement (Chomsky 1981). A movement is a sentence rearrangement where a constituent is moved to another location. The moved constituent leaves a **trace**: an empty symbol representing its initial location. Passives correspond to a composition of two movements: one that moves the subject noun phrase into the position of a prepositional phrase headed by *by*, and another that moves the object noun phrase into the empty subject position (Table 11.1).

Paradigms of movement are questions beginning with an interrogative pronoun or determiner: the *wh*-movements. A *wh*-word – *who, which, what, where* – is moved to the beginning of the sentence to form a question. Consider the sentence *John ate an apple in the dining room.* According to questions and to the *wh*-word type in front of the question, a trace is left at a specific location in the original sentence (Table 11.2). Traces correspond to noun phrases.

Transformations or movements use a syntactic model of both the original phrase – or sentence – and its transformed counterpart. These models form the left and right members of a T-rule. Applying a transformation to a phrase or conversely unfolding a transformation from it, requires knowing its tree structure. In consequence, transformational rules or movements need a prior PS analysis before being applied.

11.2.4 Gap Threading

Gap threading is a technique to parse *wh*-movements (Pereira 1981; Pereira and Shieber 1987). Gap threading uses PS rules that consider the sentence after the movement has occurred. This requires new rules to account for interrogative pronouns or interrogative determiners moved in front of sentence, as for:

John ate an apple
What did John eat?

with a rule to parse the declaration:

```
s --> np, vp.
```

and a new one for the question:

```
s --> [what, did], np, vp.
```

One aim of gap threading is to keep changes in rules minimal. Thus the trace – or **gap** – should be handled by rules similar to those of a sentence before the movement. The rule describing a verb phrase with a transitive verb should remain unchanged:

```
vp --> v, np.
```

with the noun phrase symbol being possibly empty in case of a gap:

```
np --> [].
```

However, such a rule is not completely adequate because it would not differentiate a gap: the absence of a noun phrase resulting from a movement, from the pure absence of a constituent. Rules could insert empty lists wrongly in sentences such as:

John ate

To handle traces properly, gap threading keeps a list of the moved constituents – or **fillers** – as the parsing mechanism reads them. In our example, fillers are *wh*-terms. When a constituent contains a moved term, it is stored in the filler list. When a constituent contains a gap – a missing noun phrase – a term is reclaimed from the head of the filler list.

Gap threading uses two lists as arguments that are added to each constituent of the DCG rules. These lists act as input and output of gaps in the current constituent, as in:

```
s(In, Out) --> np(In, Out1), vp(Out1, Out).
```

At a given point of the analysis, the first list holds fillers that have been stored before, and the second one returns the remaining fillers once gaps have been filled in the constituent.

In the sentence:

What did John eat —?

the verb phrase *eat* — contains a gap. Before processing this phrase, the filler list must have accumulated *what*, which is removed when the verb phrase is completely parsed. Hence, input and output arguments of the vp constituent must be:

```
% vp(In, Out)
vp([what], [])
```

or, to be more general,

```
vp([what | T], T)
```

The noun phrase rule handling the gap accepts no word as an input (because it is a gap). Its right-hand side is then an empty list. The real input is received from the filler list. The rule collects the filler from the first argument of np and returns the resulting list in the second one:

```
np([what | T], T) --> [].
```

The whole set of rules is finally:

```
s(In, Out) -->
  [what, did],
  np([ what | In], Out1),
  vp(Out1, Out).
s(In, Out) --> np(In, Out1), vp(Out1, Out).

np(X, X) --> ['John'].      % no gap here
np(X, X) --> det, n.        % no gap here
np([what | T], T) --> [].   % the gap

vp(In, Out) --> v, np(In, Out).

v --> [eat]; [ate].

det --> [an].

n --> [apple].
```

When parsing a sentence with a movement, initial and final filler lists are set to empty lists:

```
?- s([], [], [what, did, 'John', eat], []).
true
```

as in the initial declaration:

```
?- s([], [], ['John', ate, an, apple], []).
true
```

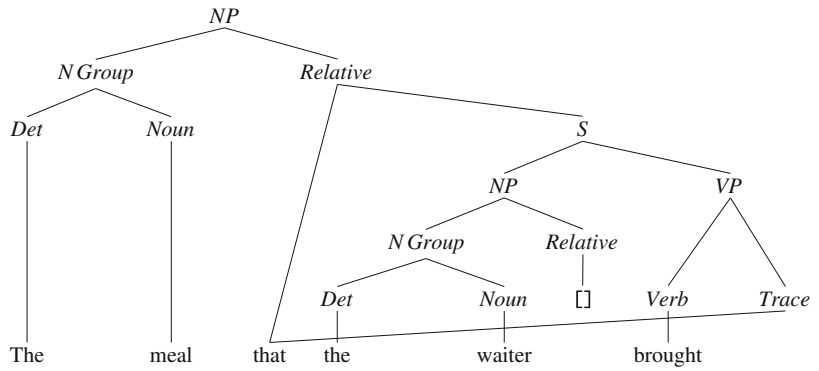


Fig. 11.3 The parse tree of *The meal that the waiter brought* with gap threading

11.2.5 Gap Threading to Parse Relative Clauses

Gap threading can also be used to parse relative clauses. Relative clauses are sentences complementing a noun phrase whose subject or object has been replaced by a relative pronoun. Consider the noun phrase:

The meal that the waiter brought

The rule describing such a phrase is:

```
np --> ngroup, relative
```

where `ngroup` maps *the meal* and `relative` maps *that the waiter brought*.

The modified sentence corresponding to the relative clause here is:

The waiter brought the meal

where the noun phrase *the meal* has been moved from its object position to the front of the relative and has been replaced by the object pronoun *that* (Fig. 11.3). The phrase:

The waiter who brought the meal

is similar, but the movement occurs on the subject noun phrase, which is replaced by subject pronoun *who*.

Let us write a grammar using gap threading to parse such noun phrases. The top rule has two new variables to hold fillers:

```
np(In, Out) --> ngroup(In, Out1), relative(Out1, Out).
```

The relative clause is a sentence that starts with a pronoun, and this pronoun is stored in the filler input list of the sentence symbol:

```
relative(In, Out) --> [that], s([that | In], Out).
relative(In, Out) --> [who], s([who | In], Out).
```

There might also be no relative clause:

```
relative(X, X) --> [].
```

When we encounter a trace, a noun phrase is missing. The head pronoun is then removed from the filler list:

```
np([PRO | T], T) --> [].
```

The rest of the grammar is straightforward:

```
s(In, Out) --> np(In, Out1), vp(Out1, Out).

vp(In, Out) --> v, np(In, Out).

ngroup(X, X) --> det, n.

det --> [the].
n --> [waiter].
n --> [meal].
v --> [brought].
```

When launching the parse, both filler lists are empty:

```
?- np([], [],[the, meal, that, the, waiter, brought],
[]). true

?- np([], [], [the, waiter, who, brought, the, meal],
[]). true
```

In the examples above, we have made no distinction between object and subject pronouns. The program could have been refined to take this difference into account.

11.3 Standardized Phrase Categories for English

The aim of a standard for phrase categories is to define an annotation set that would be common to people working on syntax. Such a standard would facilitate corpus and program sharing, assessment, and communication between computational linguists. Currently, there is no universally accepted standard. Defining an annotation set requires finding a common ground on the structure or the denomination of a specific group of words. It proves to be more difficult than expected. There is a consensus on main categories but details are sometimes controversial.

Most annotation schemes include phrase categories mapping the four main parts of speech, namely nouns, verbs, adverbs, and adjectives. Category names correspond to those of constituent heads:

- **Noun phrases** (NP), phrases headed by a noun.
- **Verb phrases** (VP), phrases headed by a verb together with its objects.

Table 11.3 The Penn Treebank phrase labels (After Marcus et al. (1993))

	Categories	Description
1.	ADJP	Adjective phrase
2.	ADVP	Adverb phrase
3.	NP	Noun phrase
4.	PP	Prepositional phrase
5.	S	Simple declarative clause
6.	SBAR	Clause introduced by subordinating conjunction or 0 (see below)
7.	SBARQ	Direct question introduced by *wh*-word or *wh*-phrase
8.	SINV	Declarative sentence with subject-aux inversion
9.	SQ	Subconstituent of SBARQ excluding *wh*-word or *wh*-phrase
10.	VP	Verb phrase
11.	WHADVP	*wh*-adverb phrase
12.	WHNP	*wh*-noun phrase
13.	WHPP	*wh*-prepositional phrase
14.	X	Constituent of unknown or uncertain category
	Null elements	
1.	*	"Understood" subject of infinitive or imperative
2.	0	Zero variant of *that* in subordinate clauses
3.	T	Trace – marks position where moved *wh*-constituent is interpreted
4.	NIL	Marks position where preposition is interpreted in pied-piping context

- **Adjective phrase** (AdjP), a phrase headed by an adjective, possibly with modifiers.
- **Adverbial phrase** (AdvP), a phrase headed by an adverb.
- Most annotation sets also feature **prepositional phrases** (PP): noun phrases beginning with a preposition.

The Penn Treebank (Marcus et al. 1993) is a corpus annotated with part-of-speech labels. Parts of it are also fully bracketed with syntactic phrase categories, and it was one of the first corpora widely available with such an annotation. Table 11.3 shows its set of phrase labels.

As an example, Fig. 11.4 shows the bracketing of the sentence:

> Battle-tested industrial managers here always buck up nervous newcomers with the tale of the first of their countrymen to visit Mexico, a boatload of samurai warriors blown ashore 375 years ago.

in the Penn Treebank, where `pseudo-attach` denotes an attachment ambiguity for `VP-1`. In effect, *blown ashore* can modify either *boatload* or *samurai warriors*. Both attachments mean roughly the same thing, and there is no way to remove the ambiguity. In this bracketing, *blown ashore* has been attached arbitrarily to *warriors*, and a pseudo-attach has been left to indicate a possible attachment to *boatload*.

Bracketing of phrases is done semiautomatically. A first pass uses an automatic parser. The output is then complemented or corrected by hand by human annotators.

```
( (S
   (NP Battle-tested industrial managers
       here)
    always
   (VP buck
       up
       (NP nervous newcomers)
       (PP with
           (NP the tale
           (PP of
               (NP (NP the
                       (ADJP first
                             (PP of
                                 (NP their countrymen)))
                       (S (NP *)
                          to
                          (VP visit
                              (NP Mexico))))
                   ,
                   (NP (NP a boatload
                           (PP of
                               (NP (NP samurai warriors)
                                   (VP-1 blown
                                       ashore
                                       (ADVP (NP 375 years)
                                             ago)))))
                       (VP-1 *pseudo-attach*)))))))))
.)
```

Fig. 11.4 Bracketed text in the Penn Treebank (After Marcus et al. (1993, p. 325))

11.4 Unification-Based Grammars

11.4.1 Features

In the examples above, there is no distinction between types of noun phrases. They appeared under a unique category: `np`. However, noun phrases are often marked with additional grammatical information, that is, depending on the language, a person, a number, a gender, a case, etc. In German, cases correspond to a specific inflection visible on the surface form of the words (Table 11.4). In English and French, noun phrases are inflected with plural, and in French with gender. We saw in Chap. 6 that such grammatical characteristics are called **features**. Case, gender, or number are features of the noun that are also shared by the components of the noun phrase to which it belongs.

Table 11.4 Inflection imposed to noun group *der kleine Ober* 'the small waiter' by the case feature in German

Cases	Noun groups
Nominative	*der kleine Ober*
Genitive	*des kleinen Obers*
Dative	*dem kleinen Ober*
Accusative	*den kleinen Ober*

If we adopt the generative framework, it is necessary to take features into account to have correct phrases. We can get a picture of this with the German cases and a very simple noun phrase rule:

```
np --> det, adj, n.
```

Since we do not distinguish between `np` symbols, the rule will output ungrammatical phrases as:

```
?-np(L, []).
[der, kleinen, Ober];   %wrong
[der, kleinen, Obers];  %wrong
[dem, kleine, Obers]    %wrong
...
```

To avoid such a wrong generation, we need to consider cases and other features and hence to refine our model. In addition, beyond generation features are necessary in many applications such as spelling or grammar checking, style critique, and so on.

A solution could be to define new noun phrase symbols corresponding to cases such as `np_nominative`, `np_genitive`, `np_dative`, `np_accusative`. We need others to consider number, `np_nominative_singular`, `np_nominative_plural`, ..., and it is not over, because of gender: `np_nominative_singular_masc`, `np_nominative_singular_fem`, etc. This process leads to a division of main categories, such as noun phrases, nouns, and adjectives, into subcategories to account for grammatical features.

11.4.2 Representing Features in Prolog

Creating a new category for each grammatical feature is clumsy and is sometimes useless in applications. Instead, features are better represented as arguments of main grammatical categories. This is straightforward in Prolog using the DCG notation. To account for cases in noun phrases, let us rewrite `np` into:

```
np(case:C)
```

where the `C` value is a member of list `[nom, gen, dat, acc]` denoting nominative, genitive, dative, and accusative cases.

We can extend the number of arguments to cover the rest of the grammatical information. Prolog functors then represent main categories such as noun phrases,

and arguments represent the grammatical details. Arguments are mapped onto feature structures consisting of pairs feature/values as for gender, number, case, person, and type of determiner:

```
np(gend:G, num:N, case:C, pers:P, det:D)
```

Using Prolog's unification, features are easily shared among constituents making up a noun phrase, as in the rule:

```
np(gend:G, num:N, case:C, pers:P, det:D) -->
  det(gend:G, num:N, case:C, pers:P, det:D),
  adj(gend:G, num:N, case:C, pers:P, det:D),
  n(gend:G, num:N, case:C, pers:P).
```

Let us exemplify it with a small fragment of the German lexicon:

```
det(gend:masc, num:sg, case:nom, pers:3, det:def) -->
  [der].
det(gend:masc, num:sg, case:gen, pers:3, det:def) -->
  [des].
det(gend:masc, num:sg, case:dat, pers:3, det:def) -->
  [dem].
det(gend:masc, num:sg, case:acc, pers:3, det:def) -->
  [den].

adj(gend:masc, num:sg, case:nom, pers:3, det:def) -->
  [kleine].
adj(gend:masc, num:sg, case:gen, pers:3, det:def) -->
  [kleinen].
adj(gend:masc, num:sg, case:dat, pers:3, det:def) -->
  [kleinen].
adj(gend:masc, num:sg, case:acc, pers:3, det:def) -->
  [kleinen].

n(gend:masc, num:sg, case:nom, pers:3) --> ['Ober'].
n(gend:masc, num:sg, case:gen, pers:3) --> ['Obers'].
n(gend:masc, num:sg, case:dat, pers:3) --> ['Ober'].
n(gend:masc, num:sg, case:acc, pers:3) --> ['Ober'].
```

To consult this lexicon, Prolog needs a new infix operator ":" that we define using the `op/3` built-in predicate:

```
:- op(600, xfy, ':').
```

And our grammar generates correct noun phrases only:

```
?- np(_, _, _, _, _, L, []).
    L = [der, kleine, 'Ober'] ;
    L = [des, kleinen, 'Obers'] ;
```

```
      L = [dem, kleinen, 'Ober'] ;
      L = [den, kleinen, 'Ober'] .
?-
```

11.4.3 A Formalism for Features and Rules

In the previous section, we directly wrote features as arguments of Prolog predicates. More frequently, linguists use a notation independent of programming languages, which is referred to as unification-based grammars. This notation is close to Prolog and DCGs, however, and is therefore easy to understand. The noun phrase rule

```
np(gend:G, num:N, case:C, pers:P, det:D) -->
  det(gend:G, num:N, case:C, pers:P, det:D),
  adj(gend:G, num:N, case:C, pers:P, det:D),
  n(gend:G, num:N, case:C, pers:P).
```

is represented as:

$$\begin{matrix} NP \\ \begin{bmatrix} gend : G \\ num : N \\ case : C \\ pers : P \\ det : D \end{bmatrix} \end{matrix} \rightarrow \begin{matrix} DET \\ \begin{bmatrix} gend : G \\ num : N \\ case : C \\ pers : P \\ det : D \end{bmatrix} \end{matrix} \begin{matrix} ADJ \\ \begin{bmatrix} gend : G \\ num : N \\ case : C \\ pers : P \\ det : D \end{bmatrix} \end{matrix} \begin{matrix} N \\ \begin{bmatrix} gend : G \\ num : N \\ case : C \\ pers : P \end{bmatrix} \end{matrix}$$

Rules of a grammar describing complete sentences are similar to those of DCGs. They consist, for example, of:

$$S \rightarrow \begin{matrix} NP \\ \begin{bmatrix} num : N \\ case : nom \\ pers : P \end{bmatrix} \end{matrix} \begin{matrix} VP \\ \begin{bmatrix} num : N \\ pers : P \end{bmatrix} \end{matrix}$$

$$\begin{matrix} VP \\ \begin{bmatrix} num : N \\ pers : P \end{bmatrix} \end{matrix} \rightarrow \begin{matrix} V \\ \begin{bmatrix} trans : i \\ num : N \\ pers : P \end{bmatrix} \end{matrix}$$

$$\begin{matrix} VP \\ \begin{bmatrix} num : N \\ pers : P \end{bmatrix} \end{matrix} \rightarrow \begin{matrix} V \\ \begin{bmatrix} trans : t \\ num : N \\ pers : P \end{bmatrix} \end{matrix} \begin{matrix} NP \\ [case : acc] \end{matrix}$$

$$\begin{array}{ccc} NP & \rightarrow & Pronoun \\ \begin{bmatrix} gen : G \\ num : N \\ pers : P \\ case : C \end{bmatrix} & & \begin{bmatrix} gen : G \\ num : N \\ pers : P \\ case : C \end{bmatrix} \end{array}$$

with lexicon entries such as:

$$\begin{array}{cc} DET & \rightarrow der \\ \begin{bmatrix} gend : masc \\ num : sg \\ case : nom \\ det : def \end{bmatrix} & \end{array}$$

11.4.4 Features Organization

A feature structure is a set of pairs consisting of a feature name – or attribute – and its value.

$$\begin{bmatrix} feature_1 : value_1 \\ feature_2 : value_2 \\ \vdots \\ feature_n : value_n \end{bmatrix}$$

Unlike arguments in Prolog or DCGs, the feature notation is based solely on the name and not on the position of the argument. Hence, both

$$\begin{bmatrix} gen : fem \\ num : pl \\ case : acc \end{bmatrix} \text{ and } \begin{bmatrix} num : pl \\ case : acc \\ gen : fem \end{bmatrix}$$

denote the same feature structure. Feature structures can be pictured by a graph as shown in Fig. 11.5.

The value of a feature can be an atomic symbol, a variable, or another feature structure to yield a hierarchical organization as in:

$$\begin{bmatrix} f_1 : v_1 \\ f_2 : \begin{bmatrix} f_3 : v_3 \\ f_4 : \begin{bmatrix} f_5 : v_5 \\ f_6 : v_6 \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

whose corresponding graph is shown in Fig. 11.6.

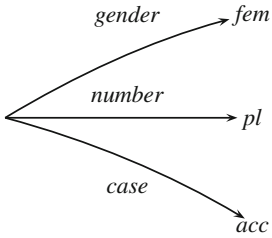


Fig. 11.5 Graph representing a feature structure

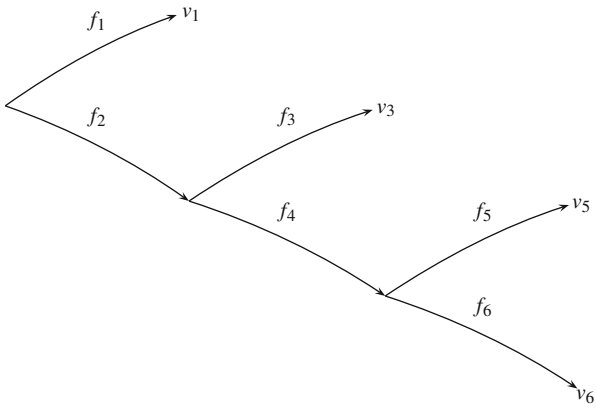


Fig. 11.6 Graph corresponding to embedded feature structures

Grouping a set of features into a substructure enables the simplification of notations or rules. A feature denoted `agreement` can group gender, number, and person, and can be encoded as a single structure. German nominative and accusative pronouns *er* 'he' and *ihn* 'him' can then be represented as:

$$Pronoun \rightarrow er \quad \begin{bmatrix} agreement : \begin{bmatrix} gender : masc \\ number : sg \\ pers : 3 \end{bmatrix} \\ case : nom \end{bmatrix}$$

$$Pronoun \rightarrow ihn \quad \begin{bmatrix} agreement : \begin{bmatrix} gender : masc \\ number : sg \\ pers : 3 \end{bmatrix} \\ case : acc \end{bmatrix}$$

which enables us to simplify the noun phrase rule in:

$$\begin{matrix} NP \\ \begin{bmatrix} agreement : X \\ case : C \end{bmatrix} \end{matrix} \rightarrow \begin{matrix} Pronoun \\ \begin{bmatrix} agreement : X \\ case : C \end{bmatrix} \end{matrix}$$

We can even push categories into structures and rewrite the previous rule as

$$\begin{bmatrix} cat : np \\ agreement : X \\ case : C \end{bmatrix} \rightarrow \begin{bmatrix} cat : pronoun \\ agreement : X \\ case : C \end{bmatrix}$$

Unlike the case for DCGs, unspecified or nonshared features are simply omitted in unification-based grammars. There is no need for an equivalent of the anonymous variable then.

11.4.5 Features and Unification

Unification of feature structures is similar to term unification of Prolog but is more general. It is a combination of two recursive operations:

- Structures merge the set of all their features, checking that identical features have compatible values.
- Variables unify with values and substructures.

Feature structure unification is usually denoted $\cup$.

Unification results in a merger of features as in

$$\begin{bmatrix} feature_1 : v_1 \\ feature_2 : v_2 \end{bmatrix} \cup \begin{bmatrix} feature_2 : v_2 \\ feature_3 : v_3 \end{bmatrix} = \begin{bmatrix} feature_1 : v_1 \\ feature_2 : v_2 \\ feature_3 : v_3 \end{bmatrix}.$$

Variable unification considers features of same name and applies to values, other variables, or recursive feature structures, just as in Prolog but regardless of their position. Here are a couple of examples:

- $[feature_1 : v_1]$ and $[feature_1 : v_2]$ fail to unify if $v_1 \neq v_2$.
- $$\begin{bmatrix} f_1 : v_1 \\ f_2 : X \end{bmatrix} \cup \begin{bmatrix} f_5 : v_5 \\ f_2 : \begin{bmatrix} f_3 : v_3 \\ f_4 : v_4 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} f_1 : v_1 \\ f_2 : \begin{bmatrix} f_3 : v_3 \\ f_4 : v_4 \end{bmatrix} \\ f_5 : v_5 \end{bmatrix}$$
- $$\begin{bmatrix} f_1 : v_1 \\ f_2 : X \end{bmatrix} \cup \begin{bmatrix} f_5 : X \\ f_2 : \begin{bmatrix} f_3 : v_3 \\ f_4 : v_4 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} f_1 : v_1 \\ f_2 : \begin{bmatrix} f_3 : v_3 \\ f_4 : v_4 \end{bmatrix} \\ f_5 : \begin{bmatrix} f_3 : v_3 \\ f_4 : v_4 \end{bmatrix} \end{bmatrix}$$

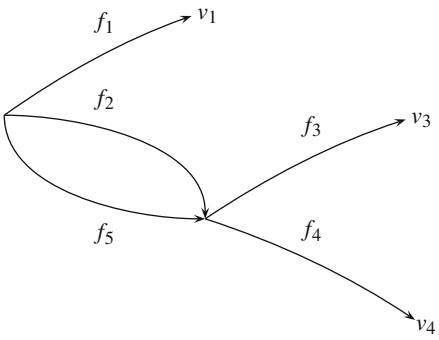


Fig. 11.7 Graph with re-entrant feature structures

In the last example, both features f_2 and f_5 result of the unification of X and are therefore identical. They are said to be re-entrant. However, the structure presentation does not make it clear because it duplicates the X value as many times as it occurs in the structure: twice here. Different structures could yield the same result, as with the unification of

$$\begin{bmatrix} f_1 : v_1 \\ f_2 : \begin{bmatrix} f_3 : v_3 \\ f_4 : v_4 \end{bmatrix} \end{bmatrix} \text{ and } \begin{bmatrix} f_5 : \begin{bmatrix} f_3 : v_3 \\ f_4 : v_4 \end{bmatrix} \\ f_2 : X \end{bmatrix}$$

where feature f_2 and f_5 have (accidentally) the same value.

To improve the structure presentation, identical features are denoted with a label. Here [1] indicates that f_2 and f_5 are the same:

$$\begin{bmatrix} f_1 : v_1 \\ f_2 : [1] \begin{bmatrix} f_3 : v_3 \\ f_4 : v_4 \end{bmatrix} \\ f_5 : [1] \end{bmatrix}$$

and Fig. 11.7 shows the corresponding graph.

11.4.6 A Unification Algorithm for Feature Structures

Unification of feature structures is close to that of terms in Prolog. However, feature structures provide partial specifications of the entities they represent, while Prolog terms are complete. Feature structure unification is merely a union of compatible characteristics, as in the example

$$[case : nom] \cup [gender : masc] = \begin{bmatrix} case : nom \\ gender : masc \end{bmatrix}$$

where both structures merge into a more specific set. As is, corresponding Prolog terms `struct(case: nom)` and `struct(gender: masc)` would fail to unify.

There are possible workarounds. Given a syntactic category, we could itemize all its possible attributes and map them onto a Prolog term. We would have to assign each feature to a specific argument rank, for instance, `case` to the first argument, `gender` to the second one, and so on. We would then fill the specified arguments and leave the others empty using the anonymous variable '_'. For the example above, this would yield terms

```
struct(case: nom, gender:_)
```

and

```
struct(case: _, gender: masc)
```

that unify properly:

```
?- X = struct(case: nom, gender:_), Y =
struct(case: _, gender: masc), X = Y.

X = struct(case: nom, gender: masc)
Y = struct(case: nom, gender: masc)
```

However, when there are many features and hierarchical structures, such a method could be tedious or difficult.

A better idea is to use incomplete lists. Incomplete lists have their tails uninstantiated as `[a, b, c | X]`. Such lists can represent partial structures as `[case: nom | X]` or `[gender: masc | Y]` and be expanded through a Prolog unification. Merging both feature structures is simple. It consists in the unification of `X` with `[gender: masc | Y]`:

```
?- STRUCT = [case: nom | X], X = [gender: masc | Y].
STRUCT = [case: nom, gender: masc | Y]
```

To be more general, we will use the anonymous variable as a tail. Converting a feature structure then consists in representing features as members of a list, where closing brackets are replaced by `| _]`. Hence, structures [*case* : *nom*] and [*gender* : *masc*] are mapped onto `[case: nom | _]` and `[gender: masc | _]`, and their unification yields `[case: nom, gender: masc | _]`. Hierarchical features as:

$$\begin{bmatrix} cat : np \\ agreement : \begin{bmatrix} gender : masc \\ number : sg \\ pers : 3 \end{bmatrix} \\ case : acc \end{bmatrix}$$

are represented by embedded incomplete lists:

```
[cat: np,
agreement: [gender: masc, number: sg, pers: 3 | _],
case: acc | _]
```

Let us now implement the unification algorithm for feature structures due to Boyer (1988). The `unif/2` predicate consists of a fact expressing the end of unification – both structures are the same – and two main rules:

- The first rule considers the case where the heads of both lists represent features of the same name. The rule unifies the feature values and unifies the rest.
- When feature names are different, the second rule uses a double recursion. The first recursion unifies the tail of the first list with the head of the second list. It yields a new list, `Rest3`, which is the unification result minus the head features `F1` and `F2`. The second recursion unifies the rest of the second list with the list made up of the head of the first list and `Rest3`:

```
:- op(600, xfx, ':').

unif(FStr, FStr) :-
  !.
unif([F1:V1 | Rest1], [F1:V2 | Rest2]) :-
  !,
  unif(V1, V2),
  unif(Rest1, Rest2).
unif([F1:V1 | Rest1], [F2:V2 | Rest2]) :-
  F1 \= F2,
  unif(Rest1, [F2:V2 | Rest3]),
  unif(Rest2, [F1:V1 | Rest3]).
```

Consulting `unif/2` and querying Prolog with:

```
?- X = [case: nom | _], Y =[gender: masc | _],
unif(X, Y).
```

results in:

```
X = [case: nom, gender: masc | _]
Y = [gender: masc, case: nom | _]
```

11.5 Dependency Grammars

11.5.1 Presentation

Dependency grammars form an alternative to constituent-based theories. These grammars describe a sentence's structure in terms of syntactic links – connections or dependencies – between its words (Tesnière 1966). Each link reflects a dominance

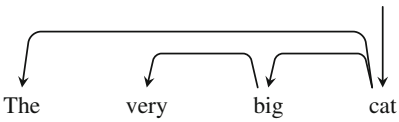


Fig. 11.8 Dependency graph of the noun group *The very big cat*

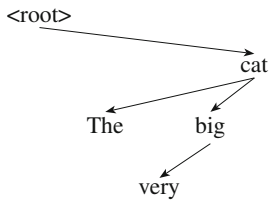


Fig. 11.9 Tree representing dependencies in the noun group *The very big cat*

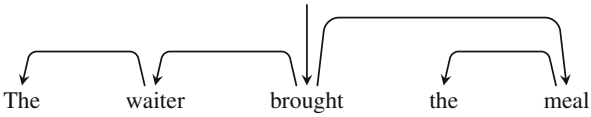


Fig. 11.10 Dependency graph or stemma of the sentence *The waiter brought the meal*

relation (conversely a dependence) between a headword and a dependent word. Examples of simple dependencies tie a determiner to its noun, or a subject noun to its main verb. Dependency links are pictured by arrows flowing from headwords to their dependents (or the reverse).

In noun groups, determiners and adjectives depend on their noun; adverbs depend on their adjective (Fig. 11.8), as in:

The very big cat

where the noun *cat* is the head of *the* and *big*, and the adjective *big* is the head of *very*. In addition, *cat* is the head – or the root – of the whole phrase. Figure 11.9 shows an alternate equivalent representation of the dependencies that uses an explicit `<root>` symbol. In the rest of this book, we will use the convention that `<root>` corresponds to the first word of the sentence at index 0.

According to the classical dependency model, each word is the dependent of exactly one head with the exception of the head of the sentence. Conversely, a head may have several dependents (or modifiers). This means a dependency graph is equivalent to a tree. Figure 11.10 shows a graph representing the structure of a simple sentence where determiners depend on their noun; nouns depend on the main verb, which is the root of the sentence. Tesnière used the word **stemma** – garland or stem in Greek – to name the graphic representation of these links.

Although dependency and constituency are often opposed, stemmas embed sorts of constituents that Tesnière called *nœuds*. Deriving a *nœud* from a dependency graph simply consists in taking a word, all its dependents, and dependents of dependents recursively. It then corresponds to the subtree below a certain word.[1] And in many cases, stemmas and phrase-structure trees yield equivalent structures hinting that dependency and constituency are in fact comparable formalisms.

[1] *Nœud* is the French word for node. It shouldn't be mistaken with a node in a graph, which is a single element. Here a nœud is a whole subtree.

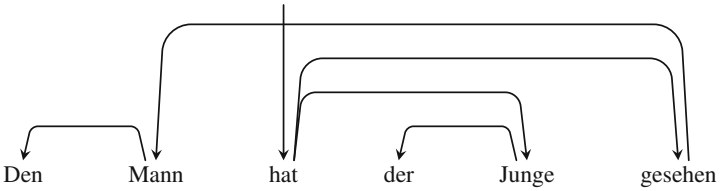


Fig. 11.11 Dependency graph of *Den Mann hat der Junge gesehen* (Modified from Bröker (1998))

There are a couple of differences, however. One is the importance given to words in dependency grammars. There are no intermediate representation symbols such as phrases of constituent grammars. Syntactic relations involve words only, and nodes in stemmas are purely lexical.

Another difference is that dependency grammars do not need a fixed word order or word contiguity in the *nœuds* to establish links. In that sense, dependency theory is probably more suited than constituent grammars to model languages where the word order is flexible. This is the case for Latin, Russian, and German to a lesser extent. Figure 11.11 gives an example with the sentence (Bröker 1998):

Den Mann hat der Junge gesehen
The man$_{/obj}$ *has the boy*$_{/subj}$ *seen* 'The boy has seen the man.'

where positions of noun groups *den Mann* and *der Junge* can be inverted and yield another acceptable sentence: *Der Junge hat den Mann gesehen*. Meaning is preserved because functions of noun groups are marked by cases, nominative and accusative here.

In the example above, stemmas of both sentences are the same, whereas a phrase-structure formalism requires more rules to take the word order variability into account. Modeling the verb phrase needs two separate rules to describe the position of the accusative noun group:

hat den Mann gesehen

and the nominative one:

hat der Junge gesehen

$$VP \rightarrow AUX \quad NP \quad V$$
$$\begin{bmatrix} num : N \\ pers : P \end{bmatrix} \quad \begin{bmatrix} num : N \\ pers : P \end{bmatrix} [case : acc] \begin{bmatrix} tense : pastpart \\ num : N \\ pers : P \end{bmatrix}$$

and

$$VP \rightarrow AUX \quad NP \quad V$$
$$\begin{bmatrix} num : N \\ pers : P \end{bmatrix} \quad \begin{bmatrix} num : N \\ pers : P \end{bmatrix} [case : nom] \begin{bmatrix} tense : pastpart \\ num : N \\ pers : P \end{bmatrix}$$

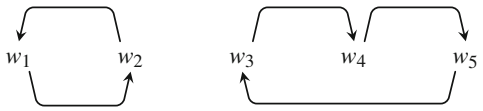


Fig. 11.12 Cyclic dependencies in a graph

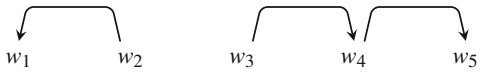


Fig. 11.13 A nonconnected graph spanning sentence $w_1w_2w_3w_4w_5$

When word order shows a high degree of freedom, the constituent structure tends to become combinatorial making grammars resorting on it impracticable. For this reason, many linguists, especially in Europe, believe dependency grammar to be a more powerful formalism than constituency. On the contrary, constituency is a property of English that possibly makes dependency less useful in this language.

11.5.2 Properties of a Dependency Graph

After Tesnière, followers extended or modified the definition of dependency grammars. This has led to variations from the original theory. However, some common principles have emerged from the variety of definitions. We expose here features that are the most widely accepted. They result in constraints on dependency graphs. As for constituent grammars, dependency grammars also received formal mathematical definitions.

The first principle is that dependency graphs are acyclic. This means that there is no loop in the graph. Figure 11.12 shows two structures that are not acceptable.

The second principle is that dependency graphs should be connected. This corresponds to the assumption that a sentence has one single head, the root, to which all the other words are transitively connected. Figure 11.13 shows a sentence $w_1w_2w_3w_4w_5$ with two nonconnected subgraphs.

The third principle, called **projectivity** or **adjacency**, was described by Lecerf and Ihm (1960), when they observed that the set of direct and indirect dependents of a word formed a contiguous sequence – a segment. They defined projectivity as a constraint on the graph where each pair of words $(Dep, Head)$, which are directly connected, is only separated by direct or indirect dependents of $Head$ or Dep. All the words in-between are hence dependents of $Head$. This can be restated more formally as: for all dependency relations in a sentence between a word w_i and its head w_{h_i}, either direction, $w_i \leftarrow w_{h_i}$, respectively $w_{h_i} \rightarrow w_i$, and $\forall w_j$, so that $i < j < h_i$, respectively $h_i < j < i$, w_j is transitively connected to w_{h_i}. In a

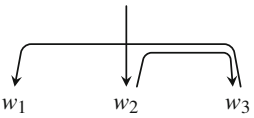


Fig. 11.14 A nonprojective graph. The dependents of w_3 do not form a contiguous sequence (After Lecerf and Ihm (1960))

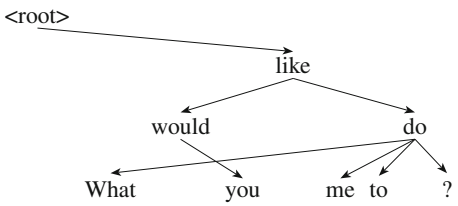


Fig. 11.15 Dependency graph of *What would you like me to do?* (After Järvinen and Tapanainen (1997))

dependency graph, projectivity results in the absence of crossing arcs. Nonprojective graphs are graphs that contain at least one nonprojective link (Fig. 11.14).

The projectivity principle is more controversial that the two first ones. Lecerf and Ihm (1960) assumed that this property was universal. However, although less frequent than projective examples, there are many cases of nonprojective sentences. Figures 11.15 and 11.16 show two examples in English and Latin. The sentence *What would you like me to do?* shows a dependency link between *what* and *do*. The projectivity principle would require that *would*, *you*, and *like* are dependent of *do*, which is untrue. The sentence is thus nonprojective. The Latin verse *Ultima Cumaei venit iam carminis aetas* 'The last era of the Cumean song has now arrived' shows a dependency link between *carminis* and *Cumaei*, but neither *venit* nor *iam* are dependent of *carminis*.

As shown by these examples, instead of considering projectivity as a universal principle, we can better reformulate it as a frequent property that suffers exceptions.

11.6 Differences Between Tesnière's Model and Current Conventions in Dependency Analysis

Current conventions in dependency analysis are not completely unified and can diverge significantly from Tesnière's model. In this section, we discuss two features that show a notable variation according to annotation styles and syntactic parsers, namely the analysis of prepositions and auxiliaries, and coordination.

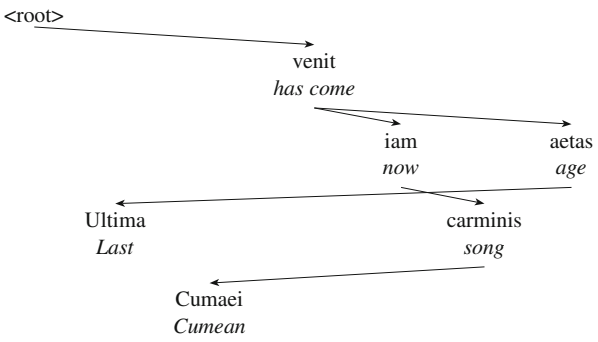


Fig. 11.16 Dependency graph of *Ultima Cumaei venit iam carminis aetas*. 'The last era of the Cumean song has now arrived' (Virgil, *Eclogues* IV.4) (After Covington (1990))

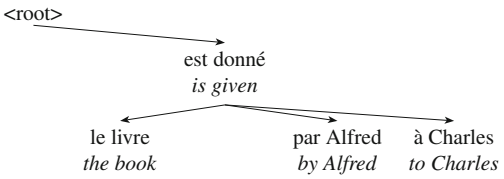


Fig. 11.17 Dependency graph of *Le livre est donné par Alfred à Charles* 'The book is given by Alfred to Charles' (After Tesnière (1966), stemma 96)

11.6.1 Prepositions and Auxiliaries

In Tesnière's stemmas (graphs), prepositions and auxiliaries are treated more or less as sorts of morphemes of either their object noun or their main verb. Both the preposition and its object noun and the auxiliary and its main verb are gathered in the same node. Figure 11.17 shows an example of it with the sentence *Le livre est donné par Alfred à Charles* 'The book is given by Alfred to Charles,' where the two words of the verb group *est donné* 'is given' form a unique node. Tesnière also sets some determiners – the definite articles – as part of their noun's node, as the *le livre* 'the book' in Fig. 11.17.

In the current conventions, dependency graphs have only one word per node. This means that for verb groups consisting of an auxiliary and a main verb, one of the two words will be the head and the other one, a dependent. This then yields two possible annotations. In addition, depending on the convention, the verb's subject, object, or adjuncts will either attach to the auxiliary or to the main verb. Figures 11.18 and 11.19 show four different possible graphs for the sentence *I have eaten the meal*. As the annotation must be uniform across a given corpus, the annotators have to decide on one convention. The choice is often arbitrary and is usually explained in the annotation guidelines of each corpus.

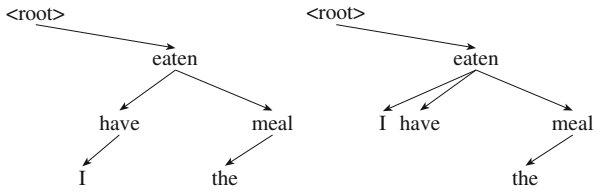


Fig. 11.18 Dependency graph of *I have eaten the meal* with different conventions to relate an auxiliary to its main verb. Here the main verb is the head of its auxiliary

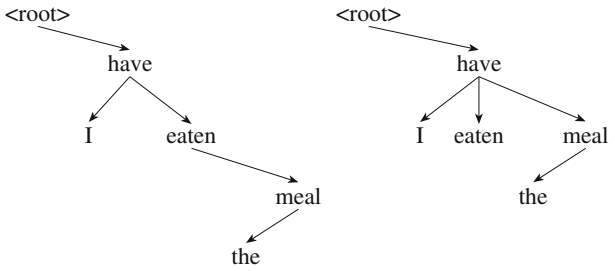


Fig. 11.19 Dependency graph of *I have eaten the meal* with different conventions to relate an auxiliary to its main verb. Here the auxiliary is the head of its main verb

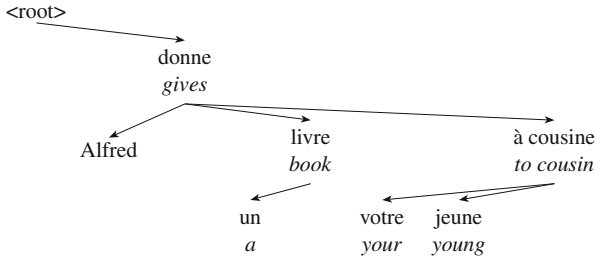


Fig. 11.20 Dependency graph of *Alfred donne un livre à votre jeune cousine* 'Alfred gives a book to your young cousin.' The preposition *à* and its object noun *cousine* are in the same node (After Tesnière (1966), stemma 135)

For auxiliaries, we can justify Tesnière's style by comparing the construction of the future tense between French, on one side, and English and German, on the other side. In French, future is rendered by specific suffixes that we have seen in Chap. 6. In English and German, future uses an auxiliary, *will* or *werden*. This means that with the current annotation conventions, for a same word, we will have two nodes in English and German and only one in French. Using one node, as Tesnière did, was a way to generalize the annotation and make it less dependent on the language.

A similar treatment applies to prepositions that Tesnière placed in the same node as their object noun. Figure 11.20 shows the graph of the sentence *Alfred donne un livre à votre jeune cousine* 'Alfred gives a book to your young cousin.' Current

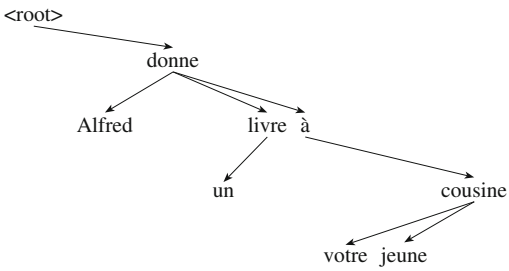


Fig. 11.21 Dependency graph of *Alfred donne un livre à votre jeune cousine* 'Alfred gives a book to your young cousin', where the preposition is the head of its object noun

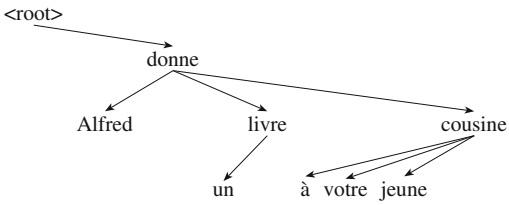


Fig. 11.22 Dependency graph of *Alfred donne un livre à votre jeune cousine* 'Alfred gives a book to your young cousin', where the noun is the head of its preposition

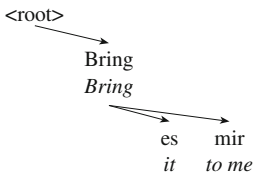


Fig. 11.23 Dependency graph of the German sentence *Bring es mir* 'Bring it to me'

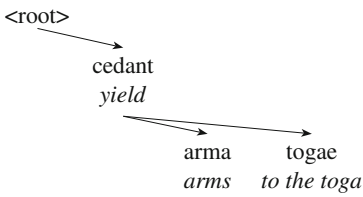


Fig. 11.24 Dependency graph of the Latin sentence *cedant arma togae* 'let arms yield to the toga', Cicero, *De Officiis*, I, 22

annotation conventions would either make the preposition, here *à* 'to', the head of the prepositional phrase (Fig. 11.21) or would use the main noun, here *cousine* 'cousin' (Fig. 11.22).

Tesnière's stemma follows a tradition of German linguists, where prepositions are considered as case markers. See Hjelmslev (1935–1937) for a discussion. In the German sentence *Bring es mir* 'Bring it to me', the English phrase *to me* is rendered by the German word *mir*, which corresponds to the word *mich* 'me' in the dative case (Fig. 11.23). The preposition *to* would then act as an equivalent of the dative case in this sentence. A similar example is given by the Latin sentence *cedant arma togae* 'let arms yield to the toga' (let the military power give way to civilian power), where the word *togae* is the inflection of *toga*, a clothing of ancient Rome, in the dative case (Fig. 11.24).

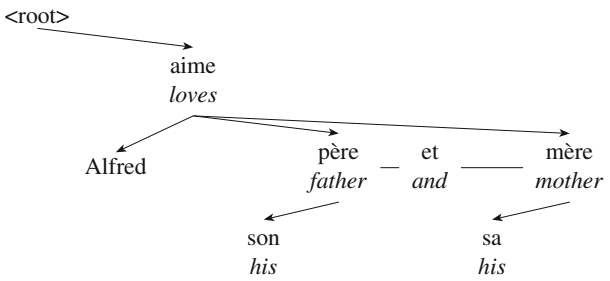


Fig. 11.25 Dependency graph of *Alfred aime son père et sa mère* 'Alfred loves his father and his mother' (After Tesnière (1966), stemma 251)

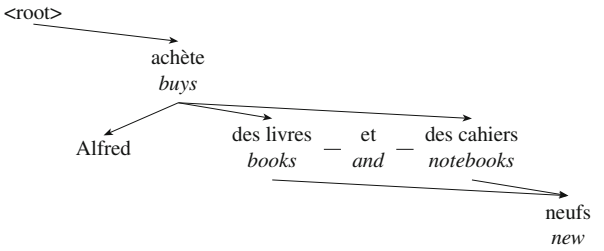


Fig. 11.26 Dependency graph of *Alfred achète des livres et des cahiers neufs* 'Alfred buys new books and notebooks' (After Tesnière (1966), stemma 255)

The decision on which style to adopt with auxiliaries as well as with prepositions has sometimes practical consequences when parsing. In general, they are, however, limited because the graphs are roughly equivalent. We summarize these consequences here. As prepositions are very informative on which word the prepositional phrase will attach, most annotation standards prefer to have them as the head of their object noun. For auxiliaries, using the main verb as the head gives probably more information on its dependents than auxiliaries, but this can create more nonprojective links that are a problem to many parsers.

11.6.2 Coordination and Apposition

Differently to prepositions and auxiliaries, Tesnière's treatment of coordination is radically different from what can be found in the current annotation practices. In his stemmas, Tesnière uses two different types of links to denote, respectively, dependencies and conjunctions, where dependencies are directed arcs whereas conjunctions are not. Figures 11.25 and 11.26 show the annotation of the sentences *Alfred aime son père et sa mère* 'Alfred loves his father and his mother' and *Alfred achète des livres et des cahiers neufs* 'Alfred buys new books and notebooks.' In the

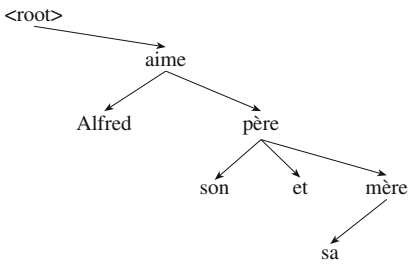


Fig. 11.27 Dependency graph of *Alfred aime son père et sa mère* 'Alfred loves his father and his mother.' The first conjunct is the head of the conjunction and the second conjunct

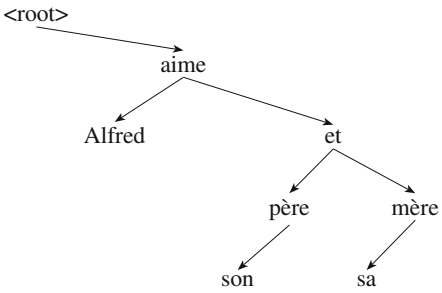


Fig. 11.28 Dependency graph of *Alfred aime son père et sa mère* 'Alfred loves his father and his mother.' The conjunction is the head of the conjuncts

latter sentence, the adjective *neufs* 'new' has two heads, which departs from nearly all current annotation conventions.

In current dependency analysis, there is no difference between the two types of links that are merged as dependencies. One must then decide which word will be the head in coordinated structures. As for prepositions and auxiliaries, two different conventions are used that either make one of the conjuncts (Fig. 11.27) or the conjunction (Fig. 11.28), the head of the coordinated structure.

From these two possibilities, choosing a conjunct as the head is probably a better option because it exhibits the lexical links between the head and the dependent, as in the sentence *Alfred aime son père et sa mère* (Fig. 11.27), between *aime* 'loves' and *père* 'father', unless the annotators wish to have a symmetrical presentation of the graphs.

A last example of tricky representation of conjunctions is given by the sentence *L'un portait sa cuirasse, l'autre son bouclier* 'The one carried his cuirass, the other his buckler', where the verb is shared by the two members of the sentence: *l'un sa cuirasse* and *l'autre son bouclier*. Tesnière represents the conjunction with a duplicated verb node as shown in Fig. 11.29.

11.7 Valence

Tesnière and others stressed the importance of verbs in European languages: main verbs have the highest position in the node hierarchy and are the structural centers of sentences. All other types of phrases are organized around them. Hence verbs tend to

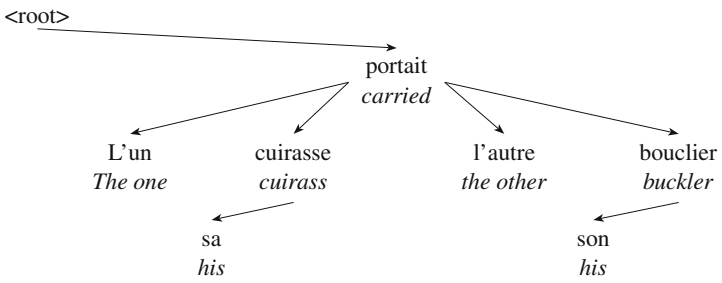


Fig. 11.29 Dependency graph of *L'un portait sa cuirasse, l'autre son bouclier* 'The one carried his cuirass, the other his buckler.' *Chanson de Malbrough* (After Tesnière (1966), stemma 273)

impose a certain structure to sentences. Connected to this observation, a major claim of dependency grammars is that verbs have specific complement patterns. Verb complements in a broad sense include groups that accompany it: subject, objects, and adjuncts.

Different verbs combine differently with their complements. Certain complements are essential to a verb, like its subject, most of the time. Essential complements cannot be removed from the sentence without making it incorrect or incomplete. Other complements are optional – or circumstantial – like adjuncts that give information on space, time, or manner. Removing them would modify the meaning of the sentence but would still result into something acceptable.

The valence is the number of essential complements of a verb. Using an analogy with chemistry, valence is the attraction power of a verb for potential complements and a specific property of each verb. Just as for chemical elements, the valence is not a strict requirement but rather reflects a sort of most current, stable construction. Common valence values are (Table 11.5):

- 0, for verbs describing weather, *it's raining, snowing*
- 1, corresponding to the subject of intransitive verbs, *he's sleeping, vanishing*
- 2, the subject and object of transitive verbs, *she read this book*.
- 3, the subject and two objects – direct and indirect objects – of ditransitive verbs, *Elke gave a book to Wolfgang, I said it to my sister*.
- 4, the subject, object, source, and destination of certain verbs like *move* or *shift: I moved the car from here to the street* (Heringer 1993).

From a quantitative definition: the number of slots or arguments attached to a verb and filled with its essential complements, valence is also frequently extended to cover qualitative aspects. It includes the grammatical form and the meaning of these slots. Grammatical properties include possible prepositions and syntactic patterns allowed to each complement of a verb: noun group, gerund, or infinitive. Many dictionaries, especially learners' dictionaries, itemize these patterns, also referred to as **subcategorization frames**. Tables 11.6–11.8 summarize some verb–complement structures.

Table 11.5 Valence values and examples, where $iobject$ denotes the indirect object

Valences	Examples	Frames
0	*it's raining*	*raining* []
1	*he's sleeping*	*sleeping* [*subject* : *he*]
2	*she read this book*	*read* $\begin{bmatrix} subject : she \\ object : book \end{bmatrix}$
3	*Elke gave a book to Wolfgang*	*gave* $\begin{bmatrix} subject : Elke \\ object : book \\ iobject : Wolfgang \end{bmatrix}$
4	*I moved the car from here to the street*	*moved* $\begin{bmatrix} subject : I \\ object : car \\ source : here \\ destination : street \end{bmatrix}$

Table 11.6 Verb–complement structures in English

Verb	Complement structure	Example
slept	None (Intransitive)	*I slept*
bring	NP	*The waiter brought the meal*
bring	NP + to + NP	*The waiter brought the meal to the patron*
depend	on + NP	*It depends on the waiter*
wait	for + NP + to + VP	*I am waiting for the waiter to bring the meal*
keep	VP(ing)	*He kept working*
know	that + S	*The waiter knows that the patron loves fish*

Table 11.7 Verb–complement structures in French

Verb	Complement structure	Example
dormir	None (Intransitive)	*J'ai dormi*
apporter	NP (Transitive)	*Le serveur a apporté un plat*
apporter	NP + à + NP	*Le serveur a apporté un plat au client*
dépendre	de + NP	*Ça dépend du serveur*
attendre	que + S(Subjunctive)	*Il a attendu que le serveur apporte le plat*
continuer	de + VP(INF)	*Il a continué de travailler*
savoir	que + S	*Le serveur sait que le client aime le poisson*

Table 11.8 Verb–complement structure in German

Verb	Complement structure	Example
schlafen	None (Intransitive)	*Ich habe geschlafen*
bringen	NP(Accusative)	*Der Ober hat eine Speise gebracht*
bringen	NP(Dative) + NP(Accusative)	*Der Ober hat dem Kunde eine Speise gebracht*
abhängen	von + NP(Dative)	*Es hängt vom Ober ab*
warten	auf + S	*Er wartete auf dem Ober, die Speise zu bringen*
fortsetzen	NP	*Er hat die Arbeit fortgesetzt*
wissen	NP(Final verb)	*Der Ober weiß, das der Kunde Fisch liebt*

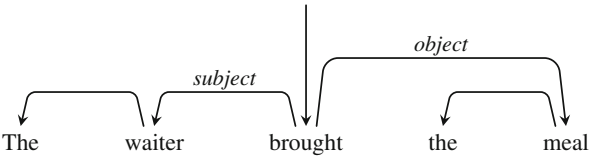


Fig. 11.30 Dependency graph of the sentence *The waiter brought the meal*

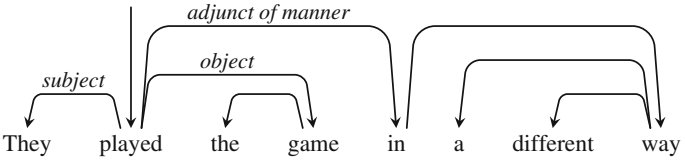


Fig. 11.31 Dependency graph of the sentence *They played the game in a different way* (After Järvinen and Tapanainen (1997))

In addition, typical complements of a verb often belong to broad semantic categories. The verb *read* generally involves a person as a subject and a written thing as an object. This extension of valence to the semantic domain is called the selectional restrictions of a verb and is exemplified by the frame structure of *gave*:

$$gave \begin{bmatrix} subject : PERSON \\ object : THING \\ iobject : PERSON \end{bmatrix}$$

Chapter 15 gives more details on this aspect.

11.8 Dependencies and Functions

The dependency structure of a sentence – the stemma – generally reflects its traditional syntactic representation, and therefore its links can be annotated with function labels. In a simple sentence, the two main functions correspond to subject and object relations that link noun groups to the sentence's main verb (Fig. 11.30).

Adjuncts form another class of functions that modify the verb they are related to. They include prepositional phrases whose head is set arbitrarily to the front preposition (Fig. 11.31). In the same way, adjuncts include adverbs that modify a verb (Fig. 11.32).

As for phrase categories in constituent grammars, a fixed set of function labels is necessary to annotate stemmas. Tables 11.9 and 11.10 reproduce the set of dependency functions proposed by Järvinen and Tapanainen (1997). Figures 11.33 and 11.34 show examples of annotations.

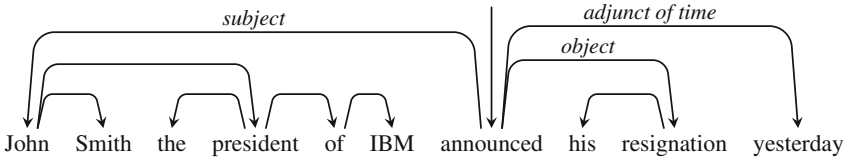


Fig. 11.32 Dependency graph of the sentence *John Smith, the president of IBM, announced his resignation yesterday* (After Collins (1996))

Table 11.9 Main functions used by Järvinen and Tapanainen (1997) in their functional dependency parser for English. Intranuclear links combine words inside a *nœud* (a constituent). Verb complementation links a verb to its core complements. Determinative functions generally connect determiners to nouns. Modifiers are pre- or postmodifiers of a noun, i.e., dependents of a noun before or after it

Name	Description	Example
Main functions		
main	Main element, usually the verb	*He doesn't* ***know*** *whether to send a gift*
qtag	Question tag	*Let's play another game,* ***shall*** *we?*
Intranuclear links		
v-ch	Verb chain, connects elements in a complex verb group	*It* ***may have been being*** *examined*
pcomp	Prepositional complement, connects a preposition to the noun group after it.	*They played the game* ***in*** *a different way*
phr	Verb particle, connects a verb to a particle or preposition.	*He asked me who would look* ***after*** *the baby*
Verb complementation		
subj	Subject	
obj	Object	*I gave him my* ***address***
comp	Subject complement, the second argument of a copula.	*It has become* ***marginal***
dat	Indirect object	*Pauline gave it* ***to*** *Tom*
oc	Object complement	*His friends call him* ***Ted***
copred	Copredicative	*We took a swim* ***naked***
voc	Vocative	*Play it again,* ***Sam***
Determinative functions		
qn	Quantifier	*I want* ***more*** *money*
det	Determiner	***Other*** *members will join...*
neg	Negator	*It is* ***not*** *coffee that I like, but tea*
Modifiers		
attr	Attributive nominal	***Knowing*** *no French, I couldn't express my thanks*
mod	Other postmodifiers	*The baby, Frances* ***Bean****, was...* *The people* ***on*** *the bus were singing*
ad	Attributive adverbial	*She is* ***more*** *popular*
Junctives		
cc	Coordination	*Two* ***or*** *more cars...*

Table 11.10 Adverbial functions used by Järvinen and Tapanainen (1997). Adverbial functions connect adjuncts to their verb

Name	Description	Example
Adverbial functions		
tmp	Time	*It gives me very great pleasure this **evening***
dur	Duration	*They stay in Italy all **summer** through*
frq	Frequency	*I **often** catch her playing*
qua	Quantity	*It weighed almost a **ton***
man	Manner	*They will support him, however **grudgingly**...*
loc	Location	*I don't know **where** to meet him*
sou	Source	*They traveled slowly **from** Hong Kong*
goa	Goal	*They moved **into** the kitchen every stick of furniture they possessed*
cnd	Condition	*If I were **leaving**, you should know about it*
meta	Clause adverbial	*Will somebody **please** open the door?*
cla	Clause initial element	***In** the view of the authorities, Jones was...*

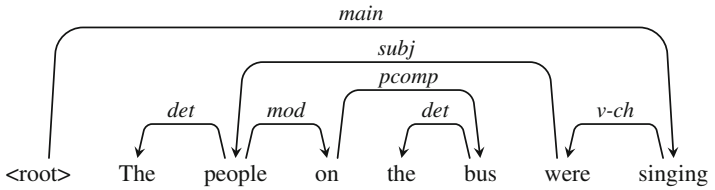


Fig. 11.33 Graph representing *The people on the bus were singing* (After Järvinen and Tapanainen (1997))

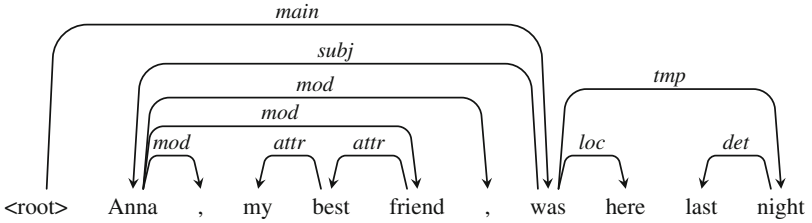


Fig. 11.34 Stemma representing *Anna, my best friend, was here last night* (After Järvinen and Tapanainen (1997))

11.9 Corpus Annotation for Dependencies

The dependency graph of a sentence of n words is compactly expressed by the sequence:

$$D = \{< \text{Head}(1), \text{Rel}(1) >, < \text{Head}(2), \text{Rel}(2) >, \ldots, < \text{Head}(n), \text{Rel}(n) >\},$$

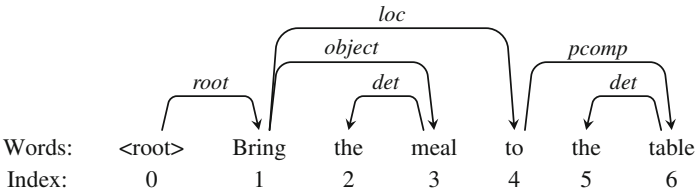


Fig. 11.35 Dependency graph of the sentence *Bring the meal to the table*. <root> is at index 0

Table 11.11 A representation of dependencies due to Lin (1995). *Direction* gives the direction of the head. Symbol '>' means that it is the first occurrence of the word to the right, '>>' the second one, etc. '*' denotes the root of the sentence

Index	Word	Direction	Head	Head index	Function
1	*Bring*	*		Root	Main verb
2	*the*	>	*meal*	3	Determiner
3	*meal*	<	*Bring*	1	Object
4	*to*	<	*Bring*	1	Location
5	*the*	>	*table*	6	Determiner
6	*table*	<	*to*	4	Prepositional complement

which maps each word of index i to its head, Head(i), with the relation Rel(i). The head is defined by its position index in the sentence. By convention, we set the position of the root to index 0 in the sentence.

The representation of the sentence *Bring the meal to the table*, whose graph is shown in Fig. 11.35 is:

$$D = \{< 0, \text{root} >, < 3, \text{det} >, < 1, \text{object} >, < 1, \text{loc} >, < 6, \text{det} >, < 4, \text{pcomp} >\},$$

where $< 0, \text{root} >$ denotes the root of the dependency graph.

The sentence structure is probably easier to read if we use a tabular format. Table 11.11 is a representation due to Lin (1995), where the words are in the first column, one word per row. On each row, we indicate the head word and its grammatical function in the last column.

11.9.1 Dependency Annotation Using XML

MALT XML is a representation format created to annotate the Talbanken corpus in Swedish (Einarsson 1976; Nilsson et al. 2005). The corpus is organized as a sequence of sentences, where each sentence consists of a sequence of words. The annotation uses XML and has two main elements: `sentence` and `word`. The sentence element has one main attribute, `id`, that denotes the sentence number, and the word element has five attributes:

- `id`, a unique, sequential identifier within the sentence.
- `form`, the word form.
- `pos`, the word part of speech followed by it grammatical features using the Stockholm–Umeå tagset (Ejerhed et al. 1992), see Chap. 7, Sect. 7.5.3.
- `head`, the head position using `id`.
- `deprel`, the grammatical relation between the word and its head.

The representation is based on the assumption that each word has at most one head. By convention, the word `id`s start at 1, and a root word has dummy head located at index 0 with the `ROOT` grammatical function. A dependency tree for the Swedish sentence *Dessutom höjs åldersgränsen till 18 år* 'In addition, the age limit is raised to 18' can be represented as follows:

```
<sentence id="24">
  <word id="1" form="Dessutom" postag="ab" head="2"
    deprel="ADV"/>
  <word id="2" form="höjs" postag="vb.prs.sfo"
    head="0" deprel=""/>
  <word id="3" form="åldergränsen"
    postag="nn.utr.sin.def.nom" head="2"
    deprel="SUB"/>
  <word id="4" form="till" postag="pp" head="2"
    deprel="ADV"/>
  <word id="5" form="18" postag="rg.nom" head="6"
    deprel="DET"/>
  <word id="6" form="år" postag="nn.neu.plu.ind.nom"
    head="4" deprel="PR"/>
  <word id="7" form="." postag="mad" head="2"
    deprel="IP"/>
</sentence>
```

11.9.2 The CoNLL Annotation

In 2006 and 2007, the conferences on natural language learning (CoNLL) organized shared tasks to measure the performance of dependency parsers. They used a tabular annotation that is now a widely accepted standard in the field (Buchholz and Marsi 2006).

In the corpora using this annotation, the rows represent sequential tokens, one token per row, and the columns correspond to the token annotation. Each sentence is followed by one single blank line, and the columns are separated by single tabulations. The encoding format is UTF-8. Table 11.12 shows a description of the ten columns, called fields in the CoNLL tasks and Tables 11.13, and 11.14 show annotation examples in German and Italian.

Table 11.12 The CoNLL annotation scheme with the column names and descriptions. When unavailable in the corpus, the cell is replaced with an underscore character (Buchholz and Marsi 2006). The tagsets for parts of speech, morphological features, and functions are all language dependent

#	Name	Description
1	ID	Token index, starting at 1 for each sentence.
2	FORM	Word form or punctuation.
3	LEMMA	Lemma or stem.
4	CPOSTAG	Part-of-speech tag.
5	POSTAG	Fine-grained part-of-speech tag.
6	FEATS	Unordered set of morphological features separated by a vertical bar (|).
7	HEAD	Head of the current token, which is either a value of ID or zero (0) if this is the root.
8	DEPREL	Dependency relation to the HEAD.
9	PHEAD	Projective head of current token, which is either a value of ID or zero (0). The dependency structure resulting from the PHEAD column is guaranteed to be projective, when available in the corpus.
10	PDEPREL	Dependency relation to the PHEAD.

Table 11.13 Dependency annotation of the German sentence *Um ihn dennoch anzuschieben, wollte sich die Privatwirtschaft erstmals "in erheblichem Umfang an den Risiken" der Investition beteiligen* 'Nevertheless in order to push it, the private sector wanted to take part for the first time "to a substantial risk extension" in the investment,' which exhibits a nonprojective structure. Example from CoNLL-X (Brants et al. 2002). In this corpus, the lemmas as well as the features are not available

1	Um	_	KOUI	KOUI	_	4	CP	4	CP
2	ihn	_	PPER	PPER	_	4	OA	4	OA
3	dennoch	_	ADV	ADV	_	4	MO	4	MO
4	anzuschieben	_	VVIZU	VVIZU	_	6	MO	6	MO
5	,	_	$,	$,	_	6	PUNC	6	PUNC
6	wollte	_	VMFIN	VMFIN	_	0	ROOT	0	ROOT
7	sich	_	PRF	PRF	_	21	OA	6	OA
8	die	_	ART	ART	_	9	NK	9	NK
9	Privatwirtschaft	_	NN	NN	_	6	SB	6	SB
10	erstmals	_	ADV	ADV	_	6	MO	6	MO
11	"	_	$(	$(	_	6	PUNC	6	PUNC
12	in	_	APPR	APPR	_	21	MO	21	MO
13	erheblichem	_	ADJA	ADJA	_	12	NK	12	NK
14	Umfang	_	NN	NN	_	12	NK	12	NK
15	an	_	APPR	APPR	_	21	OP	21	OP
16	den	_	ART	ART	_	15	NK	15	NK
17	Risiken	_	NN	NN	_	15	NK	15	NK
18	"	_	$(	$(	_	15	PUNC	15	PUNC
19	der	_	ART	ART	_	20	NK	20	NK
20	Investition	_	NN	NN	_	15	AG	15	AG
21	beteiligen	_	VVINF	VVINF	_	6	OC	6	OC
22	.	_	$.	$.	_	6	PUNC	6	PUNC

Table 11.14 Dependency annotation of the Italian sentence *Non ci rendiamo conto del lavoro psicologico, dei prodigi di equilibrio, di diplomazia che fanno per noi* 'We do not realize the psychological work, the prodigious balancing acts, the diplomacy, they do for us.' Example from CoNLL 2007 (Montemagni et al. 2003). In this corpus, the projectivized structure is not available

1	Non	non	B	B	_	3	mod	_	_
2	ci	ci	P	PQ	gen=N|num=P|per=1	3	clit	_	_
3	rendiamo	rendere	V	V	num=P|per=1|mod=I|tmp=P	0	ROOT	_	_
4	conto	conto	S	S	gen=M|num=S	3	ogg_d	_	_
5	del	di	E	E	gen=M|num=S	4	mod	_	_
6	lavoro	lavoro	S	S	gen=M|num=S	5	prep	_	_
7	psicologico	psicologico	A	A	gen=M|num=S	6	mod	_	_
8	,	,	PU	PU	_	5	con	_	_
9	dei	di	E	E	gen=M|num=P	5	cong	_	_
10	prodigi	prodigio	S	S	gen=M|num=P	9	prep	_	_
11	di	di	E	E	_	10	mod	_	_
12	equilibrio	equilibrio	S	S	gen=M|num=S	11	prep	_	_
13	,	,	PU	PU	_	11	con	_	_
14	di	di	E	E	_	11	cong	_	_
15	diplomazia	diplomazia	S	S	gen=F|num=S	14	prep	_	_
16	che	che	P	PR	gen=N|num=N	17	ogg_d	_	_
17	fanno	fare	V	V	num=P|per=3|mod=I|tmp=P	6	mod_rel	_	_
18	per	per	E	E	_	17	mod	_	_
19	noi	noi	P	PQ	gen=N|num=P|per=1	18	prep	_	_
20	.	.	PU	PU	_	19	punc	_	_

11.10 Projectivization

In Sect. 11.5.2, we introduced the projective property of dependency graphs. Although nonprojectivity is more an exception than the rule, recent corpus studies showed that nonprojective links were from 0.1 to 5.4 % of the total number of arcs, depending on the language (Buchholz and Marsi 2006). This can seem negligible. However, when we consider the whole graph instead, this signifies that in some languages like Dutch, up to a third of the sentences contain one or more nonprojective links.

This has serious consequences as some parsers only accept projective structures. It is therefore important to be able to identify nonprojective links and sentences, which we do in the next section with a Prolog program. This would not be a big gain if we were not able to propose a solution for them. In fact, it is possible to approximate nonprojective structures with a projective equivalent, which allows parsers to analyze them. It is also possible to mark "projectivized" graphs with backtraces and retrieve the nonprojective originals from them. Finally, it can be demonstrated that projective dependency graphs and constituent parse trees are weakly equivalent.

11.10.1 A Prolog Program to Identify Nonprojective Arcs

We saw that projective arcs corresponded to contiguous segments (Fig. 11.14). This means that inside a segment defined by w_i, a word of index i, and w_{h_i}, its head of index h_i, all the words are transitively connected to w_{h_i}. The identification of nonprojective arcs is carried out using the negation of this property. A dependency arc (i, h_i) is nonprojective if there is at least one word w_j which has its index inside the range $i..h_i$ and none of this word's transitive heads is w_{h_i}.

Here is the Prolog program that implements the definition. We encode dependencies as a list of words `w/1` with a subset of features taken from the CoNLL format:

```
w([id=ID, form=FORM, head=HEAD, deprel=DEPREL])
```

where `id` is the position of the word, and `head`, the position of its head.

```
% nonproj_links(+DepGraph, -NPL) returns
nonprojective links.
% DepGraph is the set of dependency relations and NPL,
% the nonprojective links.

nonproj_links(DepGraph, NPL) :-
  nonproj_links(DepGraph, DepGraph, NPL).

nonproj_links(_, [], []).
nonproj_links(DepGraph, [Arc | T], NPL) :-
  range(Arc, MIN, MAX),
  proj_in_range(DepGraph, Arc, MIN, MAX),
  !,
  nonproj_links(DepGraph, T, NPL).
nonproj_links(DepGraph, [Arc | T], [Arc | NPL]) :-
  !,
  nonproj_links(DepGraph, T, NPL).

% range(+Arc, -MIN, +MAX) finds the range of Arc
range(w(Arc), MIN, MAX) :-
  member(id=DepPos, Arc),
  member(head=HeadPos, Arc),
  MIN is min(DepPos, HeadPos),
  MAX is max(DepPos, HeadPos).

% proj_in_range(+DepGraph, +Arc, +MIN, +MAX) succeeds
if all
% the arcs inside Arc are transitively  connected to
its head
% or fails otherwise
proj_in_range(DepGraph, Arc, MIN, MAX) :-
```

```
    findall(
      w([id=DepPos, form=Dep, head=HeadPos]),
      (
        member(w([id=DepPos, form=Dep, head=HeadPos
        | _]), DepGraph),
        DepPos > MIN,
        DepPos < MAX
      ),
      InRange),
    head_chains(DepGraph, InRange, Arc).

  % head_chains(+DepGraph, +ArcsInRange, +Arc) succeeds
  % if all the arcs in ArcsInRange are transitively
  % connected to Arc
  head_chains(_, [], _).
  head_chains(DepGraph, [w([id=DepPosIR, form=DepIR,
        head=HeadPosIR]) | T],
        w([id=DepPos, form=Dep, head=HeadPos | _])) :-
    !,
    chain(DepGraph,
          w([id=DepPosIR, form=DepIR, head=HeadPosIR]),
          w([id=HeadPos, form=_, head=_])),
    head_chains(DepGraph, T,
          w([id=DepPos, form=Dep, head=HeadPos | _])).

  % chains(+DepGraph, +Arc, +Head) succeeds
  % if Arc is transitively connected to Head
  chain(_, w([id=_, form=_, head=HeadPos]),
        w([id=HeadPos, form=_, head=_])) :- !.
  chain(DepGraph, w([id=_, form=_, head=HeadPosIR]),
        w([id=HeadPos, form=_, head=_])) :-
    member(
          w([id=HeadPosIR, form=HeadIR, head=HHPosIR
          | _]), DepGraph),
    chain(DepGraph,
          w([id=HeadPosIR, form=HeadIR, head=HHPosIR]),
          w([id=HeadPos, form=_, head=_])).
```

Using our representation, we can encode the dependencies shown in Fig. 11.10 and Table 11.13, and check whether they are projective or not. The `deprel_1` fact represents the first graph:

```
  deprel_1([w([id=1, form=the, head=2, deprel=det]),
    w([id=2, form=waiter, head=3, deprel=sub]),
    w([id=3, form=brought, head=0, deprel=root]),
```

```
    w([id=4, form=the, head=5, deprel=det]),
    w([id=5, form=meal, head=3, deprel=obj])]).
```

for which the program returns an empty list of nonprojective links, meaning that the graph is projective:

```
?- deprel_1(L), nonproj_links(L, NONPROJ).
L = ...
NONPROJ = []
```

The second one is nonprojective, and the program returns the link that breaks the projectivity:

```
deprel_2([w([id=1, form='Um', head=4, deprel='CP']),
  w([id=2, form=ihn, head=4, deprel='OA']),
  w([id=3, form=dennoch, head=4, deprel='MO']),
  w([id=4, form=anzuschieben, head=6, deprel='MO']),
  w([id=5, form=',', head=6, deprel='PUNC']),
  w([id=6, form=wollte, head=0, deprel='ROOT']),
  w([id=7, form=sich, head=21, deprel='OA']),
  w([id=8, form=die, head=9, deprel='NK']),
  w([id=9, form='Privatwirtschaft', head=6,
   deprel='SB']),
  w([id=10, form=erstmals, head=6, deprel='MO']),
  w([id=11, form='``', head=6, deprel='PUNC']),
  w([id=12, form=in, head=21, deprel='MO']),
  w([id=13, form=erheblichem, head=12, deprel='NK']),
  w([id=14, form='Umfang', head=12, deprel='NK']),
  w([id=15, form=an, head=21, deprel='OP']),
  w([id=16, form=den, head=15, deprel='15']),
  w([id=17, form='Risiken', head=15, deprel='NK']),
  w([id=18, form='''', head=15, deprel='PUNC']),
  w([id=19, form=der, head=20, deprel='NK']),
  w([id=20, form='Investition', head=15,
   deprel='AG']),
  w([id=21, form=beteiligen, head=6, deprel='OC']),
  w([id=22, form='.', head=6, deprel='PUNC'])]).

?- deprel_2(L), nonproj_links(L, NPROJ).
L = ...
NPROJ = [w([id=7, form=sich, head=21, deprel='OA'])]
```

11.10.2 A Method to Projectivize Links

Kunze (1967) proposed a simple method to build projective dependency trees from trees containing nonprojective links. The idea is to replace all the nonprojective links (i, h_i) between a word i and its head h_i with (i, h_{h_i}), where h_{h_i} is the head of h_i. As a graphical analogy, we can say that we move up the head of a nonprojective link to its head's head. In Fig. 11.15, we would replace

```
w([id=1, form='What', head=7])
```

with

```
w([id=1, form='What', head=4])
```

and in Table 11.13,

```
w([id=7, form=sich, head=21]),
```

with

```
w([id=7, form=sich, head=6]),
```

To build a complete projective tree, we can proceed iteratively. We replace the nonprojective links with projective equivalents, one at a time, until we obtain a projective tree. As there can be two or more nonprojective links in a tree, a simple operation ordering is to always select the nonprojective link that has the smallest span. This algorithm is guaranteed to converge.

Once we have projectivized a graph, it is possible to carry out the reverse operation to recover the original structure. To do this, we can keep the list of operations as backpointers. Another idea that is used in parsing is to replace the arc labels – the grammatical functions – that have been modified with traces indicating the way up and the way down (Nivre and Nilsson 2005).

In Table 11.13, the two predicates:

```
w([id=7, form=sich, head=21, deprel='OA']),
w([id=21, form=beteiligen, head=6, deprel='OC']),
```

where `'OA'` and `'OC'` are the original functions, would be replaced, respectively, with

```
w([id=7, form=sich, head=6, deprel='OA+OC']),
```

for the link that is moved up and with

```
w([id=21, form=beteiligen, head=6, deprel='OC-']),
```

for the path that was used by the movement. In the new label `'OA+OC'`, the `+` sign indicates that this link was moved, and the suffix `OC`, the path it used. The change from `OC` to `OC-` is to help disambiguation when a head has two or more dependents with the same label.

11.11 From Constituency to Dependency

In spite of their different graphical representation, constituency and dependency are related formalisms, and it is possible to some extent to convert constituent parse trees into dependency trees. The reverse is also true, at least partly: unlabeled projective dependency trees have a constituent equivalent. We will now review techniques to convert constituents into dependencies.

11.11.1 Transforming a Constituent Parse Tree into Dependencies

The procedure to convert constituents into dependencies is based on the idea of assigning each constituent in the parse tree a unique head selected amongst the constituent's children (Jensen et al. 1993). In the grammar below, we indicate the heads in the right-hand side of the rules with an asterisk:

```
s  --> np, vp*
np --> det, noun*
vp --> verb*, np
```

By following the child–parent links from the word level up to the root of the tree, we can label every constituent with a head word. The heads of the respective constituents can then be used to create dependency trees. Our small grammar would select the noun as the lexical head of a noun phrase `np`, the verb as the head of a verb phrase `vp`, and the head of `vp` as the head of the sentence `s`.

As an application example, let us consider the sentence *The boy hit the ball*, shown in Chap. 1. Using the rules above and the parse tree in Fig. 1.4, we can establish that *boy* and *ball* are the respective heads of *the* in *the boy* and *the* in *the ball*, that *hit* is the head of the second noun phrase, and hence the head of *ball*, and finally using the sentence rule `s`, that *hit* is also the head of the first noun phrase, i.e., of *boy* and the root of the sentence. We have thus recreated the dependency tree shown in Fig. 1.7.

In most constituent treebanks, however, the head is not indicated and the conversion is not straightforward. Magerman (1994) and Collins (1999) designed rules to create dependencies out of the Penn treebank. To do this, they singled out one symbol in the right-hand side of each phrase-structure rule to be the head of the remaining symbols. Table 11.15 is an example of such rules to convert the Penn treebank constituents into dependencies. The rules scan the constituent daughters from left to right ($\rightarrow$) or from right to left ($\leftarrow$) with a priority ordering.

Table 11.15 Head percolation rules (After Johansson and Nugues (2007a))

ADJP	←	NNS QP NN $ ADVP JJ VBN VBG ADJP JJR NP JJS DT FW RBR RBS SBAR RB
ADVP	→	RB RBR RBS FW ADVP TO CD JJR JJ IN NP JJS NN
CONJP	→	CC RB IN
FRAG	→	(NN* | NP) W* SBAR (PP | IN) (ADJP | JJ) ADVP RB
INTJ	←	**
LST	→	LS :
NAC	←	NN* NP NAC EX $ CD QP PRP VBG JJ JJS JJR ADJP FW
NP NX	←	(NN* | NX) JJR CD JJ JJS RB QP NP-ε NP
PP WHPP	→	(first non-punctuation after preposition)
PRN	→	(first non-punctuation)
PRT	→	RP
QP	←	$ IN NNS NN JJ RB DT CD NCD QP JJR JJS
RRC	→	VP NP ADVP ADJP PP
S	←	VP *-PRD S SBAR ADJP UCP NP
SBAR	←	S SQ SINV SBAR FRAG IN DT
SBARQ	←	SQ S SINV SBARQ FRAG
SINV	←	VBZ VBD VBP VB MD VP *-PRD S SINV ADJP NP
SQ	←	VBZ VBD VBP VB MD *-PRD VP SQ
UCP	→	**
VP	→	VBD VBN MD VBZ VB VBG VBP VP *-PRD ADJP NN NNS NP
WHADJP	←	CC WRB JJ ADJP
WHADVP	→	CC WRB
WHNP	←	NN* WDT WP WP$ WHADJP WHPP WHNP
X	→	**

11.11.2 Trace Revisited

As the constituent model does not easily render what corresponds to nonprojective links in dependency grammars, the Penn Treebank (Marcus et al. 1993) uses two main workarounds to encode them. The first one is to create specific categories where nonprojectivity is most frequent in English, all sorts of *wh*-phrases, to isolate it from the other phrase categories. It dedicates as many as five categories for this, more than a third of all its constituent categories: SBARQ for direct questions introduced by *wh*-words or *wh*-phrases, SQ for subconstituents of SBARQ, WHADVP for *wh*-adverb phrases, WHNP for *wh*-noun phrases, and finally WHPP for *wh*-prepositional phrases.

The second workaround is to introduce null elements and pseudo-attachments (Marcus et al. 1994). The traces denoted T are one of them. They mark the position where moved *wh*-constituents are interpreted in a parse tree. Figure 11.36 shows an example of such traces. Doing so, the Penn Treebank embeds dependencies into its constituents forming a sort of dual formalism.

A direct representation of the traces in a dependency graph is probably a more intuitive solution. Table 11.16 and Fig. 11.37 show how they can be converted into a

```
( (SBARQ-1
    (WHADVP-2 (WRB Why) )
    (PRN
      (, ,)
      (S
        (NP-SBJ (PRP they) )
        (VP (VBP wonder)
          (SBAR (-NONE- 0)
            (SBARQ (-NONE- *T*-1) ))))
      (, ,) )
    (SQ (MD should)
      (NP-SBJ (PRP it) )
      (VP (VB belong)
        (PP-CLR (TO to)
          (NP (DT the) (NNP EC) ))
        (ADVP-PRP (-NONE- *T*-2) )))
    (. ?) ))
```

Fig. 11.36 Constituent representation of the sentence *Why, they wonder, should it belong to the EC?* in the Penn treebank (Marcus et al. 1993)

Table 11.16 A nonprojective representation of the sentence *Why, they wonder, should it belong to the EC?* Projective heads are given in the fifth column (After Johansson and Nugues (2007a))

Index	Word	Head	Function	Projective head
1	Why	8	PRP	6
2	,	4	P	4
3	they	4	SBJ	4
4	wonder	6	PRN	6
5	,	4	P	4
6	should	0	ROOT-SBARQ	0
7	it	6	SBJ	6
8	belong	6	VC	6
9	to	11	PMOD	11
10	the	11	NMOD	11
11	EC	8	CLR	8
12	?	6	P	6

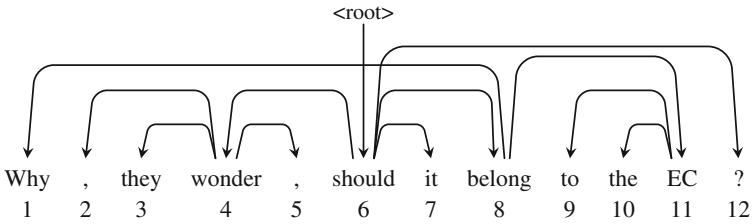


Fig. 11.37 Nonprojectivity in the Penn treebank with the sentence *Why, they wonder, should it belong to the EC?* (After Johansson and Nugues (2007a))

dependency graph by relinking them to their interpretations (Johansson and Nugues 2007a). Links are created from the heads of the traces, here *wonder* for `*T*-1` and *belong* for `*T*-2` to the heads of the pointed constituents unless the relinking causes the dependency graph to become cyclic. This explains the arc between *belong* and *Why*, while a potential cycle prevents a link from *wonder*.

11.12 Further Reading

Literature on Chomsky's works and generative transformational grammar is uncountable. Most linguistics textbooks in the English-speaking world retain this approach. Recent accounts include Radford (1988), Ruwet (1970), Haegeman and Gueron (1999), Lasnik et al. (2000).

Principles of dependency grammars stem from an old tradition dating back to the ancient Greek and Latin grammar schools. Tesnière (1966) proposes a modern formulation. Heringer (1993) provides a short and excellent summary of his work. Other accounts include Lecerf (1960a,b), Hays (1964), Kunze (1975), and Mel'čuk (1988). For an early definition and example of parsing as the analysis of bilexical relations, see Domergue (1782, pp. 205–206).

Often associated to dependencies, the concept of grammatical function is more recent. Cull introduced this term in English in 1840 and the *compléments circonstantiels* 'grammatical adjuncts' found their place in the high school grammar books in France from the 1850s (Chervel 1979). The adjunct categorization was then based on rhetorical criteria. As of today, the definition of a comprehensive annotation set for grammatical functions is still a matter of debate.

Within the work of Tesnière, valence has been a very productive concept, although it has not always been explicitly acknowledged. It provides theoretical grounds for verb subcategorization, cases, and selectional restrictions that we find in other parts of this book (Chap. 15). In addition to verbs, valence can apply to adjectives and nouns.

Unification-based grammars were born when Alain Colmerauer designed the *systèmes-Q* (1970) and later the Prolog language with his colleagues. *Systèmes-Q* have been applied in the MÉTÉO system to translate weather reports from English to French (TAUM 1971). MÉTÉO is still in use today. Prolog is derived from them and was also implemented for a project aimed at dialogue and language analysis (Colmerauer et al. 1972). For a review of its history, see Colmerauer and Roussel (1996).

Unification-based grammars have been used in many syntactic theories. The oldest and probably the simplest example is that of Definite Clause Grammars (Colmerauer 1978; Pereira and Warren 1980). Since then there have been many followers. The most notable include head-driven phrase structure grammars (Pollard and Sag 1994) and lexical function grammars (Kaplan and Bresnan 1982). Unification-based grammars do not depend on a specific syntactic formalism. They are merely a tool that we used with PS rules in this chapter. Dependency grammars

can also make use of them. Dependency unification grammar (Hellwig 1980, 1986) and unification dependency grammar (Maxwell 1995) are examples. Accounts of unification formalisms in French include Abeillé (1993) and in German, Müller (1999).

Exercises

11.1. Describe step-by-step how the Prolog search mechanism would generate the sentence *the boy hit the ball*, and compare this trace with that of Fig. 11.1.

11.2. Write a Prolog program that converts a DCG grammar into its Chomsky normal form equivalent.

11.3. Write a grammar using the DCG notation to analyze simple sentences: a noun phrase and a verb phrase, where the verb phrase is either a verb or a verb and an object. Write transformation rules that map declarative sentences into their negation.

11.4. Complement PS rules of Exercise 11.3 to parse a possible prepositional phrase within the verb phrase. Write transformation rules that carry out a topicalization of the prepositional phrase.

11.5. Write DCG rules using the gap threading technique to handle sentences and questions of Table 11.2.

11.6. Find a text of 10–20 lines in a language you know and bracket the constituents with the phrase labels of Table 11.3.

11.7. Unify $\begin{bmatrix} gen : fem \\ case : acc \end{bmatrix}$ and $\begin{bmatrix} gen : fem \\ num : pl \end{bmatrix}$, $\begin{bmatrix} gen : fem \\ num : pl \\ case : acc \end{bmatrix}$ and $\begin{bmatrix} gen : fem \\ num : sg \end{bmatrix}$, $\begin{bmatrix} gen : masc \\ num : X \\ case : nom \end{bmatrix}$ and $\begin{bmatrix} gen : masc \\ num : pl \\ case : Y \end{bmatrix}$ when possible.

11.8. Unify $\begin{bmatrix} f_1 : v_1 \\ f_2 : X \end{bmatrix}$ and $\begin{bmatrix} f_1 : v_5 \\ f_2 : \begin{bmatrix} f_3 : v_3 \\ f_4 : v_4 \end{bmatrix} \end{bmatrix}$, $\begin{bmatrix} f_1 : v_1 \\ f_2 : X \end{bmatrix}$ and $\begin{bmatrix} f_1 : Y \\ f_2 : \begin{bmatrix} f_3 : v_3 \\ f_4 : Y \end{bmatrix} \end{bmatrix}$, $\begin{bmatrix} f_1 : v_1 \\ f_2 : X \end{bmatrix}$ and $\begin{bmatrix} f_5 : X \\ f_2 : Y \\ f_1 : Y \end{bmatrix}$.

11.9. Using the unification grammar formalism, write rules describing the noun group in a language you know.

11.10. Write a `norm/2` predicate that transforms complete lists into incomplete ones as, for example, `[a, b, [c, d], e]` into `[a, b, [c, d | _], e, | _]`.

11.11. Find a text of approximately ten lines in a language you know and draw the stemmas (dependency links).

11.12. Draw the stemmas of the sentences in Tables 11.13 and 11.14.

11.13. Annotate the stemmas of the sentences in Tables 11.13 and 11.14 with their corresponding functions.

11.14. Use the program in Sect. 11.10 and complete it to projectivize the graphs.

11.15. Write a program to convert constituents into dependency graphs.

11.16. Compute the theoretical number of dependency graphs of sentences of seven words, given that a word has only one head. Out of this set, compute the number of graphs that are projective.

Chapter 12
Constituent Parsing

12.1 Introduction

In the previous chapters, we used Prolog's built-in search mechanism and the DCG notation to parse sentences and constituents. This search mechanism has drawbacks, however. To name some of them: its depth-first strategy does not handle left-recursive rules well, and backtracking is sometimes inefficient. In addition, if DCGs are appropriate to describe constituents, we haven't seen means to parse dependencies until now.

This chapter describes algorithms and data structures to improve the efficiency of constituent parsing. It begins with a basic bottom-up algorithm and then introduces techniques using well-formed substring tables or charts. Charts are arrays to store parsing results and hypotheses. They are popular parsing devices because of some superior features: charts accept left-recursive rules, avoid backtracking, and can work with a top-down or bottom-up control.

Frequently, sentences show an ambiguous structure – exhibit more than one possible parse. Search strategies, either bottom-up or top-down, produce solutions blindly; the ordering of the resulting parse trees being tied to that of the rules. For most cases, however, sentences are not ambiguous to human readers who retain one single sensible analysis. To come to a similar result, parsers require a disambiguation mechanism.

Early disambiguation methods implemented common sense rules to assess parse trees and to discard implausible ones. Current solutions, inspired from speech recognition and part-of-speech tagging, use statistical or machine-learning techniques. They enable us to properly parse most ambiguous sentences. Recent approaches based on dependencies yield a very high rate of performance for unrestricted texts. This chapter outlines symbolic techniques as well as probabilistic methods applicable to constituency grammars. The next chapter will introduce dependency parsing.

P.M. Nugues, *Language Processing with Perl and Prolog*, Cognitive Technologies,
DOI 10.1007/978-3-642-41464-0_12, © Springer-Verlag Berlin Heidelberg 2014

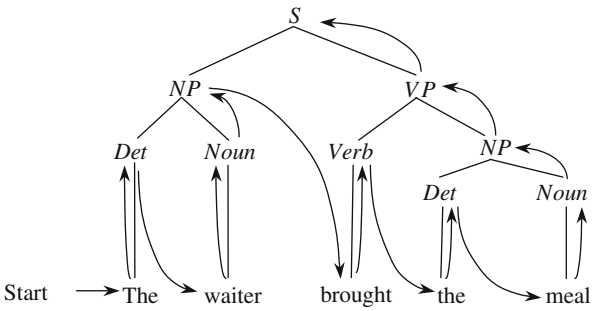


Fig. 12.1 Bottom-up parsing. The parser starts with the words and builds the syntactic structure up to the top node

12.2 Bottom-Up Parsing

12.2.1 The Shift–Reduce Algorithm

We saw in Chap. 9 that left-recursive rules may cause top-down parsing to loop infinitely. Left-recursion is used to express structures such as noun phrases modified by a prepositional phrase, or conjunctions of noun phrases as, for example,

```
np --> np, pp.
np --> np, conj, np.
```

It is possible to eliminate left-recursive rules using auxiliary symbols and rules. However, this results in larger grammars that are less regular. In addition, parsing with these new rules yields slightly different syntactic trees, which are often less natural.

A common remedy to handle left-recursive rules is to run them with a bottom-up search strategy. Instead of expanding constituents from the top node, a bottom-up parser starts from the words. It looks up their parts of speech, builds partial structures out of them, and goes on from partial structure to partial structure until it reaches the top node. Figure 12.1 shows the construction order of partial structures that goes from the annotation of *the* as a determiner up to the root `s`.

The shift and reduce algorithm is probably the simplest way to implement bottom-up parsing. As input, it uses two arguments: the list of words to parse and a symbol, `s`, `np`, for example, representing the parsing goal. The algorithm gradually reduces words, parts of speech, and phrase categories until it reaches the top node symbol – the parsing goal. The algorithm consists of a two-step loop:

1. **Shift** a word from the phrase or sentence to parse onto a stack.
2. Apply a sequence of grammar rules to **reduce** elements of the stack.

This loop is repeated until there are no more words in the list and the stack is reduced to the parsing goal. Table 12.1 shows an example of shift and reduce operations applied to the sentence *The waiter brought the meal*.

Table 12.1 Steps of the shift–reduce algorithm to parse *the waiter brought the meal*. At iteration 7, a further reduction of the stack yields `[s]`, which is the parsing goal. However, since there are remaining words in the input list, the algorithm fails and backtracks to produce the next states of the stack. The table does not show the exploration of paths leading to a failure

It.	Stack	S/R	Word list
0			`[the, waiter, brought, the, meal]`
1	`[the]`	Shift	`[waiter, brought, the, meal]`
2	`[det]`	Reduce	`[waiter, brought, the, meal]`
3	`[det, waiter]`	Shift	`[brought, the, meal]`
4	`[det, noun]`	Reduce	`[brought, the, meal]`
5	`[np]`	Reduce	`[brought, the, meal]`
6	`[np, brought]`	Shift	`[the, meal]`
7	`[np, v]`	Reduce	`[the, meal]`
8	`[np, v, the]`	Shift	`[meal]`
9	`[np, v, det]`	Reduce	`[meal]`
10	`[np, v, det, meal]`	Shift	`[]`
11	`[np, v, det, n]`	Reduce	`[]`
12	`[np, v, np]`	Reduce	`[]`
13	`[np, vp]`	Reduce	`[]`
14	`[s]`	Reduce	`[]`

12.2.2 Implementing Shift–Reduce Parsing in Prolog

We implement both arguments of the `shift_reduce/2` predicate as lists: the words to parse and the symbol – or symbols – corresponding to the parsing goal. We represent grammar rules and the vocabulary as facts, as shown in Table 12.2.

Using this grammar, `shift_reduce` should accept the following queries:

```
?- shift_reduce([the, waiter, brought, the, meal],
  [s]).
true

?- shift_reduce([the, waiter, brought, the, meal],
  [np, vp]).
true

?- shift_reduce([the, waiter, slept], X).
X = [s];
X = [np, vp];
X = [np, v];
...
```

To implement this predicate, we need an auxiliary stack to hold words and categories where we carry out the reduction step. This initial value of the stack is an empty list

Table 12.2 Rules and vocabulary of a shift–reduce parser

Rules	Vocabulary	
`rule(s, [np, vp]).`	`word(d, [the]).`	`word(v, [brought]).`
`rule(np, [d, n]).`	`word(n, [waiter]).`	`word(v, [slept]).`
`rule(vp, [v]).`	`word(n, [meal]).`	
`rule(vp, [v, np]).`		

```
% shift_reduce(+Sentence, ?Category)
shift_reduce(Sentence, Category) :-
  shift_reduce(Sentence, [], Category).
```

Then `shift_reduce/3` uses two predicates, `shift/4` and `reduce/2`. It repeats the reduction recursively until it no longer finds a reduction. It then applies `shift`. The parsing process succeeds when the sentence is an empty list and `Stack` is reduced to the parsing goal:

```
% shift_reduce(+Sentence, +Stack, ?Category)
shift_reduce([], Category, Category).
shift_reduce(Sentence, Stack, Category) :-
  reduce(Stack, ReducedStack),
  write('Reduce: '), write(ReducedStack), nl,
  shift_reduce(Sentence, ReducedStack, Category).
shift_reduce(Sentence, Stack, Category) :-
  shift(Sentence, Stack, NewSentence, NewStack),
  write('Shift: '), write(NewStack), nl,
  shift_reduce(NewSentence, NewStack, Category).
```

`shift/4` removes the first word from the word list currently being parsed and puts it on the top the stack – here appends it to the end of the `Stack` list – to produce a `NewStack`.

```
% shift(+Sentence, +Stack, -NewSentence, -NewStack)
shift([First | Rest], Stack, Rest, NewStack) :-
  append(Stack, [First], NewStack).
```

`reduce/2` simplifies the `Stack`. It searches the rules that match a sequence of symbols in the stack using `match_rule/2` and `match_word/2`.

```
%reduce(+Stack, -NewStack)
reduce(Stack, NewStack) :-
  match_rule(Stack, NewStack).
reduce(Stack, NewStack) :-
  match_word(Stack, NewStack).
```

`match_rule/2` attempts to find the `Expansion` of a rule on the top of `Stack`, and replaces it with `Head` to produce `ReducedStack`:

```
match_rule(Stack, ReducedStack) :-
  rule(Head, Expansion),
  append(StackBottom, Expansion, Stack),
  append(StackBottom, [Head], ReducedStack).
```

`match_word/2` is similar:

```
match_word(Stack, NewStack) :-
  append(StackBottom, Word, Stack),
  word(POS, Word),
  append(StackBottom, [POS], NewStack).
```

The stack management of this program is not efficient because `shift/4`, `match_word/2`, and `match_rule/2` have to traverse it using `append/3`. It is possible to avoid the traversal using a reversed stack. For an optimization, see Exercise 12.1.

12.2.3 Differences Between Bottom-Up and Top-Down Parsing

Top-down and bottom-up strategies are fundamental approaches to parsing. The top-down exploration is probably more intuitive from the viewpoint of a Prolog programmer, at least for a neophyte. Once a grammar is written, Prolog relies on its built-in search mechanism to parse a sentence. On the contrary, bottom-up parsing requires additional code and may not be as natural.

Whatever the parsing strategy, phrase-structure rule grammars are written roughly in the same way. There are a couple of slight differences, however. As we saw, bottom-up parsing can handle left-recursive rules such as those describing conjunctions. In contrast, top-down parsers can handle null constituents like

```
det --> [].
```

Bottom-up parsers could not use such a rule with an empty symbol because they are able to process actual words only.

Both parsing methods may fail to find a solution, but in different ways. Top-down parsing explores all the grammar rules starting from the initial symbol, whatever the actual tokens. It leads to the expansion of trees that have no chance to yield any solution since they will not match the input words. On the contrary, bottom-up parsing starts with the words and hence builds trees that conform to the input. However, a bottom-up analysis annotates the input words with every possible part of speech, and generates the corresponding partial trees, even if they have no chance to result into a sentence.

For both strategies, Prolog produces a solution – whenever it exists – using backtracking. When Prolog has found a dead-end path, whether in the bottom-up or the top-down mode, it selects another path and explores this path until it completes the parse or fails. Backtracking may repeat a same operation since Prolog does not

store intermediate or partial solutions. We will see in the next section a parsing technique that stores incomplete solutions using a table or a **chart** and thus avoids parsing repetitions.

12.3 Chart Parsing

12.3.1 Backtracking and Efficiency

Backtracking is an elegant and simple mechanism, but it frequently leads to reparsing a same substructure to produce the final result. Consider the noun phrase *The meal of the day* and DCG rules in Fig. 12.2 to parse it.

The DCG search algorithm first tries rule `np --> npx`, uses `npx` to parse *The meal*, and fails because of the remaining words *of the day*. It then backtracks with the second `np` rule, reuses `npx` to reparse *The meal*, and finally completes the analysis with `pp`. Backtracking is clearly inefficient here. The parser twice applies the same rule to the same group of words because it has forgotten a previous result: *The meal* is an `npx`.

Chart – or tabular – parsing is a technique to avoid a parser repeating a same analysis. A chart is a memory where the parser stores all the possible partial results at a given position in the sentence. When it needs to process a subsequent word, the parser fetches partial parse structures obtained so far in the chart instead of reparsing them. At the end of the analysis, the chart contains all possible parse trees and subtrees that it represents tidily and efficiently.

12.3.2 Structure of a Chart

A chart represents intervals between words as nodes of a graph. Considering a sentence of N words, nodes are numbered from left to right, from 0 to N. The chart – which can also be viewed as a table – then has $N + 1$ entries or positions. Figure 12.3 shows word numbering of the sentence *Bring the meal* and the noun phrase *The meal of the day*. A chart node is also called a vertex.

Directed arcs (or edges) connect nodes and define constituents. Each arc has a label that corresponds to the syntactic category of the group it spans (Fig. 12.4). Charts consist then of sets of nodes and directed labeled arcs. This algorithmic structure is also called a directed acyclic graph (DAG).

A chart can store alternative syntactic representations. As we saw in Chap. 9, the grammar in Fig. 12.5 yields two parse trees for the sentence

Bring the meal of the day.

```
np --> npx.     npx --> det, noun. pp --> prep, np.
np --> npx, pp.
```

Fig. 12.2 A small set of DCG rules where left-recursion has been eliminated

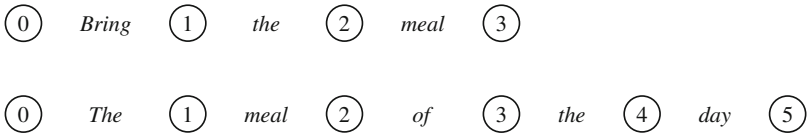


Fig. 12.3 Nodes of a chart

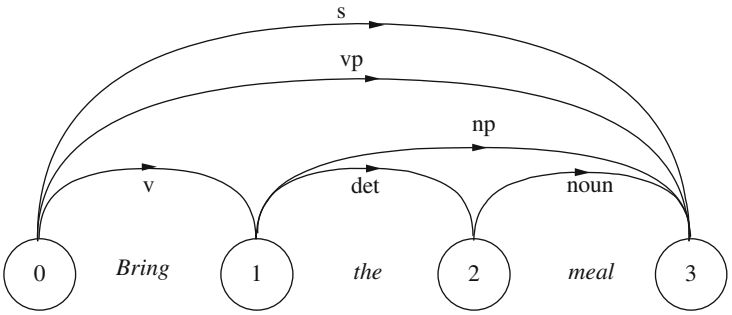


Fig. 12.4 Nodes and arcs of a chart associated with the sentence *Bring the meal*

```
s --> vp.          np --> det, noun.      pp --> prep, np.
vp --> v, np, pp.  np --> det, adj, noun.
vp --> v, np.      np --> np, pp.
```

Fig. 12.5 A small grammar for restaurant orders in English

The chart of Fig. 12.6 shows the possible parses of this sentence. The rules:

```
vp --> v, np.
vp --> v, np, pp.
```

create two paths that connect node 0 to node 6. Starting from node 0, the first one traverses nodes 1 and 6: arc `v` from 0 to 1, then `np` from 1 to 6. The second sequence of arcs traverses nodes 1, 3, 6: arc `v` from 0 to 1, then `np` from 1 to 3, and finally `pp` from 3 to 6.

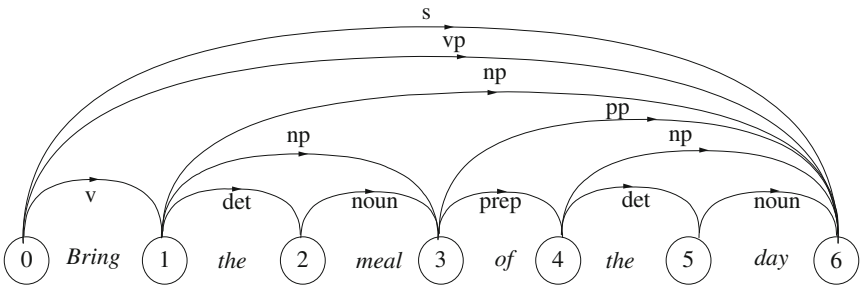


Fig. 12.6 A chart representing alternative parse trees

12.3.3 The Active Chart

So far, we used charts as devices to represent partial or complete parse trees. Charts can also store representations of constituents currently being parsed. In this case, a chart is said to be active.

In the classical chart notation, the parsing progress of a constituent is indicated using a dot (•) inserted in the right-hand side of the rules. A **dotted rule** represents what has been parsed so far, with the dot marking the position of the parser relative to the input. Thus

```
np --> det noun •
```

is a completely parsed noun phrase composed of a determiner and a noun. Since the constituent is complete, the arc is said to be inactive. Rules

```
np --> det • noun
np --> • det noun
```

describe noun phrases being parsed. Both correspond to constituent hypotheses that the parser tries to find. In the first rule, the parser has found a determiner and looks for a noun to complete the parse. The second rule represents the constituent being sought originally. Both arcs are said to be active since the parser needs more words from the input to confirm them.

Consider the sentence *Bring the meal*. Table 12.3 shows dotted-rules and arcs during the parsing process, and Fig. 12.7 shows a graphic representation of them.

Charts can be used with top-down, bottom-up, or more sophisticated strategies. We introduce now a top-down version due to Earley (1970). Its popularity comes from its complexity, which has been demonstrated as $0(N^3)$.

Table 12.3 Some dotted-rules and arcs in the chart while parsing *Bring the meal of the day*

Positions	Rules			Arcs	Constituents
0	`s`	`-->`	• `vp`	[0, 0]	• *Bring the meal*
1	`vp`	`-->`	`v` • `np`	[0, 1]	*Bring* • *the meal*
1	`np`	`-->`	• `det noun`	[1, 1]	• *the meal*
1	`np`	`-->`	• `np pp`	[1, 1]	• *the meal*
2	`np`	`-->`	`det` • `noun`	[1, 2]	*the* • *meal*
3	`np`	`-->`	`det noun` •	[1, 3]	*the meal* •
3	`np`	`-->`	`np` • `pp`	[1, 3]	*the meal* •
3	`vp`	`-->`	`v np` •	[0, 3]	*Bring the meal* •
3	`s`	`-->`	`vp` •	[0, 3]	*Bring the meal* •

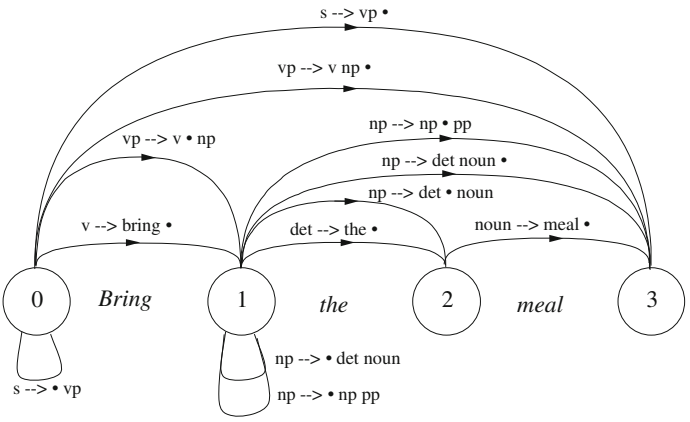


Fig. 12.7 Some arcs of a chart labeled with dotted-rules while parsing *Bring the meal of the day*

12.3.4 Modules of an Earley Parser

An Earley parser consists of three modules, the predictor, the scanner, and the completer, which are chained by the parsing process. The initial goal of the algorithm is to parse the start symbol, which is generally a sentence `s`. Here, we illustrate the algorithm with the noun phrase *The meal of the day* and the rules in Fig. 12.5. The start symbol is then `np` and is represented by the dotted-rule

`start --> • np`

The Predictor

At a given position of the parsing process, the predictor determines all possible further parses. To carry this out, the predictor selects all the rules that can process active arcs. Considering the dotted rule, `lhs --> c`$_1$ `c`$_2$ `...• c...c`$_n$, the

```
start --> • np          [0, 0]
np --> • det noun       [0, 0]
np --> • det adj noun   [0, 0]
np --> • np pp          [0, 0]
```

Fig. 12.8 Dotted-rules resulting from the recursive run of the predictor with starting goal `np`

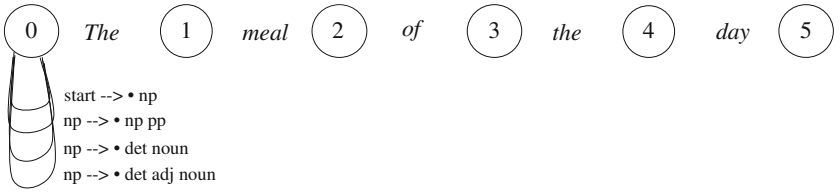


Fig. 12.9 Graphic representation of the predictor results

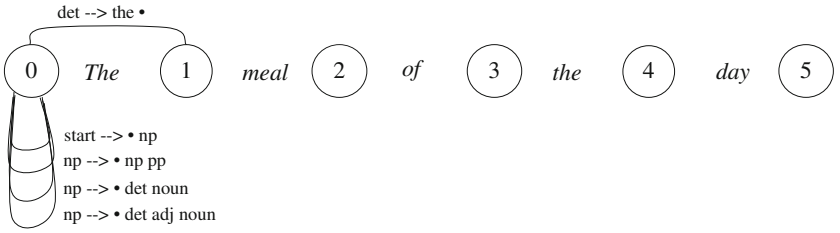


Fig. 12.10 The scanner accepts word *The* from the input

predictor searches all the rules where `c` is the left-hand-side symbol: `c --> x1 ...xk`. The predictor introduces them into the chart as new parsing goals in the form of `c --> • x1 ...xk`. The predictor proceeds recursively with nonterminal symbols until it reaches the parts of speech. Considering *The meal of the day* with `np` as the starting parsing goal and applying the predictor results in new goals shown in Fig. 12.8 and graphically in Fig. 12.9.

The Scanner

Once all possible predictions are done, the scanner accepts a new word from the input, here *the*. The parts of speech to the right of a dot are matched against the word, here in our example, rules `np --> • det noun` and `np --> • det adj noun`. The scanner inserts the word into the chart with all its matching part-of-speech readings in the form of `pos --> word •` and advances the parse position to the next node, here

```
det --> the •    [0, 1]
```

as shown in Fig. 12.10.

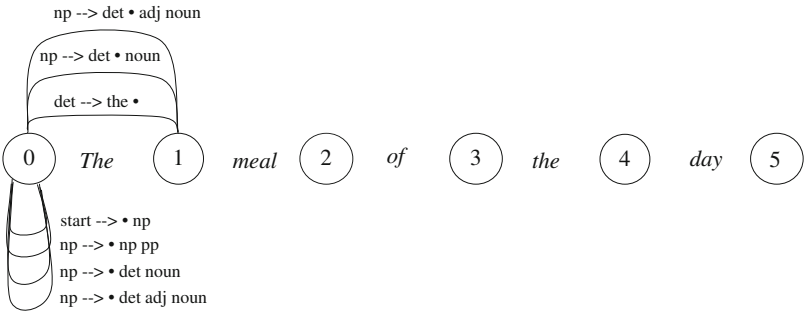


Fig. 12.11 The completer looks for completed constituents and advances the dot over them

The Completer

The scanner introduces new constituents under the form of parts of speech in the chart. The completer uses them to advance the dot of active arcs expecting them, and possibly complete the corresponding constituents. Once a constituent has been completed, it can in turn modify others expecting it in active arcs. The completer thus is applied to propagate modifications and to complete all possible arcs. It first determines which constituents are complete by looking for dots that have reached the end of a rule: `c --> x`$_1$ `...x`$_k$ •. The completer then searches all the active arcs expecting `c`, that is, the rules with a dot to the left of it: `lhs --> c`$_1$ `c`$_2$ `...•  c  ...c`$_n$, moves the dot over `c`: `lhs --> c`$_1$ `c`$_2$ `...c • ...c`$_n$, and inserts the new arc into the chart. It proceeds recursively from the parts of speech to all the possible higher-level constituents.

In our example, the only completed constituent is the part of speech `det`. The completer advances the dot over it in two active arcs and inserts them into the chart. It does not produce new completed constituents (Fig. 12.11).

```
np --> det • noun          [0, 1]
np --> det • adj noun      [0, 1]
```

From node 1, the predictor is run again, but it does not yield new arcs. The scanner accepts word *meal*, advances the position to 2, and inserts

```
noun --> meal •     [1, 2]
```

as shown in Fig. 12.12.

At node 2, the completer can advance active arc

```
np --> det noun •       [0, 2]
```

and complete a higher-level constituent (Fig. 12.13).

```
np --> np • pp      [0, 2]
```

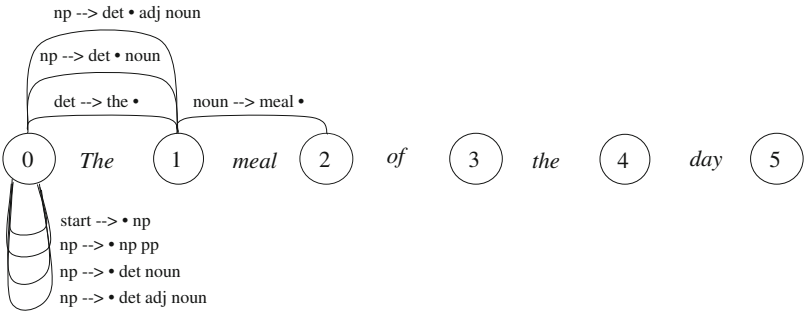


Fig. 12.12 Predictor and scanner are run with word *meal*

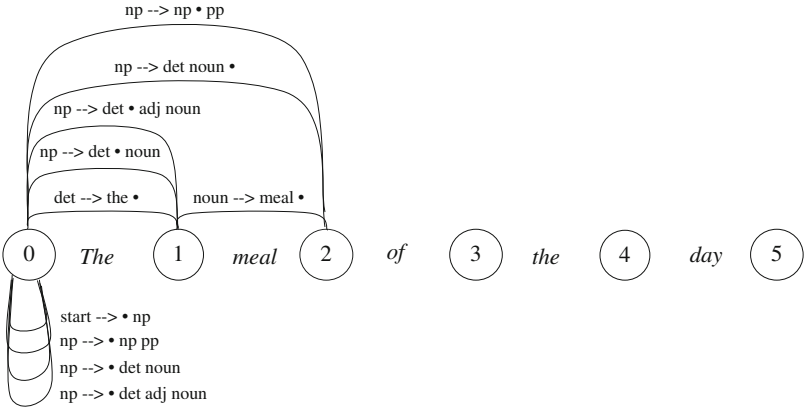


Fig. 12.13 The completer is run to produce new chart entries

12.3.5 The Earley Algorithm in Prolog

To implement the algorithm in Prolog, we must first represent the chart. It consists of arcs such as

```
np --> np • pp     [0, 2]
```

which we represent as facts

```
arc(np, [np, '.', pp], 0, 2).
```

The start symbol is encoded as:

```
arc(start, ['.', np], 0, 0).
```

Although this data representation is straightforward from the description of dotted-rules, it is not efficient from the speed viewpoint. The arc representation can be improved easily, but an optimization may compromise clarity. We leave it as an exercise.

New arcs are stored in the chart, a list called `Chart`, using the `expand_chart/1` predicate, which checks first that the new entry is not already in the chart:

```
expand_chart([], Chart, Chart).
expand_chart([Entry | Entries], Chart, NewChart) :-
  \+ member(Entry, Chart),
  !,
  expand_chart(Entries, [Entry | Chart], NewChart).
expand_chart([_ | Entries], Chart, NewChart) :-
  expand_chart(Entries, Chart, NewChart).
```

The Earley algorithm is implemented by the predicate `earley_parser/5`. It uses five arguments: the input word sequence, the current position in the sequence `CurPos`, the final position `FinalPos`, the current chart, and the final chart. The `earley_parser` main rule consists of calling the `predictor`, `scanner`, and `completer` predicates through the $N + 1$ nodes of the chart:

```
earley_parser([], FinalPos, FinalPos, Chart, Chart):-
  !.
earley_parser(Words, CurPos, FinalPos, Chart,
  FinalChart) :-
  predictor(CurPos, Chart, PredChart),
  NextPos is CurPos + 1,
  scanner(Words, RestWords, CurPos, NextPos,
      PredChart, ScanChart),
  completer(NextPos, ScanChart, NewChart),
  !,
  earley_parser(RestWords, NextPos, FinalPos,
      NewChart, FinalChart).
```

The Earley algorithm is called by `parse/3`, which takes the word sequence `Words` and the start `Category` as arguments. The `parse/3` predicate initializes the chart with the start symbol and launches the parse. The parsing success corresponds to the presence of a completed start symbol, which is, in our example, the `arc(start, [np, '.'], 0, FinalNode)` fact in the chart `FinalChart`:

```
parse(Words, Category, FinalChart) :-
  expand_chart([arc(start, ['.', Category], 0, 0)],
      [], Chart),
  earley_parser(Words, 0, FinalPos, Chart,
      FinalChart),
  member(arc(start, [Category, '.'], 0, FinalPos),
      FinalChart).
```

Table 12.4 shows the transcription of `np` rules in Fig. 12.5 encoded as Prolog facts. It contains a small vocabulary to parse the phrase *The meal of the day*.

Table 12.4 Rules and vocabulary for the chart parser

Rules	Words	
`rule(np, [d, n]).`	`word(d, [the]).`	`word(prep, [of]).`
`rule(np, [d, a, n]).`	`word(n, [waiter]).`	`word(v, [brought]).`
`rule(np, [np, pp]).`	`word(n, [meal]).`	`word(v, [slept]).`
`rule(pp, [prep, np]).`	`word(n, [day]).`	

The Predictor

The predictor looks for rules to expand arcs from a current position (`CurPos`). To carry this out, `predictor/3` searches all the arcs containing the pattern: `[..., X, '.', CAT, ...]`, where `CAT` matches the left-hand side of a rule: `rule(CAT, RHS)`. This is compactly expressed using the `findall/3` built-in predicate. `predictor/3` then adds `arc(CAT, ['.' | RHS], CurPos, CurPos)` to the chart `NewChart`. `predictor/3` is run recursively until no new arc can be produced, that is, `NewChartEntries == []`. It then returns the predictor's chart `PredChart`.

```
predictor(CurPos, Chart, PredChart) :-
  findall(
    arc(CAT, ['.' | RHS], CurPos, CurPos),
    (
      member(arc(LHS, ACTIVE_RHS, InitPos, CurPos),
        Chart),
      append(B, ['.', CAT | E], ACTIVE_RHS),
      rule(CAT, RHS),
      \+ member(arc(CAT, ['.' | RHS], CurPos,CurPos),
        Chart)
    ),
    NewChartEntries),
  NewChartEntries \== [],
  expand_chart(NewChartEntries, Chart, NewChart),
  predictor(CurPos, NewChart, PredChart),
  !.
predictor(_, PredChart, PredChart).
```

Using chart entry `arc(np, [np, '.', pp], 0, 2)` and rules in Table 12.4:

```
?- predictor(2,[arc(np, [np, '.', pp], 0, 2)],Chart).
```

adds

```
arc(pp, ['.', prep, np], 2, 2)
```

to the `Chart` list.

The Scanner

The scanner gets a new word from the input and looks for active arcs that match its possible parts of speech to the right of the dot. The scanner stores the word with its compatible parts of speech as new chart entries. Again, we use `findall/3` to implement this search.

```
scanner([Word | Rest], Rest, CurPos, NextPos, Chart,
      NewChart) :-
  findall(
    arc(CAT, [Word, '.'], CurPos, NextPos),
    (
      word(CAT, [Word]),
      once((
        member(arc(LHS, ACTIVE_RHS, InitPos, CurPos),
          Chart),
        append(B, ['.', CAT | E], ACTIVE_RHS)))
    ),
    NewChartEntries),
  NewChartEntries \== [],
  expand_chart(NewChartEntries, Chart, NewChart).
```

The Completer

The completer looks for completed constituents, that is, for arcs with a dot at the end of the right-hand-side part of the rule. They correspond to `arc(LHS, COMPLETE_RHS, InitPos, CurPos)`, where `COMPLETE_RHS` matches `[..., X, '.']`. We use the goal `append(_, ['.'], COMPLETE_RHS)` to find them. The completer then searches arcs with a dot to the right of the `LHS` category of completed constituents: `[..., '.', LHS, ...]`, advances the dot over `LHS`: `[..., LHS, '.', ...]`, and stores the new arc with updated node positions. We use `findall/3` to implement the search, and `completer/3` is run recursively until there is no arc to complete.

```
completer(CurPos, Chart, CompChart) :-
  findall(
    arc(LHS2, RHS3, PrevPos, CurPos),
    (
      member(arc(LHS, COMPLETE_RHS, InitPos, CurPos),
        Chart),
      append(_, ['.'], COMPLETE_RHS),
      member(arc(LHS2, RHS2,PrevPos,InitPos), Chart),
      append(B, ['.', LHS | E], RHS2),
```

```
        append(B, [LHS, '.' | E], RHS3),
        \+ member(arc(LHS2, RHS3, PrevPos, CurPos),
          Chart)
      ),
      CompletedChartEntries),
    CompletedChartEntries \== [],
    expand_chart(CompletedChartEntries,Chart,NewChart),
    completer(CurPos, NewChart, CompChart),
    !.
completer(_, CompChart, CompChart).
```

An Execution Example

Table 12.5 shows the arcs added to the chart while parsing the phrase *The meal of the day*. The parser is queried by:

```
?- parse([the, meal, of, the, day], np, Chart).
```

Note that the `completer` calls at position 2 that completes `np`, and at position 5 that completes `np`, `pp`, and the starting goal `np`.

12.3.6 The Earley Parser to Handle Left-Recursive Rules and Empty Symbols

The Earley parser handles left-recursive rules without looping infinitely. In effect, the predictor is the only place where the parser could be trapped into an infinite execution. This is avoided because before creating a new arc, the `predictor` predicate checks that it is not already present in the chart using the goal

```
\+ member(arc(CAT, ['.' | RHS], CrPos, CrPos), Chart)
```

So

```
start --> • np        [0, 0]
```

predicts

```
np --> • np pp        [0, 0]
np --> • det noun     [0, 0]
np --> • det adj noun [0, 0]
```

but `np --> • np pp` predicts nothing more since all the possible arcs are already in the chart.

The Earley algorithm can also parse null constituents. It corresponds to examples such as *meals of the day*, where the determiner is encoded as `word(d, [])`.

Table 12.5 Additions to the chart

Module	New chart entries
	Position 0
start	arc(start, ['.', np], 0, 0)
predictor	arc(np, [., d, n], 0, 0), arc(np, [., d, a, n], 0, 0), arc(np, [., np, pp], 0, 0)
	Position 1
scanner	arc(d, [the, .], 0, 1)
completer	arc(np, [d, ., a, n], 0, 1), arc(np, [d, ., n], 0, 1)
predictor	[]
	Position 2
scanner	arc(n, [meal, .], 1, 2)
completer	arc(np, [d, n, .], 0, 2)
completer	arc(np, [np, ., pp], 0, 2), arc(start, [np, .], 0, 2)
predictor	arc(pp, [., prep, np], 2, 2)
	Position 3
scanner	arc(prep, [of, .], 2, 3)
completer	arc(pp, [prep, ., np], 2, 3)
predictor	arc(np, [., d, n], 3, 3), arc(np, [., d, a, n], 3, 3), arc(np, [., np, pp], 3, 3)
	Position 4
scanner	arc(d, [the, .], 3, 4)
completer	arc(np, [d, ., a, n], 3, 4), arc(np, [d, ., n], 3, 4)
predictor	[]
	Position 5
scanner	arc(n, [day, .], 4, 5)
completer	arc(np, [d, n, .], 3, 5)
completer	arc(np, [np, ., pp], 3, 5), arc(pp, [prep, np, .], 2, 5)
completer	arc(np, [np, pp, .], 0, 5)
completer	arc(np, [np, ., pp], 0, 5), arc(start, [np, .], 0, 5)

As we wrote it, the scanner would fail on empty symbols. We need to add a second rule to it to handle empty lists:

```
% The first scanner rule
scanner([Word | Rest], Rest, CurPos, NextPos, Chart,
      NewChart) :-
  findall(
    arc(CAT, [Word, '.'], CurPos, NextPos),
    (
      word(CAT, [Word]),
      once((
        member(arc(LHS, ACTIVE_RHS, InitPos, CurPos),
          Chart),
        append(B, ['.', CAT | E], ACTIVE_RHS)))
    ),
```

```
    NewChartEntries),
  NewChartEntries \== [],
  expand_chart(NewChartEntries, Chart, NewChart),
  !.

% The second rule to handle empty symbols
scanner(Words, Words, CurPos, NextPos, Chart,
      NewChart) :-
  findall(
    arc(CAT, [[], '.'], CurPos, NextPos),
    (
      word(CAT, []),
      once((
        member(arc(LHS, ACTIVE_RHS, InitPos, CurPos),
          Chart),
        append(B, ['.', CAT | E], ACTIVE_RHS)))
    ),
    NewChartEntries),
  NewChartEntries \== [],
  expand_chart(NewChartEntries, Chart, NewChart),
  !.
```

Let us add

```
word(d, []).
word(n, [meals]).
```

to the database to be able to parse *meals of the day*:

```
?- parse([meals, of, the, day], np, Chart).
```

12.4 Probabilistic Parsing of Context-Free Grammars

So far, parsing methods made no distinction between possible parse trees of an ambiguous sentence. They produced trees either through a systematic backtracking or simultaneously in a chart with the Earley algorithm. The reason is that the parsers considered all rules to be equal and tried them sequentially.

We know this is not the case in reality. Some rules describe very frequent structures, while others are rare. As a solution, a parser could try more frequent rules first, prefer certain rules when certain words occur, and rank trees in an order of likelihood. To do that, the parser can integrate statistics derived from bracketed corpora. Because annotation is done by hand, frequencies captured by statistics reflect preferences of human beings.

There are many possible **probabilistic parsing** techniques. They all aim at finding an optimal analysis considering a set of statistical parameters. A major

difference between them corresponds to the introduction of lexical statistics or not – statistics on words as opposed to statistics on rules. We begin here with a description of nonlexicalized probabilistic context-free grammars, or PCFG.

12.5 A Description of PCFGs

A PCFG is a constituent context-free grammar where each rule describing the structure of a left-hand-side symbol is augmented with its probability $P(lhs \rightarrow rhs)$ (Charniak 1993). Table 12.6 shows a small set of grammar rules with imaginary probabilities.

According to figures in the table, the structure of a sentence consists 4 times out of 5 in a noun phrase and a verb phrase – $P(s \rightarrow np, vp) = 0.8$ – and 1 time out of 5 in a verb phrase – $P(s \rightarrow vp) = 0.2$. Such figures correspond in fact to conditional probabilities: knowing the left-hand-side symbol they describe proportions among the right-hand-side expansions. The probability could be rewritten then as

$$P(lhs \rightarrow rhs|lhs).$$

The sum of probabilities of all possible expansions of a left-hand-side symbol must be equal to 1.0.

Probabilities in Table 12.6 are fictitious and incomplete. A sentence has, of course, many more possible structures than those shown here. Real probabilities are obtained from syntactically bracketed corpora – treebanks. The probability of a given rule $lhs \rightarrow rhs_i$ is obtained by counting the number of times it occurs in the corpus and by dividing it by the count of all the expansions of symbol `lhs`.

$$P(lhs \rightarrow rhs_i|lhs) = \frac{Count(lhs \rightarrow rhs_i)}{\sum_j Count(lhs \rightarrow rhs_j)}.$$

Parsing with a PCFG is just the same as with a context-free grammar except that each tree is assigned with a probability. The probability for sentence S to have the parse tree T is defined as the product of probabilities attached to rules used to produce the tree:

$$P(T, S) = \prod_{rule(i) \text{ producing } T} P(rule(i)).$$

Let us exemplify probabilistic parsing for an ambiguous sentence using the grammar in Table 12.6. *Bring the meal of the day* has two possible parse trees, as shown in Table 12.7. We consider trees up to the verb phrase symbol only.

Table 12.6 A small set of phrase-structure rules augmented with probabilities, P

Rules			P	Rules			P
s	-->	np vp	0.8	det	-->	the	1.0
s	-->	vp	0.2	noun	-->	waiter	0.4
np	-->	det noun	0.3	noun	-->	meal	0.3
np	-->	det adj noun	0.2	noun	-->	day	0.3
np	-->	pronoun	0.3	verb	-->	bring	0.4
np	-->	np pp	0.2	verb	-->	slept	0.2
vp	-->	v np	0.6	verb	-->	brought	0.4
vp	-->	v np pp	0.1	pronoun	-->	he	1.0
vp	-->	v pp	0.2	prep	-->	of	0.6
vp	-->	v	0.1	prep	-->	to	0.4
pp	-->	prep np	1.0	adj	-->	big	1.0

Table 12.7 Possible parse trees for *Bring the meal of the day*

Parse trees	
T1:	vp(verb(bring), np(np(det(the), noun(meal)), pp(prep(of), np(det(the), noun(day)))))
T2:	vp(verb(bring), np(np(det(the), noun(meal))), pp(prep(of), np(det(the), noun(day))))

The probability of T_1 is defined as (Fig. 12.14):

$$
\begin{aligned}
&P(T_1, \text{Bring the meal of the day}) = \\
&P(vp \rightarrow v, np) \times P(v \rightarrow Bring) \times P(np \rightarrow np, pp) \times \\
&P(np \rightarrow det, noun) \times P(det \rightarrow the) \times P(noun \rightarrow meal) \times \\
&P(pp \rightarrow prep, np) \times P(prep \rightarrow of) \times P(np \rightarrow det, noun) \times \\
&P(det \rightarrow the) \times P(noun \rightarrow day) = \\
&0.6 \times 0.4 \times 0.2 \times 0.3 \times 1.0 \times 0.3 \times 1.0 \times 0.6 \times 0.3 \times 1.0 \times 0.3 = 0.00023328,
\end{aligned}
$$

and that of T_2 as (Fig. 12.15) as:

$$
\begin{aligned}
&P(T_2, \text{Bring the meal of the day}) = \\
&P(vp \rightarrow v, np, pp) \times P(v \rightarrow Bring) \times P(np \rightarrow det, noun) \times \\
&P(det \rightarrow the) \times P(noun \rightarrow meal) \times P(pp \rightarrow prep, np) \times P(prep \rightarrow of) \times \\
&P(np \rightarrow det, noun) \times P(det \rightarrow the) \times P(noun \rightarrow day) = \\
&0.1 \times 0.4 \times 0.3 \times 1.0 \times 0.3 \times 1.0 \times 0.6 \times 0.3 \times 1.0 \times 0.3 = 0.0001944.
\end{aligned}
$$

T_1 has a probability higher than that of T_2 and then corresponds to the most likely parse tree. Thus PCFG would properly disambiguate among alternative structures for this sentence. However, we can notice that PCFGs are certainly not flawless because they would not properly rank trees of *Bring the meal to the table*.

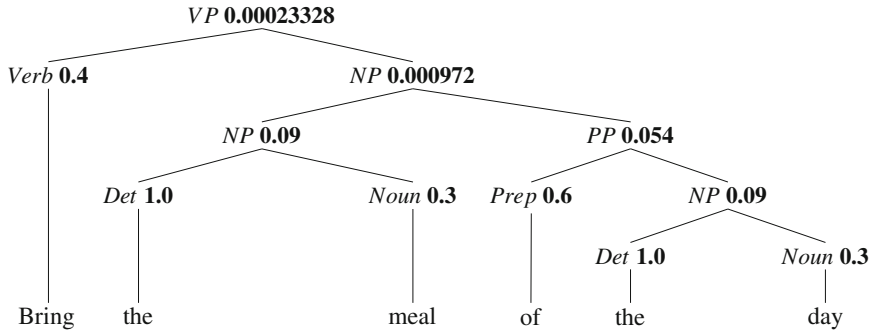


Fig. 12.14 Parse tree T_1 with nodes annotated with probabilities

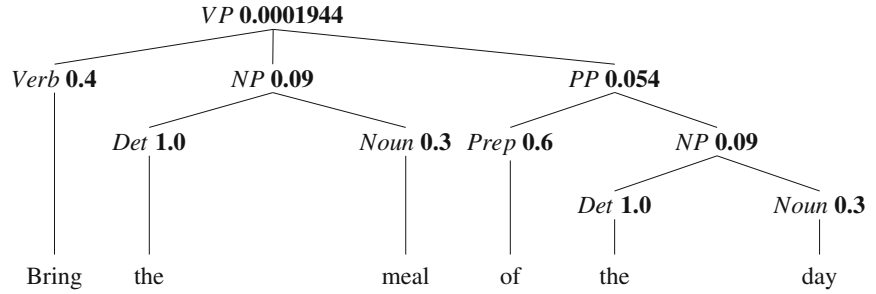


Fig. 12.15 Parse tree T_2 with nodes annotated with probabilities

12.5.1 The Bottom-Up Chart

Figures 12.14 and 12.15 show a calculation of parse tree probabilities using a bottom-up approach. Although it is possible to use other types of parsers, this strategy seems the most natural because it computes probabilities as it assembles partial parses. In addition, a chart would save us many recalculations. We will combine these techniques to build a probabilistic context-free parser. We introduce them in two steps. First, we present a symbolic bottom-up chart parser also known as the Cocke–Younger–Kasami (CYK) algorithm (Kasami 1965). We then extend it to probabilistic parsing in a next section.

The CYK algorithm uses grammars in Chomsky normal form (CNF, Chap. 11) where rules are restricted to two forms:

```
lhs --> rhs1, rhs2.
lhs --> [terminal_symbol].
```

However, the CYK algorithm can be generalized to any type of grammar (Graham et al. 1980).

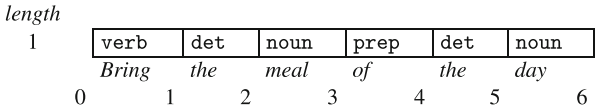


Fig. 12.16 Annotation of the words with their possible part of speech. Here words are not ambiguous

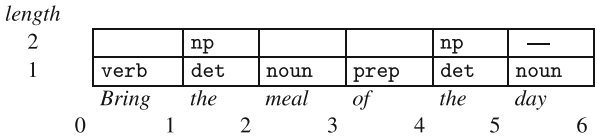


Fig. 12.17 Constituents of length 1 and 2

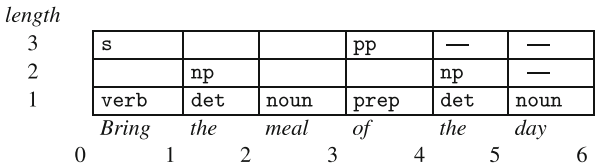


Fig. 12.18 Constituent of lengths 1, 2, and 3

Let N be the length of the sentence. The idea of the CYK parser is to consider constituents of increasing length from the words – length 1 – up to the sentence – length N. In contrast to the Earley parser, the CYK algorithm stores completely parsed constituents in the chart. It proceeds in two steps. The first step annotates the words with all their possible parts of speech. Figure 12.16 shows the result of this step with the sentence *Bring the meal of the day*. It results in chart entries such as `arc(v, [bring, '.'], 0, 1)`, `arc(det, [the, '.'], 1, 2)`, etc. This first step is also called the base case.

The second step considers contiguous pairs of chart entries that it tries to reduce in a constituent of length l, l ranging from 2 to N. Considering the rule

```
lhs --> rhs1, rhs2
```

the parser searches couples of arcs corresponding to `arc(rhs1, [..., '.'], i, k)` and `arc(rhs2, [..., '.'], k, j)` such that $i < k < j$ and $j - i = l$. It then adds a new arc: `arc(lhs, [rhs1, rhs2, '.'], i, j)` to the chart. Since constituents of length 2, 3, 4, ..., N are built in that order, it ensures that all constituents of length less than l have already been built. This second step is called the recursive case.

Let us consider constituents of length 2 in our example. We can add two noun phrases that we insert in the second row, as shown in Fig. 12.17. They span nodes 1–3 and 4–6. Since their length is 2, no constituent can start in cell 5–6, otherwise it would overflow the array. We insert the symbol "—" in the corresponding cell. This property is general for any constituent of length l and yields a triangular array.

The parse is complete and successful when length N, here 6, has been reached with the start symbol. Figure 12.18 shows constituents of length 3, and Fig. 12.19 shows the completed parse, where constituents are indexed vertically according to their length.

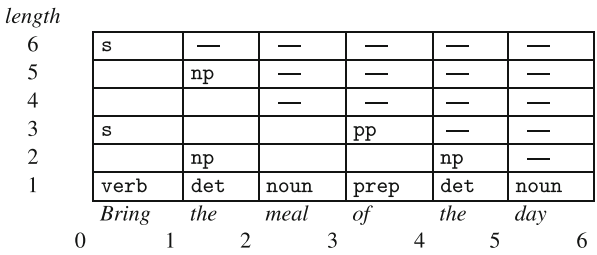


Fig. 12.19 The completed parse

12.5.2 The Cocke–Younger–Kasami Algorithm in Prolog

From the algorithm description, the Prolog implementation is relatively straightforward. We use two predicates to carry out the base case and the recursive case: `tag_words/5` and `cyk_loop/4`.

Since we use a CNF, we need to rewrite some rules in Table 12.6:

- `rule(np, [d, a, n])` is rewritten into `rule(np, [det, np])` and `rule(np, [a, n])`.
- `rule(vp, [v, np, pp])` is rewritten into `rule(vp, [vp, pp])` and `rule(vp, [v, np])`.
- `rule(s, [vp])` is rewritten into `rule(s, [vp, pp])` and `rule(s, [v, np])`.
- `rule(vp, [v])` is rewritten into `word(vp, [brought])`, `word(vp, [bring])`, and `word(vp, [slept])`.
- `rule(np, [pronoun])` is rewritten into `word(np, [he])`.

The parsing predicate `parse/2` consists of tagging the words (the base case) and calling the reduction loop (the recursive case).

```
parse(Sentence, Chart) :-
  tag_words(Sentence, 0, FinalPosition,[],WordChart),
  cyk_loop(2, FinalPosition, WordChart, Chart).
```

{`tag_words/3` tags the words with their possible parts of speech and adds the corresponding arcs using the `expand_chart/1` predicate.

```
tag_words([], FinalPos, FinalPos, Chart, Chart).
tag_words([Word | Rest], Pos, FinalPos, Chart,
    WordChart) :-
  NextPos is Pos + 1,
  findall(
    arc(LHS, [Word, '.'], Pos, NextPos),
    word(LHS, [Word]),
    ChartEntries),
  expand_chart(ChartEntries, Chart, NewChart),
  tag_words(Rest, NextPos, FinalPos, NewChart,
    WordChart).
```

`cyk_loop/4` implements the recursive case. It proceeds from length 2 to the sentence length and attempts to reduce constituents using `inner_loop/5`. The new constituents are added to the chart using `expand_chart/3`.

```
cyk_loop(FinalPos, FinalPos, Chart, FinalChart) :-
  inner_loop(0, FinalPos, FinalPos,Chart,FinalChart).
cyk_loop(Length, FinalPos, Chart, FinalChart) :-
  inner_loop(0, Length, FinalPos, Chart, ILChart),
  NextLength is Length + 1,
  cyk_loop(NextLength, FinalPos, ILChart,FinalChart).

inner_loop(StartPos, Length, FinalPos, Chart,Chart):-
  FinalPos < StartPos + Length.
inner_loop(StartPos, Length, FinalPos, Chart,
      ILChart) :-
  EndPos is StartPos + Length,
  findall(
    arc(LHS3, [LHS1, LHS2, '.'], StartPos, EndPos),
    (
      member(arc(LHS1, RHS1,StartPos,MidPos), Chart),
      member(arc(LHS2, RHS2, MidPos, EndPos), Chart),
      StartPos < MidPos,
      MidPos < EndPos,
      rule(LHS3, [LHS1, LHS2])
    ),
    ChartEntries),
  expand_chart(ChartEntries, Chart, NewChart),
  NextStartPos is StartPos + 1,
  inner_loop(NextStartPos, Length, FinalPos, NewChart,
      ILChart).
```

12.5.3 Adding Probabilities to the CYK Parser

Considering sentence S, the parser has to find the most likely tree T defined as the maximum probability

$$T(S) = \arg\max_{T} P(T, S).$$

Let us suppose that sentence S consists of constituents A and B: $S \rightarrow A, B$. The most likely parse tree corresponds to that yielding the maximum probability of both A and B. This is valid recursively for substructures of A and B down to the words.

To obtain most likely constituents for any given length, we need to maintain an array that stores the maximum probability for all the possible constituents spanning

all the word intervals $i \ldots j$ in the chart. In other words, that means that if there are two or more competing constituents with the same left-hand-side label spanning $i \ldots j$, the parser retains the maximum and discards the others. Let *lhs* be the constituent label, and $\pi(i, j, lhs)$ this probability.

The base case initializes the algorithm with part-of-speech probabilities:

$$\pi(i, i+1, part\_of\_speech \rightarrow word).$$

The recursive case maintains the probability of the most likely structure of `lhs`. It corresponds to:

$$\pi(i, j, lhs) = \max(\pi(i, k, rhs_1) \times \pi(k, j, rhs_2) \times P(lhs \rightarrow rhs_1, rhs_2)),$$

where the maximum is taken over all the possible values of k with $i < k < j$ and all possible values of rhs_1, rhs_2 with $lhs \rightarrow rhs_1, rhs_2$ in the grammar.

12.6 Evaluation of Constituent Parsers

12.6.1 Metrics

We have a variety of techniques to evaluate parsers. The PARSEVAL measures (Black et al. 1991) are the most frequently cited for constituent parsing. They take a manually bracketed treebank as the reference – the gold standard – and compare it to the results of a parser.

PARSEVAL uses a metric similar to that of information extraction, that is, recall and precision. Recall is defined as the number of correct constituents generated by the parser, i.e., exactly similar to that of the manually bracketed tree, divided by the number of constituents of the treebank. The precision is the number of correct constituents generated by the parser divided by the total number of constituents – wrong and correct ones – generated by the parser.

$$\text{Recall} = \frac{\text{Number of correct constituents generated by the parser}}{\text{Number of constituents in the manually bracketed corpus}}.$$

$$\text{Precision} = \frac{\text{Number of correct constituents generated by the parser}}{\text{Total number of constituents generated by the parser}}.$$

A third metric is the number of crossing brackets. It corresponds to the number of constituents produced by the parser that overlap constituents in the treebank. Table 12.8 shows two possible analyses of *Bring the meal of the day* with crossing brackets between both structures. The number of crossing brackets gives an idea of the compatibility between structures and whether they can be combined into a single structure.

Table 12.8 Bracketing of order *Bring the meal of the day* and crossing brackets

Bracketing	Crossing brackets
(((bring) (the meal)) (of the day))	( ) ( )
((bring) ((the meal) (of the day)))	( ) ( )

12.6.2 Performance of PCFG Parsing

PCFGs rank the possible analyses. This enables us to select the most probable parse tree and to evaluate it. Charniak (1997b) reports approximately 70 % recall and 75 % precision for this parsing method.

In terms of accuracy, PCFG parsing does not show the best performance. This is mainly due to its poor use of lexical properties. An example is given with prepositional-phrase attachment. While prepositional phrases attach to the preceding noun phrase 6 to 7 times out of 10 on average, there are specific lexical preferences. Some prepositions attach more often to verbs in general, while others attach to nouns. There are also verb/preposition or noun/preposition couples, showing strong affinities.

Let us exhibit them with the orders:

Bring the meal to the table

and

Bring the meal of the day

for which a parser has to decide where to attach prepositional phrases *to the table* and *of the day*. Alternatives are the verb *Bring* and the noun phrase *the meal*. Prepositional phrases headed by *of* attach systematically to the preceding noun phrase, here *the meal*, while *to* attaches here to the verb. Provided that part-of-speech annotation of both sentences is the same, the ratio

$$\frac{P(T1\,|\text{Bring the meal of the day})}{P(T2\,|\text{Bring the meal of the day})} = \frac{P(T1\,|\text{Bring the meal to the table})}{P(T2\,|\text{Bring the meal to the table})},$$

$$= \frac{P(vp \rightarrow v, np) \times P(np \rightarrow np, pp)}{P(vp \rightarrow v, np, pp)}.$$

depends only on rule probabilities and not on the lexicon. In our example, the PCFG does not take the preposition value into account: any prepositional phrase would always attach to the preceding noun, thus accepting an average error rate of 30 to 40 %.

12.7 Improving Probabilistic Context-Free Grammars

We saw in Sect. 11.4 that we could extend phrase-structure rules with grammatical features to get a better generation of sentences. We will describe here how we can use such features to improve probabilistic parsing as well.

We have reviewed a set of possible features to generate syntactically correct sentences: number, gender, etc. In this section, we will restrict ourselves to one feature, the grammatical case, and more specifically to the nominative and accusative cases. Both cases are used in Latin, German, and Russian, easy to understand, and correspond roughly to the subject and object functions in English and French.

Starting from this small grammar:

$$\begin{aligned} S &\rightarrow NP \quad VP \\ NP &\rightarrow DET \; N \\ VP &\rightarrow V \quad NP \end{aligned}$$

we introduce the nominative and accusative cases using unification grammars and the feature notation of Sect. 11.4, where *nom* stands for nominative, *i.e.* the subject, and *acc* for accusative, *i.e.* the object:

$$\begin{array}{ccccc} S & \rightarrow & NP & & VP \\ & & [case : nom] & & \\ NP & \rightarrow & DET & & N \\ [case : C] & & [case : C] & & [case : C] \\ VP & \rightarrow & V & & NP \\ & & & & [case : acc] \end{array}$$

As we want to build a probabilistic parser, we need to derive estimates from a corpus. The Penn Treebank is the standard candidate for English; however, it was not annotated with case, at least originally. A possible solution could have been to relabel it manually. This would have certainly required a sizable and costly amount of human effort.

An ingenious and automatic work around came from Johnson (1998), who encoded an equivalent of the case feature by marking the daughters of a rule with their parent using the caret symbol, ^. He could then subcategorize systematically the nonterminal symbols with about the same effect as inflecting the noun phrases with a case and capturing their function.

Using our small rule set, this means that we split *NP* into two symbols: *NP^S*, corresponding to a subject, and *NP^VP*, to an object. Following Johnson (1998), we relabel *VP* similarly and we rewrite the grammar:

```
s --> np, vp.
np --> det, n.
vp --> v, np.
```

as

```
s --> np_s, vp_s.
np_s --> det, n.
np_vp --> det, n.
vp_s --> v, np_vp.
```

where we used the underscore instead of the caret.

Johnson (1998) showed that a systematic parent annotation of PS rules could improve the precision and recall of PCFG by 8 %. This figure can further be increased by dividing phrase categories, as we have done with *NP*, and refining them with other grammatical features. See Klein and Manning (2003) for a description.

12.8 Lexicalized PCFG: Charniak's Parser

Charniak's parser (1997a) (second, improved version in Charniak 2000) builds on PCF grammars and introduces lexical preferences into the phrase-structure rules. To do this, Charniak creates phrase subcategories based on the lexical head of each phrase. It then one step beyond the subcategorization with features that we have studied in the previous section. We call this a rule lexicalization.

We have seen in previous sections that the attachment of prepositional phrases to a verb or a noun phrase was extremely sensitive to the value of the preposition. That is why instead of using the rule:

$$PP \rightarrow Prep\ NP$$

Charniak replaces them with as many rules as there are prepositions:

$$\begin{aligned} PP{:}of &\rightarrow of\ NP \\ PP{:}on &\rightarrow on\ NP \\ PP{:}with &\rightarrow with\ NP \end{aligned}$$

Charniak (1997a) extends this idea to all the constituents and Fig. 12.20 shows the parse tree of the sentence: *Corporate profits rose.* This category of grammars is called lexicalized PCFG, where we extract the head of constituents using percolation rules like the ones shown in Table 11.15.

As with PCFG, Charniak's parser maximizes the probability of a parse tree T given a sentence S:

$$T(S) = \arg\max_{T} P(T, S).$$

using the probabilities of the rules producing T.

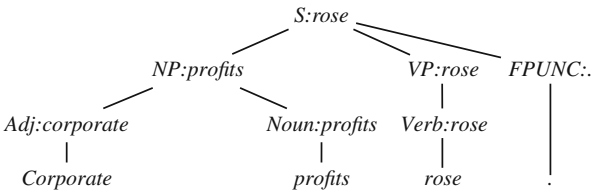


Fig. 12.20 Tree structure of *Corporate profits rose* (After Charniak (1997a))

The final probability is the product of the probabilities of all the constituents in the tree. These probabilities are similar to those in Table 12.6, but there are many more PS rules however as we have to pair each constituent category, for instance *NP*, with all its possible heads, here potentially all the nouns.

To compute the probability of a constituent, for instance *Corporate profits* to be a *NP* with *profits* as a head (*NP:profits*), Charniak's parser (1997a) uses the following steps:

1. Determine the probability of the constituent head, *profits* in the example.
   - This probability takes into account the previous parsing steps and is computed using this formula: $P(h|hp,t,tp)$, where h is the head of the constituent, hp, the head of the parent constituent, t, the type of the constituent and tp, the type of the parent constituent.
   - In Charniak's example, the probability of *profits* to be the head of the *NP*, *Corporate profits* is $P(profits|rose,NP,S)$.
2. Determine the probability of the constituent given the head.
   - As for the first term, we take the previous steps into account and we use the formula: $P(r|h,t,tp)$, where r is the phrase-structure rule producing the constituent.
   - In Charniak's example, the probability of $NP \rightarrow Adj\ Noun$ is computed using $P(NP \rightarrow Adj\ Noun|profits,NP,S)$
3. Apply the parser recursively to parse the subconstituents.

This yields a product of probabilities:

$$P(T,S) = \prod_{r \in T} P(r|h,t,tp)P(h|hp,t,tp)$$

that the parser maximizes to find the parse tree. The parser algorithm uses an extension to the CYK algorithm (Charniak et al. 1998).

The estimates of $P(r|h,t,tp)$ and $P(h|hp,t,tp)$ are impossible to compute given the current size of annotated corpora: The data is far too sparse. Charniak used a sophisticated linear interpolation to approximate both of them:

$$P(h|hp,t,tp) = \lambda_1 P(h|hp,t,tp) + \lambda_2 P(h|c_{hp},t,tp) + \lambda_3 P(h|t,tp) + \lambda_4 P(h|t)$$

and

$$P(r|h,t,tp) = \lambda_1 P(r|h,t,tp) + \lambda_2 P(r|h,t) + \lambda_3 P(r|c_h,t) \\ +\lambda_4 P(r|t,tp) + \lambda_5 P(r|t),$$

where c_{hp} are clusters of hp words linked by their behavior in $P(h|hp,t,tp)$.

12.9 Further Reading

Parsing techniques have been applied to compiler construction as well as to human languages. There are numerous references reviewing formal parsing algorithms, both in books and articles. Aho et al. (1986) is a starting point.

Most textbooks in computational linguistics describe parsing techniques for natural languages. Pereira and Shieber (1987), Covington (1994b), Gazdar and Mellish (1989), and Gal et al. (1989) introduce symbolic techniques and include implementations in Prolog. Allen (1994), Jurafsky and Martin (2008), and Manning and Schütze (1999) are other references that include surveys of statistical parsing. All these books mostly describe, if not exclusively, constituent parsing.

Prepositional phrase attachment is a topic that puzzled many of those adopting the constituency formalism. It often receives special treatment – a special section in books. For an introduction, see Hindle and Rooth (1993). Techniques to solve it involved the investigation of lexical preferences that probably started a shift of interest toward dependency grammars.

Statistical parsing is more recent than symbolic approaches. Charniak (1993) is a good account to probabilistic context-free grammars (PCFG). Manning and Schütze (1999) is a comprehensive survey of statistical techniques used in natural language processing. See also the two special issues of *Computational Linguistics* (1993, vol. 19, nos. 1 and 2). PCFGs were a first attempt to introduce probabilities into phrase–structure rules. Johnson (1998) and Klein and Manning (2003) showed they could be significantly improved with grammatical features. Although unlexicalized PCFGs are merely a reformulation of feature grammars, they had a significant impact in the constituent parsing community. Charniak (1997a, 2000) introduced an efficient lexicalized parser that still defines the state-of-the-art in constituent parsing. It is available for download from this location: http://cs.brown.edu/~ec/.

Quality of statistics and rules is essential to get good parsing performance. Probabilities are drawn from manually bracketed corpora, and their quality depends on the annotation and the size of the corpus. A key problem is sparse data. For a good review on how to handle sparse data, see Collins (1999). Symbolic rules can be tuned manually by expert linguists or obtained automatically using inductive logic techniques, either for constituents or dependencies. Zelle and Mooney (1997) propose an inductive logic programming method in Prolog to obtain rules from annotated corpora.

Exercises

12.1. The shift–reduce program we have seen stores words at the end of the list representing the stack. Subsequently, we use `append/3` to traverse the stack and match it to grammar rules. Modify this program so that words are added to the beginning of the list.

Hint: you will have to reverse the stack and the rules so that rule `s --> np, vp` is encoded `rule([vp, np | X], [s | X]).`

12.2. Trace the shift–reduce parser with a null symbol `word(d, [])` and describe what happens.

12.3. Modify the shift–reduce parser so that it can handle lists of terminal symbols such as `word(d, [all, the]).`

12.4. Complete arcs of Fig. 12.13 with the Earley algorithm.

12.5. Trace the Earley algorithm in Prolog with the sentence *Bring the meal of the day*.

12.6. In the implementation of Earley's algorithm, we represented dotted rules such as

```
np --> np • pp          [0, 2]
```

by Prolog facts such as

```
arc(np, [np, '.', pp], 0, 2).
```

This representation involves searching a dot in a list, which is inefficient. Modify the program so that it can use an arc representation, where the sequence of categories to the left and to the right of the dot are split into two lists, as with `arc(np, [np], [pp], 0, 2).`

12.7. The Earley chart algorithm accepts correct sentences and rejects ill-formed ones, but it does not provide us with the sentence structure. Write a `retrieve` predicate that retrieves parse trees from the chart.

12.8. Modify the Cocke, Younger, and Kasami Prolog program to include parsing probabilities to constituents in the chart.

12.9. Modify the Cocke, Younger, and Kasami Prolog program to produce the best parse tree as a result of the analysis. Hint: to retrieve the tree more easily, use an array of back pointers: an array storing for each best constituent over the interval `i ...j`, the rule that produced it, and the value of `k`.

Chapter 13
Dependency Parsing

13.1 Introduction

Parsing dependencies consists of finding links between heads (also called governors) and modifiers (or dependents) – one word being the root of the sentence (Fig. 13.1). In addition, each link can be annotated with a grammatical function.

There is a large array of techniques to parse dependencies. In this chapter, we introduce some of them in order of increasing complexity. We begin with an extension of shift–reduce to parse dependencies, and we describe how to use symbolic and machine-learning techniques to guide the parser. We then present other parsing strategies using constraint satisfaction and statistical lexical dependencies.

13.2 Evaluation of Dependency Parsers

Before we review dependency parsing techniques, let us first describe how we will evaluate them. As for constituents, we will compare the output of an automatic parser with its corresponding manual annotation. Most evaluation metrics follow Lin (1995), who proposed to consider the links between each word in the sentence and its head. The error count is then the number of words that are assigned a wrong head (governor). Figure 13.2 shows a manually annotated dependency tree of *Bring the meal to the table* and a possible parse. The error count is 1 out of 6 links and corresponds to the wrong attachment of *to*. Lin (1995) also described a method to adapt this error count to constituent structures. This error count is probably simpler and more intuitive than the PARSEVAL metrics.

P.M. Nugues, *Language Processing with Perl and Prolog*, Cognitive Technologies,
DOI 10.1007/978-3-642-41464-0_13, © Springer-Verlag Berlin Heidelberg 2014

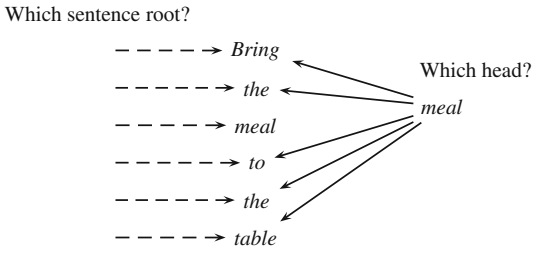


Fig. 13.1 Possible roots in the sentence *Bring the meal to the table* and heads for word *meal*. There are N possible roots, and each remaining word has theoretically $N - 1$ possible heads

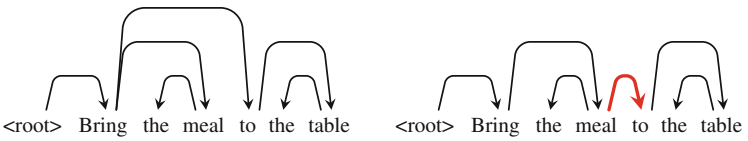


Fig. 13.2 Evaluation of dependency trees: the reference dependency tree (*left*), and a possible parse output (*right*)

13.3 Nivre's Parser

13.3.1 Extending the Shift–Reduce Algorithm to Parse Dependencies

Nivre (2003) proposed a dependency parser that creates a graph that he proved to be both projective and acyclic. The parser is an extension to the shift–reduce algorithm that we saw in Sect. 12.2. As with the regular shift–reduce, Nivre's parser uses a stack S and a list of input words W. However, instead of finding constituents, it builds a set of arcs A representing the graph of dependencies.

Nivre's parser uses two operations in addition to shift and reduce, left-arc and right-arc. Using the notation $(\text{head}, \text{modifier})$ to denote an arc from a head to a modifier:

Shift pushes the next input word onto the stack.

Reduce pops the top of the stack. We also add a constraint to the reduce operation to check that the top of the stack has a head and then ensures that the graph is connected.

Left-arc adds an arc (n', n) from the next input word n' to the top of the stack n and reduces n from the top of the stack. We set a condition to this operation: there must not be an arc (n'', n) already in the graph. Without it, the top of the stack would have two or more heads.

Right-arc adds an arc (n, n') from the top of the stack n to the next input word n' and pushes n' on the top of the stack.

Table 13.1 The parser transitions, where W is the initial word list; I, the current input word list; A, the graph of dependencies; and S, the stack. The triple $\langle S, I, A\rangle$ represents the parser state. n, n', and n'' are lexical tokens. The pair (n', n) represents an arc from the head n' to the modifier n

	Parser transitions		
Actions	**State before**	**State after**	**Conditions**
Initialization		$\langle nil, W, \emptyset\rangle$	
Termination	$\langle S, nil, A\rangle$		
Left-arc	$\langle n\|S, n'\|I, A\rangle$	$\langle S, n'\|I, A \cup \{(n', n)\}\rangle$	$\nexists n''(n'', n) \in A$
Right-arc	$\langle n\|S, n'\|I, A\rangle$	$\langle n'\|n\|S, I, A \cup \{(n, n')\}\rangle$	
Reduce	$\langle n\|S, I, A\rangle$	$\langle S, I, A\rangle$	$\exists n'(n', n) \in A$
Shift	$\langle S, n\|I, A\rangle$	$\langle n\|S, I, A\rangle$	

Table 13.1 shows the start and final parser states as well as the four transitions and their conditions.

13.3.2 Parsing an Annotated Corpus

Nivre (2006) proved that for each sentence with a projective dependency graph, there is a transition sequence that enables his parser to generate this graph. He called this procedure gold-standard parsing, because it corresponds to the sequence of parsing transitions taken in the set {left-arc, right-arc, reduce, shift} that produces the manually-obtained gold-standard graph. The parser has a linear complexity, and the number of transitions needed to parse a sentence is at most $2n - 1$, where n is the length of the sentence.

To parse a manually-constructed graph, we could set the graph as a search goal and use the Prolog backtracking mechanism to find the transitions. Although this is possible, there are more efficient methods. Given a dependency graph, we can formulate simple conditions on the stack and the current input list to execute left-arc, right-arc, shift, or reduce. The two first conditions on left-arc and right-arc are obvious:

- We execute a left-arc if the top of the stack and the next word in the list are linked by a left arc in the gold-standard graph.
- We execute a right-arc if the top of the stack and the next word in the list are linked by a right arc in the gold-standard graph.

The reduce condition is slightly more complex. We execute it when the gold-standard graph contains an arc in either direction between the next word in the list and a word in the stack, below the top. Finally, we execute a shift when no other action can be carried out. The operation we have just explained that determines which transition to apply given a certain parser state is called an **oracle**. The oracle at a given step of the parsing process is summarized by the condition below, where *TOP* is the top of the stack, *FIRST*, the first token of the input list, and *arc*, the relation holding between a head and a dependent:

Table 13.2 The transition sequence to apply to the sentence *The waiter brought the meal* to produce its dependency graph. The graph uses *left* or *right arrows* to give the direction of the dependency

Trans.	Stack	Queue	Graph
start	∅	ROOT the waiter brought the meal	{}
sh			
	ROOT	the waiter brought the meal	{}
sh			
	the ROOT	waiter brought the meal	{}
la			
	ROOT	waiter brought the meal	{the ← waiter}
sh			
	waiter ROOT	brought the meal	{the ← waiter}
la			
	ROOT	brought the meal	{the ← waiter, waiter ← brought}
ra			
	brought ROOT	the meal	{the ← waiter, waiter ← brought, ROOT → brought}
sh			
	the brought ROOT	meal	{the ← waiter, waiter ← brought, ROOT → brought}
la			
	brought ROOT	meal	{the ← waiter, waiter ← brought, ROOT → brought, the ← meal}
ra			
end	meal brought ROOT	[]	{the ← waiter, waiter ← brought, ROOT → brought, the ← meal, brought → meal}

1. **if** $arc(TOP, FIRST) \in A$, **then** right-arc;
2. **else if** $arc(FIRST, TOP) \in A$, **then** left-arc;
3. **else if** $\exists k \in Stack, arc(FIRST, k) \in A$ or $arc(k, FIRST) \in A$, **then** reduce;
4. **else** shift.

As an example, let us examine the oracle in action with the sentence:

The waiter brought the meal.

The dependency graph can be represented by the set of links:

$$\{\text{the} \leftarrow \text{waiter}, \text{waiter} \leftarrow \text{brought}, \text{ROOT} \rightarrow \text{brought}, \text{the} \leftarrow \text{meal}, \text{brought} \rightarrow \text{meal}\},$$

where we use left or right arrows to give the direction of the dependency, and the dummy word ROOT to indicate the root of a sentence. Knowing this graph, we can invoke the oracle to find the transition sequence. Table 13.2 shows the parser

states in the form of a stack, an input list, and the graph in construction. We start with an empty list of actions, and we ask the oracle at each step of the parsing procedure: What is the next action? When we reach the end of the input list, we have the complete transition sequence,

```
[sh, sh, la, sh, la, ra, sh, la, ra]
```

and we can check that the parser has built a graph that is equal to the one we constructed manually.

13.3.3 Nivre's Parser in Prolog

Nivre's parser is simple to understand, and its implementation in Prolog is easy. Let us call the top-level predicate

```
nivre_parser(+Sentence, ?Operations, ?RefGraph)
```

where `Sentence` is the input sentence, `RefGraph`, the manually-annotated dependency graph, and `Operations`, the sequence of transitions that produces this graph. We will design the program to work with two modes. The first one will use `Sentence` and `RefGraph` as input, and for each projective graph, the program will output a list of transitions. The second mode will use `Sentence` and `Operations` as input to produce a graph.

We represent the input sentences as lists of words with their corresponding position, form, and part of speech, `w([id=ID, form=FORM, postag=POSTAG])`, and we use a variation of the CONLL format described in Sect. 11.9.2 to encode dependencies, `w([id=ID, form=FORM, head=HEAD, deprel=DEPREL])`. The input sentence *The waiter brought the meal* is represented as:

```
[w([id=0, form=root, postag='ROOT']),
w([id=1, form=the, postag='DT']),
w([id=2, form=waiter, postag='NN']),
w([id=3, form=brought, postag='VBD']),
w([id=4, form=the, postag='DT']),
w([id=5, form=meal, postag='NN'])]
```

and its dependency graph as:

```
[w([id=1, form=the, head=2, deprel=det]),
w([id=2, form=waiter, head=3, deprel=sub]),
w([id=3, form=brought, head=0, deprel=root]),
w([id=4, form=the, head=5, deprel=det]),
w([id=5, form=meal, head=3, deprel=obj])]
```

Finally, the `Operations` list will consist of symbols corresponding to the initials of the transitions: {`la`, `ra`, `re`, `sh`}. For example, the sequence, shift, right-arc, left-arc, reduce, will be represented by `[sh, ra, la, re]`.

The Four Transitions

The Prolog program uses four transitions shown in Table 13.1 that we transcribe as four predicates: `left_arc/6`, `right_arc/6`, `shift/4`, and `reduce/3`.

The `left_arc/6` predicate adds an arc to the graph linking the first word of the list to the top of the stack with the condition that the first word has no head in the current graph.

```
% left_arc(+Words, +Stack, -NewStack, +Graph,
    -NewGraph)
left_arc([W | _], [T | Stack], Stack, Graph,
    [w([id=IDT, form=FORMT, head=IDW, deprel=_]) |
    Graph]) :-
  W = w([id=IDW | _]),
  T = w([id=IDT, form=FORMT | _]),
  \+ member(w([id=IDT, form=FORMT | _]), Graph).
```

The `right_arc/6` predicate adds an arc to the graph linking the top of the stack to the first word of the list with the condition that the top of the stack has no head in the current graph.

```
% right_arc(+Words, -NewWords, +Stack, -NewStack,
%  +Graph, -NewGraph)
right_arc([W | Words], Words, [T | Stack],
    [W, T | Stack], Graph, [w([id=IDW, form=FORMW,
      head=IDT, deprel=_]) | Graph]) :-
  W = w([id=IDW, form=FORMW | _]),
  T = w([id=IDT | _]),
  \+ member(w([id=IDW, form=FORMW | _]), Graph).
```

The `reduce/3` predicate reduces the `Stack` provided that the word has a head already in the graph.

```
% reduce(+Stack, -NewStack, +Graph)
reduce([T | Stack], Stack, Graph) :-
  T = w([id=IDT, form=FORMT | _]),
  member(w([id=IDT, form=FORMT | _]), Graph).
```

The `shift/4` predicate removes the next word from the list currently being parsed and pushes it on the top the stack. Here we set it as the head of the `Stack` list to produce a `NewStack`:

```
% shift(+Words, -NewWords, +Stack, -NewStack)
shift([First | Words], Words, Stack, [First | Stack]).
```

The Oracle

The oracle determines which transition to apply given a certain parser state. The `oracle/4` predicate uses the algorithm described in Sect. 13.3.2 and unifies `Operation` with a value in the set {`la`, `ra`, `re`, `sh`}. However, the first Prolog rule checks whether `Operation` is already instantiated when the predicate is called and if yes, it does not execute the algorithm. This will enable the program to work with two modes: determine a transition sequence from a graph, and also build a graph from a given transition sequence.

```
% oracle(+Words, +Stack, +Graph, -Operation)
% Predicts the next transition from the
   manually-annoted graph
oracle(_, _, _, Op) :-
  nonvar(Op),
  !.
oracle([W | _], [T | _], Graph, la) :-
  T = w([id=IDT, form=FORMT | _]),
  W = w([id=IDW | _]),
  member(w([id=IDT, form=FORMT, head=IDW | _]),
   Graph),
  !.
oracle([W | _], [T | _], Graph, ra) :-
  T = w([id=IDT | _]),
  W = w([id=IDW, form=FORMW | _]),
  member(w([id=IDW, form=FORMW, head=IDT | _]),
   Graph),
  !.
oracle([W | _], [_ |Stack], Graph, re) :-
  member(K, Stack),
  K = w([id=IDK, form=FORMK | _]),
  W = w([id=IDW, form=FORMW | _]),
  (
  member(w([id=IDK, form=FORMK, head=IDW | _]), Graph)
  ;
  member(w([id=IDW, form=FORMW, head=IDK | _]), Graph)
  ),
  !.
oracle(_, _, _, sh).
```

The Top-Level Predicate

The top-level predicate calls an auxiliary predicate that stores the stack and the current graph. If it fails to produce a transition sequence, it returns a list

containing the atom `fail`. We can use the program in two ways with the modes: `nivre_parser(+Sentence, -Transitions, +Graph)` and `nivre_parser(+Sentence, +Transitions, -Graph)`.

```
nivre_parser(Sentence, Ops, RefGraph) :-
  nivre_parser(Sentence, [], Ops, [], RefGraph).
nivre_parser(_, [fail], _).
```

The auxiliary predicate consists of three rules, where the first two represent terminal conditions, and the third is the general recursive case. The terminal conditions correspond respectively to the two possible modes. The first rule is used when the program produces a transition sequence from a reference graph. The terminal condition is met when the queue is empty. We check additionally that the two graphs – the reference graph and the graph built by the parser – are equal. This is always the case when the reference graph is well-formed and projective. The second rule corresponds to the application of a transition sequence, and the output is the graph. Finally, the recursive rule calls the oracle and executes the predicted transition.

```
nivre_parser([], _, [], CurGraph, RefGraph) :-
  nonvar(RefGraph),
  !,
  subset(RefGraph, CurGraph),
  subset(CurGraph, RefGraph).
nivre_parser([], _, [], Graph, Graph).
nivre_parser(Words, Stack, [Op | Ops], Graph,
  RefGraph) :-
  oracle(Words, Stack, RefGraph, Op),
  execute_action(Op, Words, NWords, Stack, NStack,
    Graph,NGraph),
  nivre_parser(NWords, NStack, Ops, NGraph,
    RefGraph).
```

The `execute_action/7` predicate calls the predicted operation and produces a new parser state. Its purpose is merely to select the arguments of the transitions.

```
% execute_action(+Op, +Words, -NewWords, +Stack,
% -NewStack, +Graph, NewGraph)
execute_action(la, Words, Words, Stack, NStack, Graph,
    NGraph) :-
  left_arc(Words, Stack, NStack, Graph, NGraph).
execute_action(ra, Words, NWords, Stack, NStack,
    Graph,NGraph) :-
  right_arc(Words, NWords, Stack, NStack, Graph,
    NGraph).
execute_action(re, Words, Words, Stack, NStack, Graph,
```

```
    Graph) :-
  reduce(Stack, NStack, Graph).
execute_action(sh, Words, NWords, Stack, NStack,
    Graph,Graph) :-
  shift(Words, NWords, Stack, NStack).
execute_action(Op, _, _, _, _, _, _) :-
  \+ member(Op, [la, ra, re, sh]),
  write('Illegal action. Returning'), nl.
```

Running the Parser

Applying the parser to *The waiter brought the meal* yields:

```
?- nivre_parser([w([id=0, form=root, postag='ROOT']),
    w([id=1, form=the, postag='DT']), ...], A,
        [w([id=1, form=the, head=2, deprel=det]),
          w([id=2, form=waiter, ...]).
A = [sh, sh, la, sh, la, ra, sh, la, ra] .
```

Conversely, given this transition sequence, the parser produces the original dependency graph.

13.4 Guiding Nivre's Parser

So far, we gave the parser a solution in the form of a reference graph to find the transition sequence. In most practical cases, this is not what we want. We generally have a sentence as input, and our goal is to automatically build a graph. We will now introduce techniques to carry this out. We will start with symbolic rules that resemble phrase-structure rules for dependencies: dependency rules. We will then move on to consider machine-learning techniques so that we can train our parser on manually-annotated corpora. The combination of machine learning and Nivre's parser produces highly efficient systems.

13.4.1 Parsing with Dependency Rules

Dependency Rules

Writing dependency rules or *D*-rules consists in describing possible dependency relations between word categories (Covington 1990): typically a head part of speech to a dependent part of speech (Fig. 13.3).

Fig. 13.3 Examples of D-rules

```
1. determiner ← noun.   4. noun ← verb.
2. adjective ← noun.    5. preposition ← verb.
3. preposition ← noun.  6. verb ← root.
```

These rules mean that a determiner can depend on a noun (1) (or that a noun can be the head of a determiner), an adjective can depend on a noun (2), and a noun can depend on a verb (4). The rules express ambiguity. A preposition can depend either on a verb (5) as in *Bring the meal to the table* or on a noun (3) as in *Bring the meal of the day*. Finally, rule 6 means that a verb can be the root of the sentence.

In our example, D-rules use parts of speech, which means that before parsing, we must use a POS tagger to annotate each word of the input list. D-rules can also involve the lexical value of the words or their semantic category or as, for instance,

of ← `noun.`

which means that the word *of* depends on a noun. When the rules involve word values, they are said to be **lexicalized**.

D-rules are often related to one or more functions. The first rule in Fig. 13.3 expresses the determinative function, the second one is an attributive function, and the third rule can be a subject, an object, or an indirect object function. Using a unification-based formalism, rules can encapsulate functions, as in:

$$\begin{bmatrix} category : noun \\ number : N \\ person : P \\ case : nominative \end{bmatrix} \leftarrow \begin{bmatrix} category : verb \\ number : N \\ person : P \end{bmatrix}$$

which indicates that a noun marked with the nominative case can depend on a verb. In addition, the noun and verb share the person and number features. Unification-based D-rules are valuable because they can easily pack properties into a compact formula: valence, direction of dependency relation (left or right), lexical values, etc. (Covington 1989; Koch 1993).

Using D-Rules with Nivre's Parser

We can use D-rules to guide Nivre's parser. We need first to write a grammar, a set of rules, for the language we want to process and write a guide predicate so that at a given point of the analysis it will examine the grammar and select a transition among the four possible ones.

Most languages have directional constraints for the dependencies. For instance, a determiner is always before the noun in English, French, and German. We can formulate these constraints using oriented D-rules to represent left, $POS(n') \leftarrow POS(n)$, and right, $POS(n) \rightarrow POS(n')$, dependencies. In the first case, the head is to the left of the dependent, and in the second one, it is the opposite. Oriented D-rules reduce ambiguity significantly.

Table 13.3 Conditions on the left-arc and right-arc transitions to run Nivre's parser with D-rules

Actions	Parser transitions	Conditions
Left-arc	$\langle n\|S, n'\|I, A\rangle \rightarrow \langle S, n'\|I, A \cup \{(n', n)\}\rangle$	$POS(n) \leftarrow POS(n') \in R$ $\nexists n''(n'', n) \in A$
Right-arc	$\langle n\|S, n'\|I, A\rangle \rightarrow \langle n'\|n\|S, I, A \cup \{(n, n')\}\rangle$	$POS(n) \rightarrow POS(n') \in R$

The guide links the D-rules to the parsing actions. To carry out a left-arc, the grammar must contain the rule $POS(n) \leftarrow POS(n')$ and to carry a right-arc, the grammar must contain the rule $POS(n) \rightarrow POS(n')$. Table 13.3 shows the new conditions to run Nivre's parser with D-rules.

The D-rule constraints are not sufficient to write a complete guide as the transition selection is still ambiguous. We can always apply a shift, and the grammar may contain conflicting rules; for instance, a preposition can depend on a noun and a noun can depend on a preposition. When two or more transitions are applicable, we need a procedure to select a unique one. In its original article, Nivre (2003) experimented with three parsing strategies that depended on the transition priorities. The first two are:

- The parser uses the constant priorities for the transitions: left-arc > right-arc > reduce > shift.
- The second parser uses the constant priorities left-arc > right-arc and a rule to resolve shift/reduce conflicts. If the top of the stack can be a transitive head of the next input word according to the grammar, then shift; otherwise reduce.

Both strategies are easy to implement.

D-Rules and Nivre's Parser in Prolog

Before we start writing a guide, we need to write a grammar. We represent the D-rules with a `drule/4` predicate, where each rule contains the parts of speech of the head and the dependent, the function, and the dependency direction. A simple grammar of English using the Penn Treebank tagset will be:

```
%drule(+HeadPOS, +DepPOS, +Function, +Direction)
drule('ROOT', 'VBD', root, right).
drule('NN', 'DT', determinative, left).
drule('NN', 'JJ', attribute, left).
drule('VBD', 'NN', subject, left).
drule('VBD', 'PRP', subject, left).
drule('VBD', 'NN', object, right).
drule('VBD', 'PRP', object, right).
drule('VBD', 'IN', adv, _).
drule('NN', 'IN', pmod, right).
drule('IN', 'NN', pcomp, right).
```

Let us now write a guide predicate that resembles those proposed by Nivre (2003). It consists of five rules that describe transition priorities. The first rule tries a left-arc. It looks for a D-rule that links the top of the stack to the first word in the queue. If this fails, the second rule tries a right-arc. It looks for a D-rule that links the first word in the queue to the top of the stack. If we cannot execute these actions, either because there is no D-rule or because their conditions are not met, we try shift with a lookahead condition that the top of the stack can be the head of the second-next word in the queue. We look for a corresponding D-rule. If this does not work, we try reduce and then shift.

```
% guide(+Words, +Stack, -Operation)
guide([W | _], [T | _], la) :-
  T = w([id=_, form=_, postag=POST | _]),
  W = w([id=_, form=_, postag=POSW | _]),
  drule(POSW, POST, _, left).
guide([W | _], [T | _], ra) :-
  T = w([id=_, form=_, postag=POST | _]),
  W = w([id=_, form=_, postag=POSW | _]),
  drule(POST, POSW, _, right).
guide([_, W | _], [T | _], sh) :-
  T = w([id=_, form=_, postag=POST | _]),
  W = w([id=_, form=_, postag=POSW | _]),
  drule(POST, POSW, _, right).
guide(_, _, re).
guide(_, _, sh).
```

We need to slightly modify `nivre_parser/5` to run it with the guide:

```
nivre_parser([], _, [], Graph, Graph).
nivre_parser(Words, Stack, [Op | Ops], Graph,
  RefGraph) :-
  guide(Words, Stack, Op),
  execute_action(Op, Words, NWords, Stack, NStack,
    Graph,NGraph),
  nivre_parser(NWords, NStack, Ops, NGraph,
    RefGraph).
```

Applying the parser to *The waiter brought the meal*, where each word was tagged with its part of speech, yields the correct graph:

```
[w([id=5, form=meal, head=3, deprel=_G1015]),
 w([id=4, form=the, head=5, deprel=_G915]),
 w([id=3, form=brought, head=0, deprel=_G775]),
 w([id=2, form=waiter, head=3, deprel=_G675]),
 w([id=1, form=the, head=2, deprel=_G535])]
```

with the transition sequence `[sh, sh, la, sh, la, ra, sh, la, ra]`.

Evaluating the Parser

Using annotated corpora, it is possible to derive automatically a grammar of D-rules, then run and assess the parser. There are a few freely available corpora that we can download. Talbanken05 in Swedish (Einarsson 1976; Nilsson et al. 2005) is one example. It contains annotated Swedish sentences and was used in the CoNLL 2006 shared task (Buchholz and Marsi 2006). It comes with a training set and a test set as well as an evaluation procedure that is comparable to the one described in Sect. 13.2.

We can use the training set to extract dependency rules and then run the parser on the test set. Nivre's parser does not guarantee to produce connected graphs. This means that some words from the input sentence will have no head in the graph. In this case, we assign them a `ROOT` head. The result score will depend on the number of rules we use. With the 100 most frequent ones in the CoNLL 2006 training set, we reach an attachment score of about 57 % in the test set.

Although this result is far from the state of the art, we obtained it with a minimal guide. There are many ways to improve it, and this means that using Nivre's parser with D-rules is a viable strategy. Here are a list of possible improvements: introduce lexicalized D-rules, constrain the parser to produce connected graphs, and constrain the graph to have only one root. We leave these improvements as an exercise (Exercise 13.1).

13.4.2 Using Machine-Learning Techniques

D-rules provide a simple way to control a dependency parser. However, they use limited information to make a decision: the part of speech of two words, the top of the stack, and the first word in the queue. Most current implementations of Nivre's parser use a richer set of features and hence rely on machine-learning techniques to implement the guide.

We now describe techniques that are similar to those we used with chunking (see Sect. 10.7.3) and that fit the sequential nature of Nivre's parser. We modify the guide so that before each transition it extracts features from the parser state. The features represent a sort of context to the transition and, using them as an input to a classifier, the guide predicts the next transition.

We build the classifier from a data set using a machine-learning algorithm. We collect a set of transition contexts from an annotated corpus using gold-standard parsing. This produces a list of feature values and the corresponding transition. We then automatically train a classifier such using ID3, logistic regression, or support vector machines (Chap. 4) that we have embedded in the guide.

Table 13.4 Feature vectors extracted while parsing the sentence *The waiter brought the meal*. In the *left* part of the table, the parser prints the parts of speech of the top of the stack and next in the queue before each transition. In the *middle* part, the parser prints two words from the stack and the queue

Stack	**Queue**	**Stack**		**Queue**		
POS(T_0)	POS(Q_0)	POS(T_0)	POS(T_{-1})	POS(Q_0)	POS(Q_{+1})	**Transitions**
nil	ROOT	nil	nil	ROOT	DT	sh
ROOT	DT	ROOT	nil	DT	NN	sh
DT	NN	DT	ROOT	NN	VBD	la
ROOT	NN	ROOT	nil	NN	VBD	sh
NN	VBD	NN	ROOT	VBD	DT	la
ROOT	VBD	ROOT	nil	VBD	DT	ra
VBD	DT	VBD	ROOT	DT	NN	sh
DT	NN	DT	VBD	NN	nil	la
VBD	NN	VBD	ROOT	NN	nil	ra

Predicting the Parser Transitions

We saw that gold-standard parsing could produce a transition sequence from a well-formed projective graph, and that the output sequence would exactly generate the input graph. While parsing, each transition has a corresponding parser state from which we can extract information in the form of feature vectors. To train a classifier, the idea is to associate a feature vector with the next transition.

The simplest features correspond to words' parts of speech on the top of the stack, POS(T_0), and the next word in the input queue, POS(Q_0). This is more or less the information provided by D-rules. A generalization is straightforward, and we can use more data from the stack or the queue and extend the size of the vector from two to four or more. We can also lexicalize the features and use the word forms in addition to their parts of speech. Table 13.4 shows feature vectors extracted while parsing the sentence:

> The waiter brought the meal

annotated as

> root/ROOT the/DT waiter/NN brought/VBD the/DT meal/NN.

The vectors use the part of the speech of the words and their dimension is two, POS(T_0), POS(Q_0), and four, POS(T_0), POS(T_{-1}), POS(Q_0), POS(Q_{+1}).

After the data is collected from an annotated corpus, we can apply a training procedure to create a 4-class classifier. Once trained, given a feature vector the classifier will choose a transition in the set {`la`, `ra`, `sh`, `re`}. We can then embed this classifier in the guide of Nivre's parser. When parsing a sentence, the parser extracts the current context after each transition and asks the guide to predict the next one. Support vector machines are among the most effective training algorithms we can use. In the next section, we use the C4.5 implementation of Weka (Sect. 4.3.2). The C4.5 training procedure is fast and produces decision trees that are easy to understand.

Table 13.5 Parsing performance using different feature sets on Swedish data. The training data was extracted from projective sentences in the CoNLL 2006 training set. The decision trees were trained using the C4.5 implementation available from the Weka environment. Words that had no head after parsing were assigned the ROOT head. The evaluation was carried out on the test set using the CoNLL 2006 script, which uses the attachment score. See Sect. 13.2 for a definition. The annotated data as well as the evaluation script are available from the CoNLL 2006 web page (After data provided by Buchholz and Marsi (2006))

Stack		**Queue**			**Constraints**			**Graph**	
POS(T_0)	POS(T_{-1})	POS(Q_0)	POS(Q_{+1})	POS(Q_{+2})	LA	RA	RE	LMS	Score
•	–	•	–	–	–	–	–	–	64.57
•	–	•	–	–	–	–	–	•	67.42
•	–	•	–	–	•	•	•	–	72.83
•	–	•	–	–	•	•	•	•	73.41
•	•	•	•	•	–	–	–	–	78.05
•	•	•	•	•	–	–	–	•	78.71
•	•	•	•	•	•	•	•	–	81.30
•	•	•	•	•	•	•	•	•	81.64

Feature Vectors and Parsing Performance

The feature set is a key factor in the parsing performance, and designing a good set is a very significant issue in practice. To realize it, we will experiment with our parser with two words in the stack and three words in the queue, and we will compare the results with a context involving just one word. We will also include the leftmost dependent of the top of the stack.

To help the guide, we will extract three additional Boolean parameters to check that the transition conditions are met: "can do left-arc", "can do right-arc", and "can do reduce". These features are intended to have the classifier model constraints on transitions. Hopefully, it will learn them and, as far as possible, avoid predicting illegal transitions. If this nevertheless happens and the classifier predicts a transition that violates the constraints, the guide will fall back to the priorities `la` > `ra` > `re` > `sh`. In total, the sets will have from two to eight parameters.

Table 13.5 shows the scores and hints that larger feature sets yield better results. This rule is true in general until we reach a ceiling in the size of the set. Using, say, 20 words from the queue would probably not improve the parser performance. In addition, there is a danger with larger sets to be overadapted to the training data. The classifier will then tend to learn specific properties of the data it is trained on and be biased toward it. An overadapted feature set will yield superior scores on training data, but possibly have inferior results on a different corpus. We call this an **overfit**.

The best score we obtain in Table 13.5 is 81.64. This is below the top score of the CoNLL 2006 shared task, which was 89.54. However, it is a good result given that we have only used parts of speech and that C4.5 is not as efficient as support vector machines. We can easily improve the performance using lexical features, such as the word form or the lemma. In fact, lexicalization has a strong impact on the parsing performance. Many attachment ambiguities can only be resolved by the knowledge

Table 13.6 Model for the feature set, where *top* denotes the top of the stack and *next*, the next in the queue. The feature names follow the CoNLL format described in Table 11.12 (After Nivre et al. (2006))

Location	Description	FORM	LEMMA	CPOS	POS	FEATS	DEPREL
Stack	*top*	•	•	•	•	•	•
Stack	$top - 1$				•		
Input	*next*	•	•	•	•	•	
Input	$next + 1$	•			•		
Input	$next + 2$				•		
Input	$next + 3$				•		
Graph	Head of *top*	•					
Graph	Leftmost dependent of *top*						•
Graph	Rightmost dependent of *top*						•
Graph	Leftmost dependent of *next*						•

of the lexical values. As an example of possible extensions, Nivre et al. (2006) proposed an effective feature set shown in Table 13.6.

Finding Grammatical Functions

We saw in Chap. 11 that the grammatical functions form an important layer in the description of dependency relations. In addition, the analysis of functions is essential in many applications. However, so far, our parser has only created the arcs of the dependency graphs. The time has now come to see how we can identify functions.

From a technical viewpoint, functions are just labels to add to the arcs, and there are two main ways to assign them:

1. We can first modify the guide and use a two-step classifier: the first one predicts the next transition as before and if it is a left-arc or a right-arc, a second classifier predicts the function. This second classifier will have as many classes as there are functions and can be trained from the functions extracted from gold-standard parsing.
2. A second possibility is to extend the left-arc and right-arc transitions to include the function. The transition sequence to parse *The waiter brought the meal* is shown in Table 13.2: `[sh, sh, la, sh, la, ra, sh, la, ra]`. The augmented transitions to assign the functions will be:
   `[sh, sh, la-det, sh, la-sub, ra-root, sh, la-det, ra-obj]`.
   We will just need one classifier to predict both the transition and the function. We train this classifier from the new transitions we collect with the gold-standard parsing procedure.

The annotation format of CoNLL 2006 and 2007 makes provision for grammatical functions. This means that we can train our transition and function classifiers on the same corpus. Moreover, classifiers can more or less use the same features. See Table 13.6 for an example. The evaluation is carried out the same way as for the bare dependency graphs, but this time it measures the labeled attachment score: the proportion of arcs that have both a correct head and a correct label. This label is called DEPREL in the CoNLL format.

Parsing Nonprojective Links

Identifying functions has an interesting side effect: it enables Nivre's parser to handle nonprojectivity. By construction, this parser is limited to projective graphs. However, we saw in Sect. 11.10.2 that is it possible to projectivize nonprojective graphs. We also saw that we can recover the original nonprojective graphs if we mark the arc labels with the projectivization operations. This, of course, assumes that the arcs in the graphs have labels (functions).

Now we can build a nonprojective system. It uses a projective parser that is preceded by a preprocessing step that projectivizes the training corpus and followed by a postprocessing step that recovers nonprojective arcs from the parsed sentences.

The preprocessing step corresponds to the procedure explained in Sect. 11.10.2 (Kunze 1967; Nivre and Nilsson 2005). It creates a new set of function labels, where the new labels annotate the projectivized arcs. Once this preprocessing step is applied to the training set, we can train the classifiers using a projective parser. When the classifiers are trained, we can run the parser on sentences. It labels the arcs with the original functions as well as with nonprojective markers. We finally apply the postprocessing step to identify these markers and create nonprojective arcs.

13.5 Finding Dependencies Using Constraints

Using constraints is a symbolic strategy that can be an alternative to D-rules. Although it is not as widely used as machine-learning techniques, it can be of interest when no annotated corpus is available. The parsing algorithm is then framed as a constraint satisfaction problem.

Constraint dependency parsing annotates words with dependencies and function tags. It then applies a set of constraints to find a tag sequence consistent with all the constraints. Some methods generate all possible dependencies and then discard inconsistent ones (Harper et al. 1999; Maruyama 1990). Others assign one single dependency per word and modify it (Tapanainen and Järvinen 1997).

Let us exemplify a method inspired by Harper et al. (1999) with the sentence *Bring the meal to the table*. Table 13.7 shows simplified head and function assignments compatible with a word's part of speech.

Table 13.7 Possible functions according to a word's part of speech

Parts of speech	Possible heads	Possible functions
Determiner	Noun	det
Noun	Verb	object, iobject
Noun	Prep	pcomp
Verb	Root	root
Prep	Verb, noun	mod, loc

The first step generates all possible head and function tags. Using Table 13.7, tagging yields:

Words	*Bring*	*the*	*meal*	*to*	*the*	*table*
Position	1	2	3	4	5	6
Part of speech	verb	det	noun	prep	det	noun
Possible tags	<nil, root>	<3, det>	<4, pcomp>	<3, mod>	<3, det>	<4, pcomp>
		<6, det>	<1, object>	<1, loc>	<6, det>	<1, object>
			<1, iobject>			<1, iobject>

Then, a second step applies and propagates the constraint rules. It checks that the constraints do not conflict and enforces the consistency of tag sequences. Rules for English describe, for instance, adjacency (links must not cross), function uniqueness (there is only one subject, one object, one indirect object), and topology:

- A determiner has its head to its right-hand side.
- A subject has its head to its right-hand side when the verb is at the active form.
- An object and an indirect object have their head to their left-hand side (active form).
- A prepositional complement has its head to its left-hand side.

Applying this small set of rules discards some wrong tags but leaves some ambiguity.

Words	*Bring*	*the*	*meal*	*to*	*the*	*table*
Position	1	2	3	4	5	6
Part of speech	verb	det	noun	prep	det	noun
Possible tags	<nil, root>	<3, det>	<1, object>	<3, mod>	<6, det>	<4, pcomp>
			<1, iobject>	<1, loc>		

13.6 Covington's Parser

In the previous sections, we saw that Nivre's parser was restricted to the class of projective graphs. Although projective structures are by far the most frequent in English or French, this restriction impairs the expressivity of the parser, notably in languages where nonprojectivity is not as anecdotal: German, Latin, Russian, etc.

We now introduce Covington's parser (1990, 1994a, 2001), which extends parsing to nonprojective dependencies. Covington's parser is, in fact, a family of algorithms. It starts with a brute-force search that considers all the word pair combinations on which we can gradually set constraints to comply with some generally accepted properties of dependency graphs: head uniqueness and possibly projectivity.

The parser is relatively easy to implement, and it can use D-rules or machine learning techniques. We first introduce a nonprojective version of it and we refine it to produce projective graphs. Like the other parsers we have already seen, the parser input will be a tokenized sentence, where words are tagged with their part of speech.

13.6.1 Covington's Nonprojective Parser

The brute-force version of Covington's algorithm examines each pair of words in the sentence and tries to set a link between them if the grammar or any kind of guide permits it. It can be implemented as a left-to-right pass with the following code:

Algorithm 1 The Covington algorithm

```
1: procedure PARSE(w_1, w_2, ..., w_n)
2:     for i ← 1, n do
3:         for j ← i − 1, 1 do
4:             if PERMISSIBLE(w_i, w_j) then
5:                 LINK(w_i, w_j)
```

that scans the word list and attempts to create links with the words to the left of the current word, w_i. The link operation can either create a left-arc $w_j \leftarrow w_i$, a right-arc $w_j \rightarrow w_i$, or do nothing.

A widely accepted property of dependency graphs is that each word must have a unique head. We will now use lists to represent data and enforce uniqueness. We will store the sentence's words in an input list: `InputList`. The parser accepts one word at a time and maintains two other lists: `HeadList` and `WordList`. `WordList` contains the words already read in a decreasing index order. `HeadList` contains the words that have not been assigned a head yet, also in a decreasing index order. At the beginning of the parse, both `HeadList` and `WordList` are empty.

1. Accept a word `W` from the input list. Add it to `WordList`.
2. Search `HeadList` to find dependents of `W` starting with the most recently added. Words found are removed from `HeadList`.
3. Search the elements of `WordList` to find at most one head for `W`. If no head is found, add `W` to `HeadList`.

At the end of the parse, `HeadList` should contain the head of the sentence.

At each step of the traversal of both `HeadList` and `WordList`, the parser has to decide whether or not it sets a link between the current word and the word in the list. We carry this out with a guide using the term we introduced in Sect. 13.4. Classifiers in the form of decision trees, logistic regression, or support vector machines will predict the existence or absence of a link and the relation holding between two words (if any). To train the classifiers, we need to collect data from a hand-annotated corpus. We extract features using a gold-standard parsing. In the rest of this section, we will exemplify the guide with dependency rules. Extending Covington's parser with machine-learning techniques is left as exercises (Exercises 13.4 and 13.5).

Let us now implement this algorithm in Prolog. We will use the dependency rules below. They have three arguments: the head part of speech, the dependent part of speech, and the grammatical function linking the words. We use a `guide/3` predicate between the rules and the parser to make the linking decision easier to port a classifier.

```
% drule(HeadPOS, DependentPOS, Function)
drule(noun, det, det).
drule(noun, adj, attribute).
drule(verb, noun, subject).
drule(verb, noun, object).
```

Let us now write the parsing algorithm. The input–output format will follow the CoNLL tabular structure. The input will be a sentence in the form of a list of words, where each word will have an index, a form, and a part of speech:

```
w([id=Inx, form=Word, postag=POS])
```

The output will be the dependency graph. We will use the same format as for the input, and we will add a head index and a grammatical function to the words:

```
w([id=Inx, form=Word, postag=POS, head=HdInx,
  deprel=Funct]).
```

The `parse/2` predicate needs an auxiliary `parse/4` to store the lists, `WordList` and `HeadList`. Instead of pushing the current word in `WordList`, we push the whole `w/1` fact as it is being parsed. If we can find a head to it in `WordList`, `w/1` will be fully instantiated. Otherwise, we will leave `HdInx` and `Funct` as variables until we find a word to be a suitable head. `WorldList` will store the parse result. At the end of the process, we attach the words remaining in `HeadList` to the root and we reverse `WordList`.

```
parse(InputL, Result) :-
  parse(InputL, [], [], ReversedResult),
  reverse(ReversedResult, Result).

% parse(+InputL, +HeadList, +WordList, -Result)
% We search dependents of W in HeadList and a head for
```

```
% W in WordList
parse([w([Inx, W, POS]) | InputL], HeadL, WordL,
  Result) :-
  search_headlist(w([Inx, W, POS]), HeadL, NewHeadL,
  WordL),
  search_wordlist(w([Inx, W, POS]), WordL, NewHeadL,
    NextHeadL, HeadInx, Function),
  parse(InputL, NextHeadL, [w([Inx, W, POS,
    head=HeadInx,deprel=Function]) | WordL], Result).

%The remaining headless words are assigned the ROOT
head parse([], HeadL, WordL, WordL) :-
  assign_root(HeadL, WordL).

% assign_root(+HeadList, -WordList)
% assigns roots to the words remaining in WordList
assign_root([], _).
assign_root([w([id=Inx, form=W, postag=POS]) | Rest],
    WordL) :-
  member(
    w([id=Inx, form=W, postag=POS, head=0,
    deprel='ROOT']),WordL),
  assign_root(Rest, WordL).


% search_headlist(+CurWord, +HeadL, -NewHeadL,
  -WordL).
% Searches dependency links in HeadL, and possibly
% assigns them the current word as their head
search_headlist(w([id=Inx1, W, postag=POS]),
  [w([id=Inx2, D, postag=POS_D]) | HeadL],
  NewHeadL, WordL) :-
  % Create left arc
  drule(POS, POS_D, F),
  % We instantiate the relation in WordL
  member(w([id=Inx2, D, postag=POS_D, head=Inx1,
    deprel=F]),WordL),
  search_headlist(w([id=Inx1, W, postag=POS]), HeadL,
    NewHeadL, WordL).
search_headlist(CurWord, [Word | HeadL],
    [Word | NewHeadL],WordL) :-
  % Do nothing
  search_headlist(CurWord, HeadL, NewHeadL, WordL).
search_headlist(_, [], [], _).
```

```
% search_wordlist(+CurWord, +WordL, +HeadL,
%   -NextHeadL, -Head, -Deprel),
% Tries to find a head in WordL
% We look for the first word that can be the head
% of the current word
% Nothing has been done. Shift CurWord to HeadList
search_wordlist(CurWord, [], HeadL, [CurWord | HeadL],
     _, _).
search_wordlist(w([_, _, postag=POS]),
    [w([id=Inx2, _, postag=POS_H, _, _]) | _], HeadL,
    HeadL,Inx2, F) :-
  % Create right arc
  drule(POS_H, POS, F).
% We have not found it and we go on
search_wordlist(CurWord, [_ | WordL], HeadL,
    NextHeadL,H, F) :-
  % Do nothing
  search_wordlist(CurWord, WordL, HeadL, NextHeadL,
    H, F).
```

This algorithm, which is nondeterministic, can successfully parse the sentences:

The waiter ran
The waiter brought a meal

However, the good parse is not the first one delivered by Prolog. Notably, the first parse attaches *a* to *waiter* and assigns the dependency `brought -> meal` the subject function (Fig. 13.4). Prolog must backtrack to find the correct solution.

```
?- parse([w([id=1, form=the, postag=det]),
w([id=2, form=waiter, postag=noun]),
w([id=3, form=brought, postag=verb]),
w([id=4, form=a, postag=det]),
w([id=5, form=meal, postag=noun])], R).

R = [w([id=1, form=the, postag=det, head=2,
    deprel=det]),
w([id=2, form=waiter, postag=noun, head=3,
    deprel=subject]),
w([id=3, form=brought, postag=verb, head=0,
    deprel='ROOT']),
w([id=4, form=a, postag=det, head=2, deprel=det]),
w([id=5, form=meal, postag=noun, head=3,
    deprel=subject])]
```

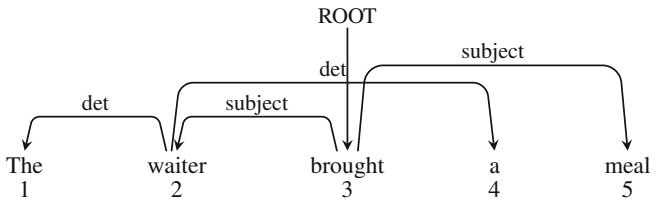


Fig. 13.4 The first parse

Table 13.8 Executing Covington's parser with *The waiter brought a meal*

Word	Procedure	Headlist	Wordlist	Actions
Init.		[]	[]	
w_1	Headlist			
		[]	[]	
	Wordlist			
		$[w_1]$	$[w_1 \leftarrow ?]$	
w_2	Headlist			la
		[]	$[w_1 \leftarrow w_2]$	
	Wordlist			noact
		$[w_2]$	$[w_2 \leftarrow ?, w_1 \leftarrow w_2]$	
w_3	Headlist			la
		[]	$[w_2 \leftarrow w_3, w_1 \leftarrow w_2]$	
	Wordlist			noact noact
		$[w_3]$	$[w_3 \leftarrow ?, w_2 \leftarrow w_3, w_1 \leftarrow w_2]$	
w_4	Headlist			noact
		$[w_3]$	$[w_3 \leftarrow ?, w_2 \leftarrow w_3, w_1 \leftarrow w_2]$	
	Wordlist			noact ra
		$[w_3]$	$[w_4 \leftarrow w_2, w_3 \leftarrow ?, w_2 \leftarrow w_3, w_1 \leftarrow w_2]$	
w_5	Headlist			noact
		$[w_3]$	$[w_4 \leftarrow w_2, w_3 \leftarrow ?, w_2 \leftarrow w_3, w_1 \leftarrow w_2]$	
	Wordlist			noact ra
		$[w_3]$	$[w_4 \leftarrow w_3, w_4 \leftarrow w_2, w_3 \leftarrow ?, w_2 \leftarrow w_3, w_1 \leftarrow w_2]$	

13.6.2 Relations Between Nivre's and Covington's Parsers

We can reframe Covington's parser and describe the link operations in terms of actions. Using this presentation, we will discover its similarities with Nivre's parser and we will be able to apply to it the same machine-learning techniques. When scanning `Headlist`, each time we set a link, it corresponds to a left-arc action of Nivre's parser. When scanning `WordList`, we create at most one link from a word to the current word, which corresponds to a right-arc action of Nivre's parser. To be compatible with the transition-based framework, we need to introduce a no-action that corresponds to list traversal steps, either `Headlist` or `WordList`, where we set no link to or from the current word (Table 13.8).

13.6.3 Covington's Projective Parser

Covington (1990) also provided a projective version of this parser. Recall the definition of a projective graph: if there is a sequence of words $w_i \ldots w_j \ldots w_k$ in which there is a dependency between w_i and w_k, in either direction, then w_j does not have a link preceding w_i or following w_k. To enforce projectivity, we modify the rules:

- When searching `HeadList`, use only the most recent elements and do not skip a word.
- When searching `WordList` for a head: first, skip all the words that are direct or indirect dependents of the current word w. Then, look at the preceding word, its head, its head's head, and so on, following a chain of dependency links.

```
search_headlist_for_dependents(w(W, POS),
    [w(D, POS_D) | HeadList], NewHeadList,
    WordList) :-
  drule(POS, POS_D, F),
  % We instantiate the relation in WordList
  member(w(D, POS_D, W, F), WordList),
  search_headlist_for_dependents(w(W, POS), HeadList,
    NewHeadList, WordList).
search_headlist_for_dependents(_, HL, HL, WordList).

% search_wordlist_for_a_head(+Word, +WordList, -Head,
% -Function),
% Tries to find a head in WordList

% We look for the first word that does not depend
% on the current word
search_wordlist_for_a_head(w(W, POS), WordList, H, F) :-
  next_with_no_link(w(W, POS), WordList, [],
    [w(H, POS_H, _, _) | NewWordList]),
  drule(POS_H, POS, F).

% next_with_no_link(+W, +WordList, -InvWordList,
  -Result)
% We go to the next word that has no link with the
% current word

next_with_no_link(w(W, POS), [w(H, POS_H, HH, F)
  | WordList],InvWordList, Result) :-
  link(w(W, POS), w(H, POS_H, HH, F), InvWordList),
  next_with_no_link(w(W, POS), WordList,
    [w(H, POS_H, HH, F) | InvWordList], Result).
```

```
next_with_no_link(w(W, POS), Result, _, Result).

% link(Word1, Word2, +InvWordList)
% Checks whether there is a link between Word1 and
% Word2 within InvWordList

link(w(W1, POS1), w(W2, POS2, H2, F2),
  InvWordList) :-
  H2 == W1.
link(w(W1, POS1), w(W2, POS2, H2, F2),
     [w(W3, POS3, H3, F3) | InvWordList]) :-
  W3 == H2,
  link(w(W1, POS1), w(W3, POS3, H3, F3),
    InvWordList).
link(w(W1, POS1), w(W2, POS2, H2, F2),
  [_ | InvWordList]) :-
  link(w(W1, POS1), w(W2, POS2, H2, F2),
    InvWordList).
```

Using this new version of the parser, we get better dependencies (Table 13.9):

```
?- parse([the, waiter, brought, a, meal], L).

L = [w(the, determiner, waiter, determinative),
w(waiter, noun, brought, subject),
w(brought, verb, _G861, _G862),
w(a, determiner, meal, determinative),
w(meal, noun, brought, subject)]
```

but *meal* is still the subject. Backtracking assigns the right function:

```
L = [w(the, determiner, waiter, determinative),
w(waiter, noun, brought, subject),
w(brought, verb, _G861, _G862),
w(a, determiner, meal, determinative),
w(meal, noun, brought, object)]
```

We could again improve the parser by enforcing some topology constraints, such as the object is after the verb when at the active form.

13.7 Eisner's Parser

Independently, Sleator and Temperley (1993) and Eisner (1996) developed chart-based dependency parsers that have an $O(n^3)$ complexity. Eisner's parser applies to projective graphs and can be combined with statistical or machine-learning methods. In CoNLL 2007 and 2008, this class of parsers delivered the highest accuracy figures for English.

Table 13.9 Executing Covington's parser with projectivity constraints with *The waiter brought a meal*

Word	Procedure	Headlist	Wordlist	Actions
Init.		[]	[]	
w_1	Headlist			
		[]	[]	
	Wordlist			
		$[w_1]$	$[w_1 \leftarrow ?]$	
w_2	Headlist			la
		[]	$[w_1 \leftarrow w_2]$	
	Wordlist			skip
		$[w_2]$	$[w_2 \leftarrow ?, w_1 \leftarrow w_2]$	
w_3	Headlist			la
		[]	$[w_2 \leftarrow w_3, w_1 \leftarrow w_2]$	
	Wordlist			skip skip
		$[w_3]$	$[w_3 \leftarrow ?, w_2 \leftarrow w_3, w_1 \leftarrow w_2]$	
w_4	Headlist			noact
		$[w_3]$	$[w_3 \leftarrow ?, w_2 \leftarrow w_3, w_1 \leftarrow w_2]$	
	Wordlist			noact
		$[w_4, w_3]$	$[w_4 \leftarrow ?, w_3 \leftarrow ?, w_2 \leftarrow w_3, w_1 \leftarrow w_2]$	
w_5	Headlist			la noact
		$[w_3]$	$[w_4 \leftarrow w_5, w_3 \leftarrow ?, w_2 \leftarrow w_3, w_1 \leftarrow w_2]$	
	Wordlist			skip skip ra
		$[w_3]$	$[w_4 \leftarrow w_3, w_4 \leftarrow w_5, w_3 \leftarrow ?, w_2 \leftarrow w_3, w_1 \leftarrow w_2]$	

Eisner's parser builds on an adaption of the CYK algorithm that it modifies to lower its complexity. We first describe the initial adaptation and then how to alter it to recreate the final parser.

13.7.1 Adapting the CYK Parser to Dependencies

Alshawi (1996) introduced a parser that resembles the CYK parser (Sect. 12.5.2) for dependencies. This parser uses the concept of dotted subtree (Eisner 2000): a sequence of words $w_s..w_t$ corresponding to the range $[s, t]$ and a root w_i inside this range: $s \leq i \leq t$, where all the words in the range are descendants of the root. We denote this subtree: (s, t, i).

A dotted subtree may not be complete; that is, the projection of the head may extend beyond the subtree in the final tree. Figure 13.5 shows an example of a dotted subtree spanning $w_3..w_5$ with *brought* as the head: $(3, 5, 3)$. This dotted subtree is not a real subtree as the projection of *brought* corresponds to the whole sentence.

As with the CYK algorithm, the parser combines dotted subtrees through a bottom up analysis. It initializes the chart with constituents of length 0 consisting of the individual words, and merges pairs of adjacent dotted subtrees (s_1, t_1, i) and

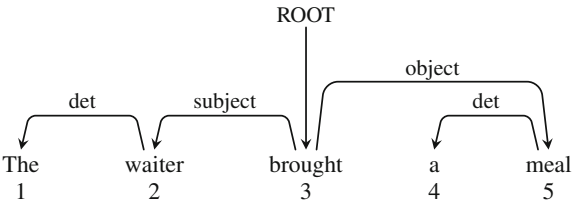


Fig. 13.5 Dependency graph of *The waiter brought a meal*

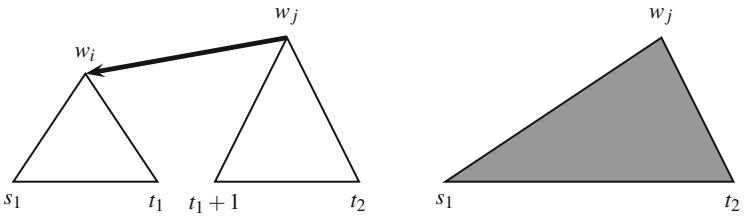


Fig. 13.6 **attach_left** operation: Merge two adjacent dotted subtrees rooted, respectively, at w_i and w_j, where w_j becomes the head of the resulting subtree

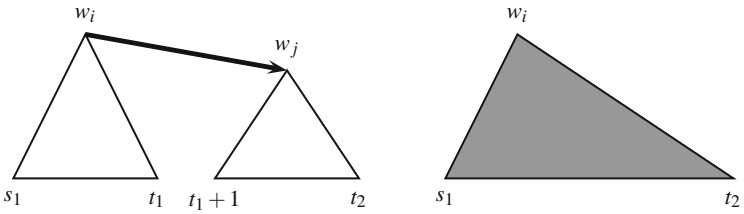


Fig. 13.7 **attach_right** operation: Merge two adjacent dotted subtrees rooted, respectively, at w_i and w_j, where w_i becomes the head of the resulting subtree

$(t_1 + 1, t_2, j)$ into a larger tree using two operations: **attach_left**$((s_1, t_1, i), (t_1 + 1, t_2, j))$ and **attach_right**$((s_1, t_1, i), (t_1+1, t_2, j))$. Their respective definitions are:

Initialization: $(0, 0, 0), (1, 1, 1), (2, 2, 2), \ldots, (i, i, i), \ldots, (n, n, n)$.

attach_left: The head of the right part, w_j, becomes the head of the resulting tree: (s_1, t_2, j) (Fig. 13.6) and

attach_right: This time the head of the left part, w_i, becomes the head of the new tree: (s_1, t_2, i) (Fig. 13.7).

Table 13.10 Chart corresponding to a sentence of length n: The chart is filled with constituents of increasing length, where the cells are filled with dotted subtrees

	Chart					
Length	0	1	2	$\ldots i$	$\ldots n-1$	n
0	$(0,0,0)$	$(1,1,1)$	$(2,2,2)$	$\ldots (i,i,i)$	$\ldots (n-1,n-1,n-1)$	(n,n,n)
1	$(0,1,0)$	$(1,2,1)$	$(2,3,2)$	$\ldots (i,i+1,i)$	$\ldots (n-1,n,n-1)$	–
	$(0,1,1)$	$(1,2,2)$	$(2,3,3)$	$\ldots (i,i+1,i+1)$	$\ldots (n-1,n,n)$	–
2	$(0,2,0)$	$(1,3,1)$	$(2,4,2)$	$\ldots (i,i+2,i)$	$\ldots -$	–
	$(0,2,1)$	$(1,3,2)$	$(2,4,3)$	$\ldots (i,i+2,i+1)$	$\ldots -$	–
	$(0,2,2)$	$(1,3,3)$	$(2,4,4)$	$\ldots (i,i+2,i+2)$	$\ldots -$	–
3	$\ldots$					
$\ldots$	$\ldots$					
$n-1$	$(0,n-1,0)$	$(1,n,1)$	–	$- -$	$- -$	–
	$(0,n-1,1)$	$(1,n,2)$	–	$- -$	$- -$	–
	$(0,n-1,2)$	$(1,n,3)$	–	$- -$	$- -$	–
	$\ldots$					
	$(0,n-1,n-1)$	$(1,n,n)$	–	$- -$	$- -$	–
n	$(0,n,0)$	–	–	$- -$	$- -$	–

Table 13.11 Chart corresponding to the dependency graph shown in Fig. 13.5. For instance, the triple $(0,3,0)$ corresponds to the subtree ranging from the root to word *brought*

Length\Word index	0	1	2	3	4	5
0	$(0,0,0)$	$(1,1,1)$	$(2,2,2)$	$(3,3,3)$	$(4,4,4)$	$(5,5,5)$
1		$(1,2,2)$			$(4,5,5)$	–
2		$(1,3,3)$		$(3,5,3)$	–	–
3	$(0,3,0)$			–	–	–
4			–	–	–	–
5	$(0,5,0)$	–	–	–	–	–
	root	the	waiter	brought	the	meal

After the initialization and similarly to CYK, the parser fills the cells of the chart with constituents (i,j,k) of increasing length: $j-i$ equals to 1, then 2, 3, ..., until the tree covers the range $[0,n]$ with the root at index 0: $(0,n,0)$. Each cell is the combination of two adjacent subtrees (Table 13.10).

Table 13.11 shows an example of the chart with the sentence *The waiter brought the meal*, where the cells are filled with triples leading to the complete tree.

We have presented a conversion of the CYK algorithm that enables us to parse dependencies. Using it, we can associate probabilities to the triples as with the CYK parser for constituents (Sect. 12.5.3), score the subtrees, and have a unique result for each sentence.

This version is far from optimal, however, as its time complexity is in $O(n^5)$: there are $O(n^3)$ triples to fill in the chart, and each triple (i,j,k) needs to examine $O(n^2)$ pairs, (i,l,k) and $(l+1,j,m)$, with varying l and m, to build it. We will see in the next section that we can bring small changes that lower its complexity to $O(n^3)$.

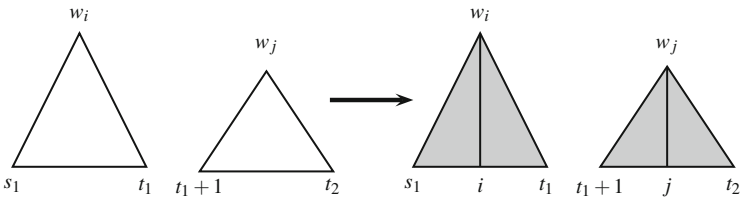


Fig. 13.8 Splitting the two trees into two parts on the head index. This results in four spans whose head is to the left or right of the span (*light gray*)

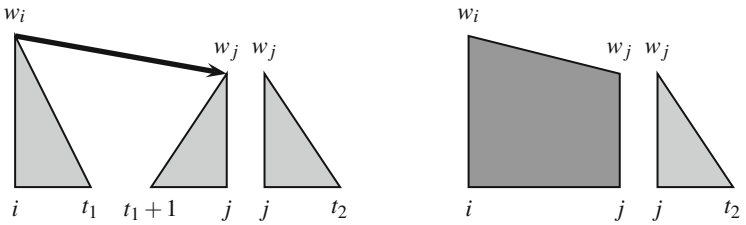


Fig. 13.9 **attach_right** creates an arc from w_i to w_j and a span that contains all the dependents to the right of w_i and to the left of w_j (*gray*)

13.7.2 A More Efficient Version

Eisner's parser is an improvement of the CYK conversion that constrains the position of the headword in the triple (i, j, k) to be either w_i or w_j. It thus eliminates the two variables representing the head positions and reduces the parsing complexity to $O(n^3)$.

To put this idea into practice, we start from the trees in Figs. 13.6 and 13.7 that we split into two parts to the left and right of their headword. Each of these parts is called a span. Figure 13.8 shows a partition that creates four spans from the two trees. To fit the new structure, we need to modify and complement the operations of the previous section. We use four functions:

1. **attach_right** that creates a right-arc between two spans rooted at w_i, respectively w_j, with its dependents to the right, respectively to the left, and results in a new span from i to j. Figure 13.9 shows this operation.
2. **complete_right** that gathers the right dependents of w_j (Fig. 13.10).
3. **attach_left** and **complete_left** that are the mirrors of the two preceding functions.

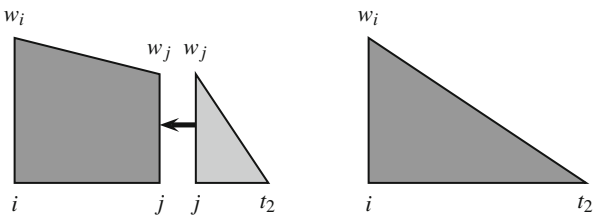


Fig. 13.10 complete_right gathers the dependents to the right of w_j and creates a span from i to t_2 (*gray*)

13.7.3 Implementation

Representing the Spans

Eisner's parser uses a chart, where we fill each cell with spans of increasing length. Their representation is similar to that of the triples from the previous section: each span is bounded by two indices, i and j, and has a head, either i or j. However, the spans need to distinguish whether they have gathered all their dependents to the left, respectively to the right, and thus whether they need to carry out a **complete_left**, respectively **complete_right** operation. We need then one more parameter to mark if a span is available for a complete operation.

Using the notation proposed by McDonald (2006), we model a span by a quadruple (i, j, d, c), where:

- i and j are the indices of the start and the end of the span,
- $d \in \{\rightarrow, \leftarrow\}$ marks the head of the span, either w_i or w_j, and
- $c \in \{0, 1\}$ is a flag that reflects if the span is either incomplete (0) or complete (1), meaning that it has dependents to acquire to the left, in the case of a head to the right, or to the right, in the case of a head to the left.

As examples, Fig. 13.9 shows five spans from left to right before and after an **attach_right** operation. We represent them by the following quadruples:

Before: $(i, t_1, \rightarrow, 1)$, $(t_1 + 1, j, \leftarrow, 1)$, and $(j, t_2, \rightarrow, 1)$,
After: $(i, j, \rightarrow, 0)$ and $(j, t_2, \rightarrow, 1)$.

Parsing Algorithm

As with the CYK parser, Eisner's parser incrementally fills the cells of the chart with spans of increasing length, where each span results from the composition of two subspans. Given a span of length k, this composition can be done in $k - 1$ different ways. To make this analysis possible, we associate a score with each span and span construction. We will then fill the chart with the maximal scoring spans and discard the others.

1: **procedure** EISNERPARSER($w_0, w_1, w_2, \ldots, w_n$)
2: $C(s,s,d,c) \leftarrow 0, \forall s \in \{0,..,n\}, d \in \{\leftarrow, \rightarrow\}, c \in \{0,1\}$
3: **for** $k \leftarrow 1, n$ **do** ▷ We build spans of increasing length
4: **for** $s \leftarrow 1, n$ **do**
5: $t \leftarrow s+k$
6: **if** $t > n$ **then**
7: break
8: $C(s,t,\leftarrow,0) \leftarrow \max_{s \leq r < t} (C(s,r,\rightarrow,1) + C(r+1,t,\leftarrow,1) + s(t,s))$ ▷ Attach left
9: $C(s,t,\rightarrow,0) \leftarrow \max_{s \leq r < t} (C(s,r,\rightarrow,1) + C(r+1,t,\leftarrow,1) + s(s,t))$ ▷ Attach right
10: $C(s,t,\leftarrow,1) \leftarrow \max_{s \leq r < t} (C(s,r,\leftarrow,1) + C(r,t,\leftarrow,0))$ ▷ Complete left
11: $C(s,t,\rightarrow,1) \leftarrow \max_{s < r \leq t} (C(s,r,\rightarrow,0) + C(r,t,\rightarrow,1))$ ▷ Complete right

Fig. 13.11 Eisner parser (After McDonald (2006))

We denote $C(i, j, d, c)$ the score of span (i, j, d, c), and we define an attachment score $s(h, d)$ between two words, w_h and w_d, where w_h is the head, and w_d, the dependent.

We initialize the chart with spans of length 0 and a score of 0, and we apply a sequence of **attach** and **complete** operations to spans of increasing length from 1 to n. We fill the quadruples with the optimal spans so that each span score corresponds to the maximal sum of the scores of two spans it can be created from plus, in the case of an **attach** operation, the attachment score. The score $C(i, j, d, c)$ of span (i, j, d, c) will then be the sum of all the scores of the individual links involved in its construction. Figure 13.11 shows the complete algorithm after McDonald (2006)'s implementation.

Let us denote W, the sequence of words $w_0, w_1, \ldots, w_n$, and G its dependency graph. The score of G is the sum of the individual scores $s(i, j)$, where $(i, j) \in G$, w_i is the head and w_j, the dependent:

$$s(W, G) = \sum_{(i,j) \in G} s(i, j).$$

This final score is given once the chart is filled by the score of the span ranging from the root w_0 to the end of the sentence w_n: $C(0, n, \rightarrow, 1)$. The dependency tree can then be extracted from the chart using a mirror chart of back pointers that store for each span the two spans it originates from.

13.7.4 Learning Graphs with the Perceptron

As we have seen in the previous section, Eisner's parser requires an attachment score $s(h, d)$ between the words w_h and w_d to carry out an **attach** operation. We define such a score as the dot product of a feature vector $\mathbf{f}(w_h, w_d)$ representing the link and a weight vector **weight**:

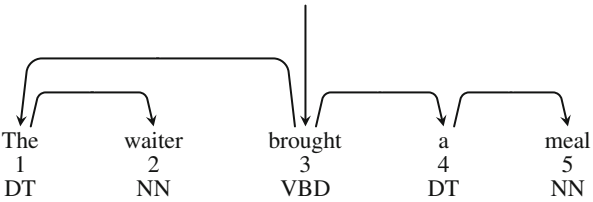


Fig. 13.12 An incorrect dependency analysis of *The waiter brought a meal*

$$s(h,d) = \textbf{weight} \cdot \mathbf{f}(w_h, w_d)$$

and we learn automatically the weight vector from manually parsed corpora using online learning algorithms such as the perceptron that we saw in Sect. 4.7. The final score of the dependency graph G of a sentence W is the sum of all the scores:

$$s(W,G) = \sum_{(h,d)\in G} \textbf{weight} \cdot \mathbf{f}(w_h, w_d)$$

This dot product may seem somewhat abstract. Let us make it concrete with a simple example. We have seen in Sect. 13.4 that D-rules indicate the possibility of a link. Let us use then the part of speech of the head and the part of speech of the dependent as elementary features. To apply the perceptron, we must represent these two parts of speech in the form of a vector of binary digits as introduced in Sect. 4.10. Given that there are about 50 parts of speech in the Penn Treebank, we need to create a vector of 100 dimensions to represent a link. Each vector will have 2 bits that are set to 1; the rest being zeros.

Now let us extract the feature vectors representing the links from graph G in Fig. 13.5, which shows the dependency tree of the sentence *The waiter brought the meal*, and from G' in Fig. 13.12 that shows a wrong analysis of the same sentence. For each link, we build pairs consisting of the parts of speech of the head and the dependent that we convert into binary vector. Table 13.12 shows these vectors with parts of speech limited to the set $\{ROOT, DT, NN, VBD\}$.

In real parsers, such as that of McDonald (2006), there are many more features, including the parts of speech and lexical values of the surrounding words and combinations of them.

The training procedure uses a corpus $\tau = \{(W_t, G_t)\}_{t=1}^{T}$ of T sentences, where W_t is a sentence, and G_t, its associated hand-annotated dependency graph. We initialize the weight vector $\textbf{weight}_{(0)}$ to $\mathbf{0}$ and we apply successive updates using the perceptron. At iteration k, we parse a sentence W_t of the corpus using the weight vector $\textbf{weight}_{(k)}$ to compute the score $s(W_t, G_t)$. The parser returns the graph $\hat{G}_{t(k)}$. The perceptron learns the next weight vector $\textbf{weight}_{(k+1)}$ from differences between G_t and $\hat{G}_{t(k)}$: we compute its update by subtracting $\mathbf{f}(W_t, \hat{G}_{t(k)})$ from $\mathbf{f}(W_t, G_t)$. And we do so until the weight vector converges or we have reached a preset epoch number. Figure 13.13 shows this algorithm modified from McDonald (2006).

Table 13.12 Feature vectors extracted from the dependency graph G shown in Fig. 13.5 and G' in Fig. 13.12. The parts of speech are encoded as Boolean values

Feature vectors		Head POS				Dependent POS			
		ROOT	DT	NN	VBD	ROOT	DT	NN	VBD
		Graph G							
$\mathbf{f}(waiter, The)$	(NN, DT)	0	0	1	0	0	1	0	0
$\mathbf{f}(brought, waiter)$	(VBD, NN)	0	0	0	1	0	0	1	0
$\mathbf{f}(ROOT, brought)$	(ROOT, VBD)	1	0	0	0	0	0	0	1
$\mathbf{f}(meal, the)$	(NN, DT)	0	0	1	0	0	1	0	0
$\mathbf{f}(brought, meal)$	(VBD, NN)	0	0	0	1	0	0	1	0
$\mathbf{f}(W, G) = \sum_{(h,d)\in G} \mathbf{f}(w_h, w_d)$		1	0	2	2	0	2	2	1
		Graph G'							
$\mathbf{f}(brought, The)$	(VBD, NN)	0	0	0	1	0	1	0	0
$\mathbf{f}(The, waiter)$	(DT, NN)	0	1	0	0	0	0	1	0
$\mathbf{f}(ROOT, brought)$	(ROOT, VBD)	1	0	0	0	0	0	0	1
$\mathbf{f}(brought, the)$	(VBD, DT)	0	0	0	1	0	1	0	0
$\mathbf{f}(the, meal)$	(DT, NN)	0	1	0	0	0	0	1	0
$\mathbf{f}(W, G') = \sum_{(h,d)\in G'} \mathbf{f}(w_h, w_d)$		1	2	0	2	0	2	2	1

1: **function** PERCEPTRON(τ)
2: $\quad$ **weight**$_{(0)} \leftarrow \mathbf{0}$
3: $\quad k \leftarrow 0$
4: $\quad$ **for** $n \leftarrow 1, N$ **do** ▷ We select the epoch number
5: $\quad\quad$ **for** $t \leftarrow 1, T$ **do** ▷ We iterate over the corpus
6: $\quad\quad\quad \hat{G}_{t(k)} \leftarrow$ EISNERPARSER(W_t, **weight**$_{(k)}$) ▷ We parse W_t with **weight**$_{(k)}$
7: $\quad\quad\quad$ **if** $\hat{G}_{t(k)} \neq G_t$ **then**
8: $\quad\quad\quad\quad$ **weight**$_{(k+1)} \leftarrow$ **weight**$_{(k)} + \mathbf{f}(W_t, G_t) - \mathbf{f}(W_t, \hat{G}_{t(k)})$ ▷ Perceptron update
9: $\quad\quad\quad\quad k \leftarrow k+1$
10: $\quad$ **return weight**$_{(k)}$

Fig. 13.13 Learning the weight vector **weight** using the perceptron (Modified from McDonald (2006))

13.8 Further Reading

While most research in English has been done using the constituency formalism – and many computational linguists still use it – dependency inspires much of the present work. Early implementations of dependency theories include Link Grammar (Sleator and Temperley 1993) and the Functional Dependency Grammar (Järvinen and Tapanainen 1997) that uses constraint rules and produces a dependency structure where links are annotated with functions. Covington (1990) described an algorithm that could parse discontinuous constituents. Constant (1991), El Guedj (1996), and Vergne (1998) provide accounts in French; Hellwig (1980, 1986) was among the pioneers in German.

Some authors reformulated parsing a constraint satisfaction problem (CSP), sometimes combining it with a chart. Constraint handling rules (CHR) is a simple, yet powerful language to define constraints (Frühwirth 1998). Constraint handling rules are available in some Prologs, notably SWI Prolog.

In 2006 and 2007, the Conference on Computational Natural Language Learning (CoNLL) organized its shared task on multilingual dependency parsing (Buchholz and Marsi 2006; Nivre et al. 2007). The conference site provides background literature, data sets, and an evaluation scheme (http://www.cnts.ua.ac.be/conll/). As a result, two main classes of parsing methods emerged from these shared tasks: the first one being transition-based, as is Nivre's parser, and the second one based on Eisner's parser. They still dominate the world of dependency parsing today. See McDonald (2006) and Kübler et al. (2009) for details as well as well as for a third technique based on maximum spanning trees.

The performance of a parser depends in a large measure on the feature set it uses. In this chapter, we reviewed relatively simple sets. Table 11.12 shows a baseline set for transition-based parsing that needs to be experimentally tuned for each language to analyze. While feature sets can be created manually, Nilsson and Nugues (2010) describe an automatic procedure to discover features for transition-based parsers. In Eisner's parser, we considered first-order features involving a single arc between a head and a dependent. It is possible to extend the set to second-order features representing two links, between a head and two adjacent dependents or between a head, a dependent, and a dependent of the dependent, as well as higher-order features. Eisner's parser needs then to be extended to accommodate these features. See Carreras (2007) for a description. Readers interested in building a high-performance parser should refer to the original papers, from CoNLL for instance, that describe the complete feature sets.

Finally, we trained Eisner's parser using the perceptron. This online learning technique can also be applied to transition-based parsing combined with beam search; the features are then extracted from the parser states and transitions (Johansson and Nugues 2007b). Although, this combination initially yielded results inferior to those obtained with local classifiers, they are now on a par with the best published performances (Zhang and Nivre 2011).

Exercises

13.1. Improve Nivre's parser with D-rules. Use the suggestions proposed in Sect. 13.4.1.

13.2. Add features to Nivre's parser and evaluate their contribution. Download an annotated corpus from the CoNLL 2006 web site and use the evaluation script to measure the attachment score. You can use features proposed in Table 13.6.

13.3. Implement a function classifier to Nivre's parser. Download an annotated corpus from the CoNLL 2006 web site and use the evaluation script to measure the labelled attachment score. You can use features proposed in Table 13.6.

13.4. Rewrite Covington's parser to parse an annotated corpus using either the projective or nonprojective version. Download a corpus from the CoNLL 2006 web site, for example, and extract features. You can use features proposed in Table 13.6.

13.5. Extend Covington's parser (projective or nonprojective) with classifiers. Train classifiers from the features you extracted in Exercise 13.4. You can use either decision trees or support vector machines. Apply your parser on a corpus from the CoNLL 2006 web site and use the evaluation script to measure the labeled attachment score.

13.6. Implement Eisner's parser.

Chapter 14
Semantics and Predicate Logic

> Insbesondere glaube ich, dass die Ersetzung der Begriffe *Subject* und *Praedicat* durch *Argument* und *Function* sich auf die Dauer bewähren wird.
> "In particular, I believe that the replacement of the concepts subject and predicate by argument and function, respectively, will stand the test of time."
>
> Gottlob Frege, Preface to *Begriffsschrift*, 1879.

14.1 Introduction

Semantics deals with the meaning of words, phrases, and sentences. It is a wide and open subject intricately interwoven with the structure of the mind. The potential domain of semantics is immense and covers many of the human cognitive activities. It has naturally spurred a great number of theories. From the philosophers of ancient and medieval times, to logicians of the nineteenth century, psychologists and linguists of the twentieth century, and now computer scientists, a huge effort has been made on this subject.

Semantics is quite subtle to handle or even to define comprehensively. It would be a reckless challenge to claim to introduce an exhaustive view of the topic. It would be even more difficult to build a unified point of view of all the concepts that are attached to it. In this chapter, we will cover formal semantics. This approach to semantics is based on logic and is the brainchild of both linguists and mathematicians. It addresses the representation of phrases and sentences, the definition of truth, the determination of reference (linking words to the world's entities), and some reasoning. In the next chapter, we will review lexical semantics.

P.M. Nugues, *Language Processing with Perl and Prolog*, Cognitive Technologies,
DOI 10.1007/978-3-642-41464-0_14, © Springer-Verlag Berlin Heidelberg 2014

Table 14.1 A dialogue between diners and the robot

Dialogue turns	Sentences
Socrates orders the dinner from the robot	*Bring the meal to the table*
The robot, after it has brought the meal, warns the diners	*The meal is on the table. It is hot*
Pierre, who was not listening	*Is this meal cold?*
...	*Miam miam*
Socrates, after the dinner is finished	*Clear the table*

14.2 Language Meaning and Logic: An Illustrative Example

Roughly defined, formal semantics techniques attempt to map sentences onto logical formulas. They cover areas of sentence representation, reference, and reasoning. Let us take an example to outline and illustrate layers involved in such a semantic processing. Let us suppose that we want to build a robot to serve us a dinner. To be really handy, we want to address and control our beast using natural language. So far, we need to implement a linguistic interface that will understand and process our orders and a mechanical device that will carry out the actions in the real world. Given the limits of this book, we set aside the mechanical topics and we concentrate on the linguistic part.

To avoid a complex description, we confine the scope of the robot's understanding to a couple of orders and questions. The robot will be able to bring meals to the table, to answer a few questions from the patrons, and to clear the table once the meals have been eaten. Now, let us imagine a quick dialogue between the two diners, Socrates and Pierre, and the robot (Table 14.1).

Processing the sentences' meaning from a logical viewpoint requires a set of steps that we can organize in operating modules making parts of a semantic interpretation system. The final organization of the modules may vary, depending on the final application.

- The first part has to **represent** the state of the world. There is a table, diners around the table, a meal somewhere, and a robot. A condition to any further processing is to have them all in a knowledge base. We represent the entities, persons and things, using symbols that we store in a Prolog database. The database should reflect at any moment the current state of the world and the properties of the entities. That is, any change in the world should update the Prolog database correspondingly. When the robot mechanically modifies the world or when it asserts new properties on objects, a corresponding event has to appear in the database.
- The second part has to **translate** phrases or sentences such as *The robot brought the meal* or *the meal on the table* into formulas a computer can process. This also involves a representation. Let us consider the phrase *the meal on the table.* There are two objects, x and y, with x being a meal and y being a table. In addition,

both objects are linked together by the relation that x is on y. A semantic module based on formal logic will translate such a phrase into a **logical form** compatible with the representation of objects into the database. This module also has to assert it into the database.

- A third part has to **reference** the logical forms to real objects represented in the database. Let us suppose that the robot asserts: *The meal is on the table. It is hot.* Referencing a word consists of associating it to an object from the real world (more accurately, to its corresponding symbol in the database). Referencing is sometimes ambiguous. There might be two meals: one being served and another one in the refrigerator. The referencing module must associate the word *meal* to the right object. In addition, referencing has also to keep track of entities mentioned in the discourse and to relate them. *It* in the second sentence refers to the same object as *the meal on the table* in the first sentence, and not to another meal in the refrigerator.
- A fourth part has to **reason** about the world and the sentences. Consider the utterance *The meal is on the table. Is it cold?* Is the latter assertion true? Is it false? To answer this question, the semantic interpreter must determine whether there is really a meal on the table and whether it is cold. To check it, the interpreter needs either to look up whether this fact is in the database or to have external devices such as a temperature sensor and a definition of cold. In addition, if a fact describes the meal as hot, a reasoning process must be able to tell us that if something is hot, it is not cold. We can implement such reasoning in Prolog using rules and an inference mechanism.

14.3 Formal Semantics

Of the many branches of semantics, formal semantics is one of the best-established in the linguistic community. The main assumption behind it is that logic can model language and, by extension, human thought. This has many practical consequences because, at hand, there is an impressive set of mathematical models and tools to exploit. The most numerous ones resort to the first-order predicate calculus. Such tools were built and refined throughout the twentieth century by logicians such as Frege, Herbrand, Russell, and Tarski.

The formal semantics approach is also based on assumptions linking a sentence to its semantic representation and, most notably, the principle of compositionality. This principle assumes that a sentence's meaning depends on the meaning of the phrases that compose it: "the meaning of the whole is a function of the meaning of its parts." A complementary – and maybe more disputable – assumption is that the phrases carrying meaning can be mapped onto syntactic units: the constituents. As a result, the principle of compositionality ties syntax and semantics together. Though

there are many utterances in English, French, or German that are not compositional, these techniques have proved of interest in some applications.

14.4 First-Order Predicate Calculus to Represent the State of Affairs

The first concrete step of semantics is to represent the **state of affairs**: objects, animals, people, and observable facts together with properties of things and relations between them. A common way to do this is to use **predicate–argument structures**. The role of a semantic module will then be to map words, phrases, and sentences onto symbols and structures characterizing things or properties in a given context: the **universe of discourse**.

First-order predicate calculus (FOPC) is a convenient tool to represent things and relations. FOPC has been created by logicians and is a proven tool to express and handle knowledge. It features constants, variables, and terms that correspond exactly to predicate–argument structures. We examine here these properties with Prolog, which is based on FOPC.

14.4.1 Variables and Constants

We can map things, either real or abstract, onto constants – or atoms – and subsequently identify the symbols to the things. Let us imagine a world consisting of a table and two chairs with two persons in it. This could be represented by five constants stored in a Prolog database. Then, the state of affairs is restrained to the database:

```
% The people:
  'Socrates'.
  'Pierre'.

% The chairs:
  chair1.    % chair #1
  chair2.    % chair #2

% The unique table:
  table1.    % table #1
```

A second kind of device, Prolog's variables such as `X`, `Y`, or `Z`, can unify with any entity of the universe and hold its value. And variable `X` can stand for any of the five constants.

14.4.2 Predicates

Predicates to Encode Properties

Predicates are symbols representing properties or relations. Predicates indicate, for instance, that `'Pierre'` has the property of being a person and that other things have the property of being objects. We state this simply using the `person` and `object` symbols as functors (predicate names) and `'Pierre'` and `table1` as their respective arguments. We add these facts to the Prolog database to reflect the state of the world:

```
person('Pierre').
person('Socrates').

object(table1).
object(chair1).
object(chair2).
```

We can be more specific and use other predicates describing that `table1` is a table, and that `chair1` and `chair2` are chairs. We assert this using the `table/1` and `chair/1` predicates:

```
table(table1).

chair(chair1).
chair(chair2).
```

Predicates to Encode Relations

Predicates can also describe relations between objects. Let us imagine that chair `chair1` is in front of table `table1`, and that `'Pierre'` is on `table1`. We can assert these relative positions using functors, such as `in_front_of/2` or `on/2`, linking respectively arguments `chair1` and `table1`, and `'Pierre'` and `table1`:

```
in_front_of(chair1, table1).

on('Pierre', table1).
```

So far, we have only used constants (atoms) as arguments in the properties and in the predicates representing them. If we want to describe more accurately three-dimensional scenes such as that in Fig. 14.1, we need more elaborate structures.

In such a scene, a coordinate system is necessary to locate precisely entities of the world. Since we are in a 3D space, 3D vectors give the position of objects that we can represent using the `v/3` predicate. `v(?x, ?y, ?z)` indicates the coordinate values of a point on x, y, and z axes. To locate objects, we will make use of `v/3`.

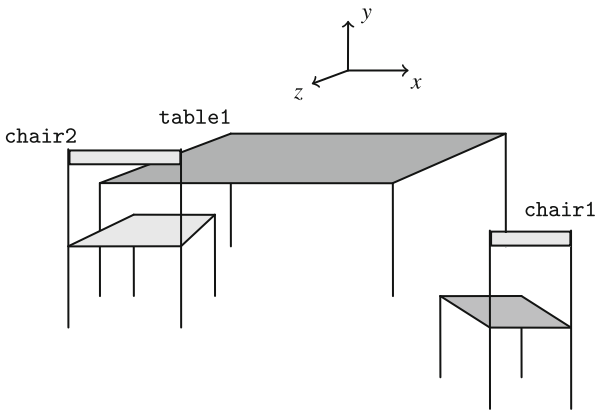


Fig. 14.1 A three-dimensional scene

For the sake of simplicity here, we approximate an object's position to its gravity center. We locate it with the `position/2` predicate. Position facts are compound terms that take the name of an object and the vector reflecting its gravity center as arguments:

```
position(table1, v(0, 0, 0)).
position(chair1, v(1, 1, 0)).
position(chair2, v(10, -10, 0)).
```

14.5 Querying the Universe of Discourse

Now, we have a database containing facts, i.e., properties and relations unconditionally true that describe the state of affairs. Using queries, the Prolog interpreter can check whether a fact is true or false:

```
?- table(chair1).
false.

?- chair(chair2).
true.
```

In addition, unification enables Prolog to determine subsets covering certain properties:

```
?- chair(X).
X = chair1;
X = chair2
```

We can get the whole subset in one shot using `bagof/3`. The alternate query yields:

```
?- bagof(X, chair(X), L).
L = [chair1, chair2]
```

The built-in `bagof/3` predicate has a cousin: `setof/3`. The difference is that `setof/3` sorts the elements of the answer and removes possible duplicates.

We may want to intersect properties and determine the set of the corresponding matching objects. Prolog can easily do this using conjunctions and shared variables. For instance, we may want to select from the set of chairs those that have the property of being in front of a table. The corresponding query is:

```
?- chair(X), in_front_of(X, Y), table(Y).
X = chair1, Y = table1
```

14.6 Mapping Phrases onto Logical Formulas

Using predicate–argument structures, we can map words, phrases, and sentences onto logical formulas. Simplifying a bit, nouns, adjectives, or verbs describe properties and relations that we can associate to predicates. Having said this, we have solved one part of the problem. We need also to determine the arguments that we will represent as logical variables.

Arguments refer to real-world entities, and the state of affairs should define their value. We then need a second process to have a complete representation that will replace – unify – each variable with a logical constant. We will first concentrate on the representation of words or phrases and leave the arguments uninstantiated for now.

As a notation, we use λ-expressions that provide an abstraction of properties or relations. The λ symbol denotes variables that we can substitute with an entity of the real world, such as:

$$\lambda x.property(x)$$

or

$$\lambda y.\lambda x.property(x, y)$$

where λx indicates that we may supply an expression or a value for x.

Supplying such a value is called a β-reduction. It replaces all the occurrences of x in the expression and eliminates λx:

$$(\lambda x.property(x))entity\#1$$

yields

$$property(entity\#1)$$

Table 14.2 Representation of nouns and adjectives

Lexical representations	Sentences	Semantic representations
Nouns		
`X^chair(X)`	*chair1 is a chair*	`chair(chair1)`
`X^patron(X)`	*Socrates is a patron*	`patron('Socrates')`
Adjectives		
`X^yellow(X)`	*table1 is yellow*	`yellow(table1)`
`X^hot(X)`	*meal2 is hot*	`hot(meal2)`

λ is a right-associative operator that we cannot get with Western keyboards. We use the symbol `^` to denote it in Prolog. And `X^property(X)` is equivalent to $\lambda x.property(x)$.

14.6.1 Representing Nouns and Adjectives

Nouns or adjectives such as *waiter*, *patron*, *yellow*, or *hot* are properties that we map onto predicates of arity 1. For example, we represent the noun *chair* by:

$$\lambda x.chair(x)$$

whose equivalent notation in Prolog is `X^chair(X)`. Let us suppose that *chair1* is an entity in the state of affairs. We can supply it to this λ-expression:

$$(\lambda x.chair(x))chair1$$

and carry out a β-reduction that yields:

$$chair(chair1).$$

Table 14.2 shows some examples of representation of nouns and adjectives.

We can consider proper nouns as well as common nouns. In this case, we will have predicates such as `X^pierre(X)` and `X^socrates(X)`. This means that there are several Pierres and Socrates that can be unified with variable `X`. We can also make a nice distinction between them and treat proper nouns as constants like we have done before. In this case, there would be one single Pierre and one single Socrates in the world. Such a choice depends on the application.

14.6.2 Representing Noun Groups

Noun groups may consist of a sequence of adjectives and a head noun. We form their semantic representation by combining each representation in a conjunction of properties (Table 14.3).

Table 14.3 Noun groups

Noun groups	Semantic representation
hot meal	`X^(hot(X), meal(X))`
fast server	`X^(fast(X), server(X))`
yellow big table	`X^(yellow(X), big(X), table(X))`

Table 14.4 Representation of verbs

Lexical representations	Sentences	Sentence representations
Intransitive verbs		
`X^ran(X)`	*Pierre ran*	`ran('Pierre')`
`X^sleeping(X)`	*Socrates is sleeping*	`sleeping('Socrates')`
Transitive verbs		
`Y^X^served(X,Y)`	*Roby served a meal*	`served('Roby', Z^meal(Z))`
`Y^X^brought(X, Y)`	*Roby brought a plate*	`brought('Roby',Z^plate(Z))`

Table 14.5 Preposition representation

Lexical representations	Phrases	Phrase representations
`Y^X^in(X, Y)`	*The fish in the plate*	`in(Z^fish(Z), T^plate(T))`
`Y^X^from(X,Y)`	*Pierre from Normandy*	`from('Pierre', 'Normandy')`
`Y^X^with(X,Y)`	*The table with a napkin*	`with(Z^table(Z), T^napkin(T))`

The case is trickier when we have compounded nouns such as:

computer room
city restaurant
night flight

Noun compounds are notoriously ambiguous and require an additional interpretation. Some compounds should be considered as unique lexical entities such a *computer room*. Others can be rephrased with prepositions. *A city restaurant* is similar to *a restaurant in the city*. Others can be transformed using an adjective. A *night flight* could have the same interpretation as a *late flight*.

14.6.3 Representing Verbs and Prepositions

Verbs such as *run*, *bring*, or *serve* are relations. We map them onto predicates of arity 1 or 2, depending on whether they are intransitive or transitive, respectively (Table 14.4).

Prepositions usually link two noun groups, and like transitive verbs, we map them onto predicates of arity 2 (Table 14.5).

14.7 The Case of Determiners

14.7.1 Determiners and Logic Quantifiers

So far, we have dealt with adjectives, nouns, verbs, and prepositions, but we have not taken determiners into account. Yet, they are critical in certain sentences. Compare:

1. *A waiter ran*
2. *Every waiter ran*
3. *The waiter ran*

These three sentences have a completely different meaning, although they differ only by their determiners. The first sentence states that there is a waiter and that s/he ran. We can rephrase it as there is an x that has a conjunction of properties: $waiter(x)$ and $ran(x)$. The second sentence asserts that all x having the property $waiter(x)$ also have the property $ran(x)$.

Predicate logic uses two quantifiers to transcribe these statements into formulas:

- The existential quantifier, denoted $\exists$, and read *there exists*, and
- The universal quantifier, denoted $\forall$, and read *for all*

that we roughly associate to determiners *a* and *every*, respectively.

The definite determiner *the* refers to an object supposedly unique over the whole universe of discourse. We can connect it to the restricted existential quantifier denoted $\exists!$ and read *there exists exactly one*. *The waiter ran* should then be related to a unique waiter.

We can also use the definite article to designate a specific waiter even if there are two or more in the restaurant. Strictly speaking, *the* is ambiguous in this case because it matches several waiters. *The* refers then to an object unique in the mind of the speaker as s/he mentions it, for instance, the waiter s/he can see at the very moment s/he is saying it or the waiter taking care of her/his table. The universe of discourse is then restricted to some pragmatic conditions. We should be aware that these conditions may bring ambiguity in the mind of the hearer – and maybe in that of the speaker.

14.7.2 Translating Sentences Using Quantifiers

Let us now consider determiners when translating sentences, and let us introduce quantifiers. For that, we associate determiner *a* with quantifier $\exists$ and *every* with $\forall$. Then, we make the quantifier the head of a logical formula that consists either of a conjunction of predicates for determiner *a* or of an implication with *every*. The arguments are different depending on whether the verb is transitive or intransitive.

Table 14.6 Representation of sentences with intransitive verbs using determiners

Sentences	Logical representations
A waiter ran	$\exists x(waiter(x) \wedge ran(x))$
	`exists(X, waiter(X), ran(X))`
Every waiter ran	$\forall x(waiter(x) \Rightarrow ran(x))$
	`all(X, waiter(X), ran(X))`
The waiter ran	$\exists! x(waiter(x) \wedge ran(x))$
	`the(X, waiter(X), ran(X))`

With intransitive verbs, the logical conjunctions or implications link the subject to the verb. Table 14.6 shows a summary of this with an alternate notation using Prolog terms. Predicates – principal functors – are then the quantifiers' names: `all/3`, `exists/3`, and `the/3`.

When sentences contain a transitive verb like:

A waiter brought a meal
Every waiter brought a meal
The waiter brought a meal

we must take the object into account. In the previous paragraph, we have represented subject noun phrases with a quantified logical statement. Processing the object is similar. In our examples, we map the object *a meal* onto the formula:

$$\exists y(meal(y))$$

Then, we link the object's variable y to the subject's variable x using the main verb as a relation predicate:

$$brought(x, y)$$

Finally, sentence *A waiter brought a meal* is represented by:

$$\exists x(waiter(x) \wedge \exists y(meal(y) \wedge brought(x, y)))$$

Table 14.7 recapitulates the representation of the examples.

14.7.3 A General Representation of Sentences

The quantifiers we have used so far are the classical ones of logic. Yet, in addition to *a*, *every*, and *the*, there are other determiners such as numbers: *two*, *three*, *four*; indefinite adjectives: *several*, *many*, *few*; possessive pronouns: *my*, *your*; demonstratives: *this*, *that*; etc. These determiners have no exact counterpart in the world of standard first-order logic quantifiers.

Table 14.7 Logical representation of sentences with transitive verbs using determiners

Sentences	Logical representation
A waiter brought a meal	$\exists x(waiter(x) \wedge \exists y(meal(y) \wedge brought(x, y)))$
	`exists(X, waiter(X),` `  exists(Y, meal(Y), brought(X, Y))`
Every waiter brought a meal	$\forall x(waiter(x) \Rightarrow \exists y(meal(y) \wedge brought(x, y)))$
	`all(X, waiter(X),` `  exists(Y, meal(Y), brought(X, Y))`
The waiter brought a meal	$\exists! x(waiter(x) \wedge \exists y(meal(y) \wedge brought(x, y)))$
	`the(X, waiter(X),` `  exists(Y, meal(Y), brought(X, Y))`

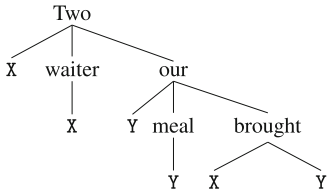


Fig. 14.2 Semantic representation of *Two waiters brought our meals*

A more general representation uses determiners themselves as functors of Prolog terms instead of logic quantifier names. The subject noun phrase's determiner will be the principal functor of term mapping the whole sentence. Subsequent determiners will be the functors of inner terms. For example,

Two waiters brought our meals

is translated into

```
two(X, waiter(X), our(Y, meal(Y), brought(X, Y)))
```

Figure 14.2 depicts this term graphically.

Such a formalism can be extended to other types of sentences that involve more complex combinations of phrases (Colmerauer 1982). The basic idea remains the same: we map sentences and phrases onto trees – Prolog terms – whose functor names are phrases' determiners and whose arity is 3. Such terms are also called ternary trees. The top node of the tree corresponds to the sentence's first determiner (Fig. 14.3). The three arguments are:

- A variable that the determiner introduces into the semantic representation, say `X`
- The representation of the first noun phrase bound to the latter variable, that is `X` here
- The representation of the rest of the sentence, which we give the same recursive structure

As a result, a sentence is transformed into the Prolog predicate (Fig. 14.3):

```
determiner(X, SemNP, SemRest).
```

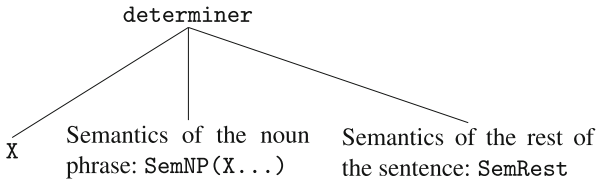


Fig. 14.3 Semantic representation using ternary trees

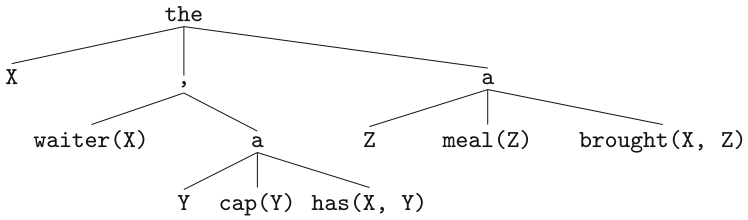


Fig. 14.4 Representation of *The waiter who has a cap brought a meal*

This representation also enables us to process relative clauses and adjuncts. We represent them as a conjunction of properties. For example,

The waiter who has a cap

is translated into

```
the(X, (waiter(X), a(Y, cap(Y), has(X, Y))), P)
```

where the second argument corresponds to the relative clause, the comma (`,`) between `waiter(X)` and `a(Y, cap(Y), has(X, Y))` stands for a conjunction of these properties, and where `P` is linked with a possible rest of the sentence. If we complement this phrase with a verb phrase:

The waiter who has a cap brought a meal

we can give a value to `P` and the complete sentence representation will be (Fig. 14.4):

```
the(X,
    (waiter(X), a(Y, cap(Y), has(X, Y))),
    a(Z, meal(Z), brought(X, Z))).
```

14.8 Compositionality to Translate Phrases to Logical Forms

In Chap. 9, we used λ-calculus and compositionality to build a logical form out of a sentence. We will resort to these techniques again to incorporate the representation of determiners. Just like the case for nouns and verbs, we will process determiners using arguments in the DCG rules that will carry their partial semantic

representation. The construction of the logical form will proceed incrementally using Prolog's unification while parsing the phrases and the sentence. The semantic composition of a sentence involves:

1. The translation of the first noun phrase – the subject
2. The translation of the verb phrase – the predicate – that contains a possible second noun phrase – the object

From the representation we provided in Chap. 9, the main change lies in the noun phrase translation. We approximated its semantics to the noun itself. Now we refine it into:

```
determiner(X, SemNP, SemRest).
```

14.8.1 Translating the Noun Phrase

To obtain `SemNP`, we have to compose the semantics of the determiner and of the noun, knowing that the noun's representation is:

```
noun(X^waiter(X)) --> [waiter].
```

Since the determiner must form the top node of the semantic tree, it has to embed an incomplete representation of the whole phrase. If we go back to the principles of λ-calculus, we know that the λ-variable indicated roughly that we request a missing value. In this case, the determiner needs the noun representation to reduce it. In consequence, variables in the noun phrase rule must be:

```
np(Sem) --> determiner((X^SemNP)^Sem), noun(X^SemNP).
```

We need to specify a variable `X` in the λ-expression of this rule, because it unifies with the `Sem` term, that is, with its first argument, as well as with `SemNP` and `SemRest`.

To write the determiner's lexical rule, we have now to proceed down into the structure details of `Sem`. The term `Sem` reflects a logical form of arity 3. It obtained its second argument `SemNP` from the subject – it did this in the `np` rule. It has to get its third argument, `SemRest`, from the verb and the object. `SemRest` will be built by the verb phrase `vp`, and since it is not complete at the moment, we denote it with a λ-expression. So, variables in the determiner rules are:

```
determiner((X^SemNP)^(X^SemRest)^a(X, SemNP, SemRest)) -->
  [a].
```

Again, we must specify the `X` variable that is to be bound in `SemRest`. Using these rules, let us process *a waiter*. They yield the logical form:

```
(X^SemRest)^a(X, waiter(X), SemRest)
```

whose λ-variable `(X^SemRest)^` requests the semantic value of the verb phrase. The sentence rule `s` provides it and builds the complete representation, where `vp` brings `SemRest`:

```
s(Sem) --> np((X^SemRest)^Sem), vp(X^SemRest).
```

14.8.2 Translating the Verb Phrase

Now, let the verb phrase rules compose the semantics of the rest (`SemRest`). The representation of the verbs remains unchanged. The verbs feature a single variable when intransitive, as in:

```
verb(X^rushed(X)) --> [rushed].
```

and two variables when transitive:

```
verb(Y^X^ordered(X, Y)) --> [ordered].
```

Verb phrase semantics is simple with an intransitive verb:

```
vp(X^SemRest) --> verb(X^SemRest).
```

It is a slightly more complicated when there is an object. As for the subject, the object's determiner embeds a ternary tree as a representation (Fig. 14.2). It introduces a new variable `Y` and contains a λ-expression that requests the representation of the verb. This λ-expression surfaces at the verb phrase level to bind the verb semantics to the third argument in the ternary tree. Let us name it `(Y^SemVerb)^`. It enables us to write the `vp` rule:

```
vp(X^SemRest) -->
  verb(Y^X^SemVerb),
  np((Y^SemVerb)^SemRest).
```

Finally, the whole program consists of these rules put together:

```
s(Sem) --> np((X^SemRest)^Sem), vp(X^SemRest).

np((X^SemRest)^Sem) -->
  determiner((X^SemNP)^(X^SemRest)^Sem),
  noun(X^SemNP).

vp(X^SemRest) --> verb(X^SemRest).
vp(X^SemRest) -->
  verb(Y^X^SemVerb),
  np((Y^SemVerb)^SemRest).
```

Let us also write a couple of vocabulary rules:

```
noun(X^waiter(X)) --> [waiter].
noun(X^patron(X)) --> [patron].
noun(X^meal(X)) --> [meal].

verb(X^rushed(X)) --> [rushed].
verb(Y^X^ordered(X, Y)) --> [ordered].
verb(Y^X^brought(X, Y)) --> [brought].

determiner((X^SemNP)^(X^SemRest)^a(X, SemNP,
SemRest)) -->  [a].
determiner((X^SemNP)^(X^SemRest)^the(X, SemNP,
SemRest)) -->  [the].
```

These rules applied to the sentence *The patron ordered a meal* yield:

```
?- s(Sem, [the, patron, ordered, a, meal], []).

Sem =
  the(_4,patron(_4),a(_32,meal(_32),ordered(_4,_32)))
```

where `_4` and `_32` are Prolog internal variables. Let us rename them `X` and `Y` to provide an easier and equivalent reading:

```
Sem = the(X, patron(X), a(Y, meal(Y), ordered(X, Y)))
```

Similarly, *the waiter rushed* produces

```
Sem = the(X, waiter(X), rushed(X))
```

14.9 Augmenting the Database and Answering Questions

Now that we have built a semantic representation of a sentence, what do we do with it? This has two answers, depending on whether it is a declaration or a question. We must keep in mind that the state of affairs – here the Prolog database – reflects the total knowledge available to the interpretation system. If it is a declaration – a statement from the user – we must add something because it corresponds to new information. Conversely, if the user asks a question, we must query the database to find a response. In this section, we will review some straightforward techniques to implement it.

14.9.1 Declarations

When the user utters a declaration, the system must add its semantic representation to the description of the state of affairs. With a Prolog interpreter, the resulting

semantic fact – corresponding, for example, to `determiner(X, NP, Rest)` – will have to be asserted to the database.

We can carry this out using one of the `asserta` or `assertz` predicates. The system builds the semantic representation while parsing and asserts the new fact when it has finished, that is, after the `sentence` rule. Since `asserta` is a Prolog predicate and we are using DCG rules, we enclose it within curly brackets (braces). The rule

```
sentence(Sem) -->
  np(...), vp(...), {asserta(Sem), ...}.
```

will result into a new `Sem` predicate asserted in the database once the sentence has been parsed.

14.9.2 Questions with Existential and Universal Quantifiers

In the case of a question, the parser must also build a representation. But the resulting semantic formula should be interpreted using inference rules that query the system to find an answer. Questions may receive *yes* or *no* as an answer. They may also provide the value of a fact from the database.

Yes/no questions generally correspond to sentences beginning with an auxiliary verb such as *do*, *is*, *have* in English, with *Est-ce que* in spoken French, and with a verb in German. Other types of questions begin with *wh*-words such as *what*, *who*, *which* in English, with *qu*-words in French such as *quel*, *qui*, with *w*-words in German such as *wer*, *wen*.

We must bring some modifications to the parser's rules to accept questions, although basically the sentence structure remains the same. Let us suppose that we deal with very simple *yes/no* questions beginning with auxiliary *do*. The rule structure after the auxiliary is that of a declaration. Once the question has been parsed, the system must "call" the semantic fact resulting from the parsing to answer it. We do this using the `call` predicate at the end of rules describing the `sentence` structure. The system will thus succeed and report a yes, or fail and report a no:

```
sentence(Sem) -->
  [do], np(...), vp(...), {call(Sem), ...}.
```

If the sentence contains determiners, the `Sem` fact will include them. Notably, the subject noun phrase's determiner will be the predicate functor: `determiner(X, Y, Z)`. For example,

Did a waiter rush?

will produce `Sem = a(X, waiter(X), rushed(X))`.

To `call` such predicates, we must write inference rules corresponding to the determiner values. The most general cases correspond to the logical quantifiers `exists`, which roughly maps *a*, *some*, *certain*, ..., and to the universal quantifier `all`.

Intuitively, a formula such as:

```
exists(X, waiter(X), rushed(X)),
```

corresponding to the sentence:

A waiter rushed

should be mapped onto to the query:

```
?- waiter(X), rushed(X).
```

and

```
a(X, waiter(X), a(Y, meal(Y), brought(X, Y)).
```

should lead to the recursive call:

```
?- waiter(X), a(Y, meal(Y), brought(X, Y)).
```

In consequence, `exists` can be written in Prolog as simply as:

```
exists(X, Property1, Property2) :-
  Property1,
  Property2,
  !.
```

We could have replaced `exists/3` with `a/3` or `some/3` as well.

The universal quantifier corresponds to logical forms such as:

```
all(X, waiter(X), rushed(X))
```

and

```
all(X, waiter(X), a(Y, meal(Y), brought(X, Y)).
```

We map these forms onto Prolog queries using a double negation, which produces equivalent statements. The first negation creates an existential quantifier corresponding to

There is a waiter who didn't rush
and
There is a waiter who didn't brought a meal

And the second one is interpreted as:

There is no waiter who didn't rush
and
There is no waiter who didn't brought a meal

Using the same process, we translate the double negation in Prolog by the rule:

```
all(X, Property1, Property2) :-
  \+ (Property1, \+ Property2),
  !.
```

We may use an extra call to `Property1` before the negation to ensure that there are *waiters*.

14.9.3 Prolog and Unknown Predicates

To handle questions, we want Prolog to retrieve the properties that are in the database and instantiate the corresponding variables. If no facts matching these properties have been asserted before, we want the predicate call to fail. With compiled Prologs, supporting ISO exception handling, a call will raise an exception if the predicate is not in the database. Fortunately, there are workarounds. If you want that the unknown predicates fail silently, just add:

```
:- unknown(_, fail).
```

in the beginning of your code.

If you know the predicate representing the property in advance, you may define it as `dynamic`:

```
:- dynamic(predicate/arity).
```

Finally, instead of calling the predicate using

```
Property
```

or

```
call(Property)
```

you can also use

```
catch(Property,
  error(existence_error(procedure, _Proc), _),
  fail)
```

which behaves like `call(Property)` except that if the predicate is undefined it will fail.

14.9.4 Other Determiners and Questions

Other rules corresponding to determiners such as *many*, *most*, and *more* are not so easy to write as the previous ones. They involve different translations that depend on the context and application. The reader can examine some of them in the exercise list.

Questions beginning with *wh*-words are also more difficult to process. Sometimes, they can be treated in a way similar to yes/no questions. This is the case for *which* or *who*, which request the list of the possible solutions to predicate `exists`. Other *wh*-words, such as *where* or *when*, involve a deeper understanding of the context, possibly spatial or time reasoning. These cases are out of the scope of this book.

From this chapter, the reader should also be aware that the presentation has been simplified. In "real" natural language, many sentences are very difficult to translate. Notably, ambiguity is ubiquitous, even in benign sentences such as

Every caterpillar is eating a hedgehog

where two interpretations are possible.

Mapping an object must also take the context into account. If a patron says *This meal*, pointing to it with his/her finger, no ambiguity is possible. But, we then need a camera or tracking means to spot what is the user's gesture.

14.10 Application: The Spoken Language Translator

14.10.1 Translating Spoken Sentences

The Core Language Engine (CLE) (Alshawi 1992) is a workbench aimed at processing natural languages such as English, Swedish, French, and Spanish. The CLE has a comprehensive set of modules to deal with morphology, syntax, and semantics. It provides a framework for mapping any kind of sentence onto logical forms. The CLE, which was designed at the Stanford Research Institute in Cambridge, England, is implemented in Prolog.

CLE has been used in applications, the most dramatic of which is definitely the Spoken Language Translator (SLT) (Agnäs et al. 1994). This system translates spoken sentences from one language into another for language pairs such as English/Swedish, English/French, and Swedish/Danish.

Translation operates in nearly real time and has reached promising quality levels. Although SLT never went beyond the demonstration stage, it was reported that it could translate more than 70 % of the sentences correctly for certain language pairs. Table 14.8 shows examples from English into French (Rayner and Carter 1995). SLT is limited to air travel information, but it is based on principles general enough to envision an extension to any other domain.

Table 14.8 Examples of French–English translations provided by the SLT (After Rayner and Carter (1995))

English	*What is the earliest flight from Boston to Atlanta?*
French	*Quel est le premier vol Boston–Atlanta?*
English	*Show me the round trip tickets from Baltimore to Atlanta*
French	*Indiquez-moi les billets aller-retour Baltimore–Atlanta*
English	*I would like to go about 9 am*
French	*Je voudrais aller aux environs de 9 heures*
English	*Show me the fares for Eastern Airlines flight one forty seven*
French	*Indiquez-moi les tarifs pour le vol Eastern Airlines cent quarante sept*

14.10.2 Compositional Semantics

The CLE's semantic component maps sentences onto logical forms. It uses unification and compositionality as a fundamental computation mechanism. This technique makes it easy to produce a representation while parsing and to generate the corresponding sentence in the target language.

Agnäs et al. (1994, pp. 42–43) give an example of the linguistic analysis of the sentence

I would like to book a late flight to Boston

whose semantic structure corresponds to the Prolog term:

```
would(like_to(i,
              book(i,
                   np_pp(a(late(flight)),
                         X^to(X, boston)))))
```

The parse rule notation is close to that of DCGs, but instead of the rule

```
Head --> Body_1, Body_2, ..., Body_n.
```

CLE uses the equivalent Prolog term

```
rule(<RuleId>,
  Head,
  [Body_1,
   Body_2,
     ...
   Body_n])
```

Table 14.9 shows the rules involved to parse this sentence. For example, rule 1 describes the sentence structure and is equivalent to

```
s --> np, vp.
```

Rules embed variables with values in the form of pairs `Feature = Value` to implement syntactic constraints and semantic composition.

The lexicon entries follow a similar principle and map words onto Prolog terms:

```
lex(<Wordform>, <Category> (Features>))
```

Table 14.10 shows lexical entries of the sentence *I would like to book a late flight to Boston*, and Fig. 14.5 shows its parse tree.

The semantic value of words or phrases is denoted with the `sem` constant in the rules. For instance, *flight* has the semantic value `flight` (Table 14.10, line 3) and *a* has the value `a(NBAR)` (Table 14.10, line 5), where `NBAR` is the semantic value of the adjective/noun sequence following the determiner.

Table 14.9 Rules in the CLE formalism (After Agnäs et al. (1994, p. 42))

#	Rules

1.	`rule(s_np_vp,`
	`  s([sem=VP]),`
	`  [np([sem=NP,agr=Ag]),`
	`  vp([sem=VP,subjsem=NP,aspect=fin,agr=Ag])]).`

2.	`rule(vp_v_np,`
	`  vp([sem=V,subjsem=Subj,aspect=Asp,agr=Ag]),`
	`  [v([sem=V,subjsem=Subj,aspect=Asp,agr=Ag,subcat=,`
	`  [np([sem=NP])]]),`
	`  np([sem=NP,agr=_])]).`

3.	`rule(vp_v_vp,`
	`  vp([sem=V,subjsem=Subj,aspect=Asp,agr=Ag]),`
	`  [v([sem=V,subjsem=Subj,aspect=Asp,agr=Ag,`
	`  subcat=[vp([sem=VP,subjsem=Subj])]]),`
	`  vp([sem=VP,subjsem=Subj,aspect=ini,agr=])]).`

4.	`rule(vp_v_to_vp,`
	`  vp([sem=V,subjsem=Subj,aspect=Asp,agr=Ag]),`
	`  [v([sem=V,subjsem=Subj,aspect=Asp,agr=Ag,`
	`  subcat=[inf([]),vp([sem=VP,subjsem=Subj])]]),`
	`  inf([]),`
	`  vp([sem=VP,subjsem=Subj,aspect=inf,agr=])]).`

5.	`rule(np_det_nbar,`
	`  np([sem=DET,agr=(3-Num)]),`
	`  [(det([sem=DET,nbarsem=NBAR,num=Num]),`
	`  nbar([sem=NBAR,num=Num])]).`

6.	`rule(nbar_adj_nbar,`
	`  nbar([sem=ADJ,num=Num])`
	`  [adj([sem=ADJ,nbarsem=NBAR]),`
	`  nbar([sem=NBAR,num=Num])]).`

7.	`rule(np_np_pp,`
	`  np([sem=np_pp(NP,PP),agr=Ag]),`
	`  [np([sem=NP,agr=Ag]),`
	`  pp([sem=PP])]).`

8.	`rule(pp_prep_np,`
	`  pp([sem=PREP]),`
	`  [prep([sem=PREP,npsem=NP]),`
	`  np([sem=NP,agr=_])]).`

The parser composes the semantic value of the noun phrase *a flight* applying the `np_det_nbar` rule (Table 14.9, line 5) equivalent to

```
np --> det, nbar.
```

in the DCG notation. It results in `sem = a(flight).`

All the semantic values are unified compositionally and concurrently with the parse in an upward movement, yielding the sentence's logical form.

Table 14.10 Lexicon entries in the CLE formalism (After Agnäs et al. (1994, p. 42))

#	Lexicon entries
1.	`lex(boston,np([sem=boston,agr=(3-s)])).`
2.	`lex(i,np([sem,agr=(1-s)])).`
3.	`lex(flight,n([sem=flight,num=s])).`
4.	`lex(late,adj([sem=late(NBAR),nbarsem=NBAR])).`
5.	`lex(a,det([sem=a(NBAR),nbarsem=NBAR,num=s])).`
6.	`lex(to,prep([sem=X^to(X,NP),npsem=NP])).`
7.	`lex(to,inf([])).`
8.	`lex(book,v([sem=have(Subj,Obj),subjsem=Subj,aspect=ini,agr=_, subcat=[np([sem=Obj])]])).`
9.	`lex(would,v([sem=would(VP),subjsem=Subj,aspect=fin,agr=_, subcat=[vp([sem=VP,aubjsem=Subj])]])).`
10.	`lex(like,v([sem=like_to(Subj,VP),subjsem=Subj,aspect=ini, agr=_,subcat=[inf([]),vp([sem=VP,subjsem=Subj])]])).`

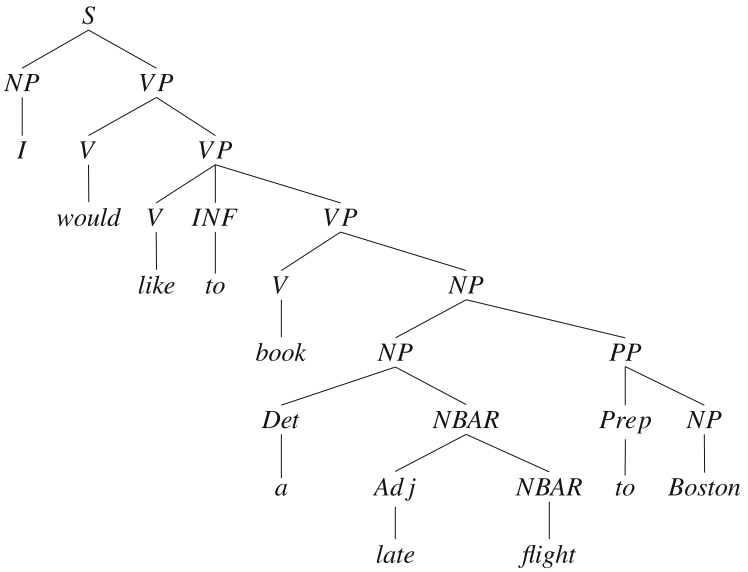


Fig. 14.5 Parse tree for *I would like to book a late flight to Boston* (After Agnäs et al. (1994, p. 43))

14.10.3 Semantic Representation Transfer

The complete CLE's semantic layer relies on two stages. The first one maps a sentence onto a so-called quasi-logical form. Quasi-logical forms are basic predicate–argument structures, as we saw in this chapter, where variables representing real objects remain uninstantiated. The second layer links these variables

to values, taking the context into account and so constructing fully resolved logical forms.

Translation from one language to another need not resolve variables. So the SLT builds a quasi-logical form from the source sentence and transfers it into the target language at the same representation level. SLT uses then a set of recursive transfer rules to match patterns in the source sentence and to replace them with their equivalent in the target language. Rules have the following format (Rayner et al. 1996):

```
trule(<Comment>
  <QLF pattern 1>
  <Operator>
  <QLF pattern 2>).
```

where `Operator` describes whether the rule is applicable from source to target (`>=`), the reverse (`=<`), or bidirectional (`==`).

Some rules are lexical, such as

```
trule([eng, fre],
      flight1 >= vol1).
```

which states that *flight* is translated as *vol*, but not the reverse. Others involve syntactic information such as:

```
trule([eng, fre],
      form(tr(relation,nn),
           tr(noun1),
           tr(noun2))
      >=
      [and, tr(noun2),
            form(prep(tr(relation)),
                 tr(noun1))]).
```

which transfers English compound nouns like *arrival time* – `noun1 noun2`. These nouns are rendered in French as: *heure d'arrivée* with a reversed noun order – `noun2 noun1` and with a preposition in-between *d'* – `prep(tr(relation))`.

14.11 RDF and SPARQL as Alternatives to Prolog

As alternative to Prolog to represent the universe of discourse, we can use tabular databases to store information and the SQL query language to extract or modify it. The resource description framework (RDF) together with SPARQL is a third option that enjoys a growing popularity. We introduce it now.

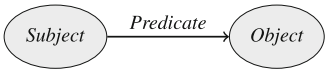


Fig. 14.6 A RDF triple consisting of two nodes *Subject* and *Object* connected by an arc with the *Predicate* label

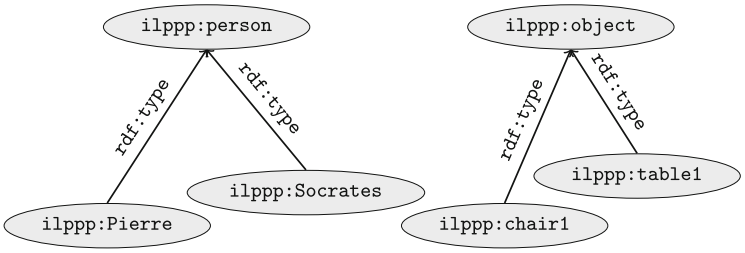


Fig. 14.7 The RDF graph representing Prolog predicates in Sect. 14.4.2

14.11.1 RDF Triples

The resource description framework represents information as a collection of triples, where each triple consists of a subject, a predicate, and an object. In Sect. 14.4.2, we encoded a small set of facts with Prolog predicates. It is easy to convert them to RDF triples. One-place predicates such as `person('Socrates')` assert the type of the argument: *Socrates is a person*. We rewrite them as triples using the built-in `rdf:type` RDF predicate:

```
ilppp:Pierre rdf:type ilppp:person.
ilppp:Socrates rdf:type ilppp:person.

ilppp:table1 rdf:type ilppp:object.
ilppp:chair1 rdf:type ilppp:object.
ilppp:chair2 rdf:type ilppp:object.
```

where each member of a triple consists of a prefix, `ilppp` or `rdf` here, a colon, an a local name; the `rdf` prefix is defined by the RDF standard, while `ilppp` is just for this book. Two-place predicates have an even more straightforward conversion from the Prolog format. We just reuse the Prolog predicate names as is in the triples and the first and second argument for the subject and object:

```
ilppp:chair1 ilppp:in_front_of ilppp:table1.
ilppp:Pierre ilppp:on ilppp:table1.
```

A set of triples forms a directed graph, where the subjects and objects are graph nodes and the predicates are the labels of the arcs linking the subjects to the objects (Fig. 14.6). Figure 14.7 shows the set of triples resulting from the translation of our small Prolog database.

RDF were designed as an extension to the World Wide Web and they followed the idea of global addresses to find the nodes. Just as http://www.springer.com/ is a universal network reference to Springer publishing, RDF nodes use names in the form of uniform resource identifiers (URIs), a variation of Web addresses. While RDF subjects and predicates are always URIs, an RDF object can either be a nonterminal node of the graph or a terminal one: a graph leaf. Accordingly, objects can be URIs or have literal values such as numbers, strings, dates, etc.

As URIs are often long and poorly legible, the RDF standard uses abridged names consisting of a namespace, what we called the prefix, a colon, and a local name. As header of our set of triples, we need to give the list of prefixes to recover the complete URIs:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix ilppp: <http://cs.lth.se/nlp/02/book#>.
```

so that the RDF predicate `rdf:type` is expanded into:

```
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
```

All the other RDF nodes or predicates would be expanded the same way using the `rdf` or `ilppp` prefixes.

14.11.2 SPARQL

Once we have stored facts in a database, Prolog enables us to extract information using queries such as:

```
?- object(X), object(Y), in_front_of(X, Y).
```

that finds that objects linked by an *in front of* relation:

```
X = chair1,
Y = table1.
```

SPARQL is the RDF query language that plays the role of Prolog for triple stores. Its syntax shows similarities with SQL and we write a query equivalent to the Prolog one this way:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ilppp: <http://cs.lth.se/nlp/02/book#>

SELECT ?x ?y
WHERE
{
  ?x rdf:type ilppp:object.
  ?y rdf:type ilppp:object.
  ?x ilppp:in_front_of ?y
}
```

Table 14.11 The nodes returned by the SPARQL query

Variables	`?x`	`?y`
Values	`ilppp:chair1`	`ilppp:table1`

where the variables start with a question mark, here `?x` and `?y`, and the lines correspond to a conjunction of goals, acting like commas in Prolog. The first and second lines of the query extract all the `?x` and `?y` that are of type ilppp:object, and the third line adds the constraint that the nodes are connected with the `ilppp:in_front_of` property.

The query returns a table, here of one line, with the values found in the triple store (Table 14.11).

14.11.3 DBpedia and Yago

DBpedia (Auer et al. 2007; Bizer et al. 2009) and Yago (Suchanek et al. 2007) are two examples of large semantic knowledge repositories based on RDF graphs. Both projects collected automatically their knowledge from the Wikipedia encyclopedia, and they used the semistructured information they could find in the entire article collection, notably the infoboxes, to derive billions of RDF triples.

Wikipedia infoboxes are tabulated data summarizing key facts of certain entries. These facts depend on the article category: if it is a country, the infobox will list the country capital, the population size, the flag, the president, king, or queen, etc. If it is a city, it lists the mayor, the population, the area, etc. The article on the Korean city of Busan (http://en.wikipedia.org/wiki/Busan) has such an infobox located at the top right of the page. Clicking on the Edit tab shows the wiki markup used to encode the infobox, abridged here:

```
{{Infobox settlement
| name                   = Busan
...
| area_total_km2         = 767.35
...
| population_total       = 3,614,950
...
}}
```

The DBpedia extraction algorithm reads all the Wikipedia articles, recognizes the infobox using its start and end delimiters, respectively `{{Infobox` and `}}`, and applies regular expressions to extract the properties and their values, such as the pairs (name, Busan), (area, 767.35), and (population, 3,614,950). DBpedia or Yago also map the named entities they extract from this process to unique identifiers following the URI format. They adopt the Wikipedia naming nomenclature and the

Table 14.12 The values returned by the DBpedia SPARQL end point

Variables	Entity	Population
Values	http://dbpedia.org/resource/Busan	3,614,950

URI of the city of Busan is dbpedia:Busan, where the `dbpedia:` prefix is the abbreviation of http://dbpedia.org/resource/.

Finally, DBpedia or Yago store the data they extract in the form of RDF triples. For Busan, this results in:

```
dbpedia:Busan foaf:name "Busan, Korea"@en .
dbpedia:Busan dbpedia-owl:populationTotal "3614950".
dbpedia:Busan dbpedia-owl:areaTotal "7.6735E8" .
...
```

with the prefixes:

```
@prefix dbpedia: <http://dbpedia.org/resource/>.
@prefix dbpedia-owl: <http://dbpedia.org/ontology/>.
@prefix foaf: <http://xmlns.com/foaf/0.1/>.
...
```

The RDF format makes it easier to link the DBpedia or Yago graphs to other similar data graphs, and using SPARQL, it is possible to extract knowledge as with tabular database queries. DBpedia provides such a SPARQL end point (http://dbpedia.org/sparql), where the query:

```
SELECT ?entity ?population
WHERE
{
  ?entity foaf:name "Busan, Korea"@en.
  ?entity dbpedia-owl:populationTotal ?population.
}
```

identifies the entity named *Busan, Korea* in English (`@en` tag), its population, and returns the values in Table 14.12.

14.12 Further Reading

Relations between logic and language have been a core concern for logicians, linguists, and philosophers. For a brief presentation and a critical discussion on philosophical issues, you may read Habermas (1988, Chap. 5). The reader can also find good and readable introductions in *Encyclopédie philosophique universelle* (Jacob 1989) and in Morton (2003).

Modern logic settings stem from foundational works of Frege (1879), Peirce (1885, 1897), and Herbrand (1930). Later, Robinson (1965) proposed algorithms

to implement logic programs. Robinson's work eventually gave birth to Prolog (Colmerauer 1970, 1978). Burke and Foxley (1996) provide a good introductory textbook on logic and notably on Herbrand bases. Sterling and Shapiro (1994) also give some insights on relations between Prolog and logic. Textbooks with examples of semantic processing in Prolog include Pereira and Shieber (1987), Gal et al. (1989, 1991), Covington (1994b), and Blackburn and Bos (2005).

Some books attribute the compositionality principle to Frege (1879). In fact, he said exactly the opposite. The investigation of rational ways to map sentences onto logical formulas dates back to the ancient Greeks and the Middle Ages. Later, Montague (1974) extended this work and developed it systematically to English. Montague has had a considerable influence on modern developments of research in this area. For a short history of compositionality, see Godart-Wendling et al. (1998). The *Handbook of Logic and Language* (van Benthem and Ter Meulen 1997) provides a comprehensive treatment on current theories in the field. A shorter and very readable introduction on the philosophy of language is that of Taylor (1998).

Text is the largest repository of human knowledge, and the availability of internet encyclopedias like Wikipedia, that anybody can download, or large corpora in many languages has made it possible to apply a systematic extraction of this knowledge. This was the idea behind projects harvesting structured information from text such as Yago, DBpedia, or Google's Knowledge Graph (Singhal 2012). Yago and DBpedia publish their results in the form of predicate–argument structures (RDF triples) with unique entity identifiers that other applications can reuse. The RDF format makes it relatively easy to link them to other data graphs or data sets. The Geonames geographic database is an example of this (http://www.geonames.org/). For an introduction to the RDF format and SPARQL, Allemang and Hendler (2011) is a good and legible reference.

Exercises

14.1. Write facts to represent
Tony is a hedgehog
A hedgehog likes caterpillars
Tony likes caterpillars
All hedgehogs likes caterpillars

14.2. Write DCG rules to get the semantic structure out of sentences of Exercise 14.1.

14.3. Write DCG rules to obtain the semantic representation of noun phrases made of one noun and one and more adjectives such as *The nice hedgehog*, *the nice little hedgehog*.

14.4. Write rules accepting sentences with embedded relative clauses, such as *The waiter that ran brought a meal* and producing a logical form out of them:

```
the(X, (waiter(X), ran(X)), a(Y, meal(Y), brought(X, Y))
```

14.5. Write rules to carry out the semantic interpretation of determiner *two*, as in the sentence *Two waiters rushed*.

14.6. Write rules to carry out the semantic interpretation of determiner *No*, as in *No waiter rushed.*

14.7. Write rules to carry out the semantic interpretation of *how many*, as in *how many waiters rushed.*

14.8. Write rules to parse questions beginning with relative pronouns *who* and *what* in sentences, such as *Who brought the meal?* and *What did the waiter bring?* and build logical forms out of them.

14.9. Write a small dialogue system accepting assertions and questions and answering them. A transcript of a session could be:

User: *the patron ordered the meal*
System: *OK*
User: *who brought the meal*
System: *I don't know*
User: *who ordered the meal*
System: *the patron*
User: *the waiter brought the meal*
System: *OK*
User: *who brought the meal*
System: *the waiter*

14.10. Some sentences such as *all the patrons ordered a meal* may have several readings. Cite two possible interpretations of this sentence and elaborate on them.

14.11. Download a dump of Wikipedia in the language you like (http://dumps.wikimedia.org/) and write DCG rules or regular expressions to extract the infoboxes.

14.12. Write DCG rules to parse the infoboxes you extracted in Exercise 14.11 and extract one or more properties from them, for instance, the population size. Should you prefer it, you can use regular expressions instead of DCG rules.

Chapter 15
Lexical Semantics

> Τῶν κατὰ μηδεμίαν συμπλοκὴν λεγομένων ἕκαστον ἤτοι οὐσίαν σημαίνει ἢ ποσὸν ἢ ποιὸν ἢ πρός τι ἢ ποὺ ἢ ποτὲ ἢ κεῖσθαι ἢ ἔχειν ἢ ποιεῖν ἢ πάσχειν. ἔστι δὲ οὐσία μὲν ὡς τύπῳ εἰπεῖν οἷον ἄνθρωπος, ἵππος· ποσὸν δὲ οἷον δίπηχυ, τρίπηχυ· ποιὸν δὲ οἷον λευκόν, γραμματικόν· πρός τι δὲ οἷον διπλάσιον, ἥμισυ, μεῖζον· ποὺ δὲ οἷον ἐν Λυκείῳ, ἐν ἀγορᾷ· ποτὲ δὲ οἷον χθές, πέρυσιν· κεῖσθαι δὲ οἷον ἀνάκειται,κάθηται· ἔχειν δὲ οἷον ὑποδέδεται, ὥπλισται· ποιεῖν δὲ οἷον τέμνειν, καίειν· πάσχειν δὲ οἷον τέμνεσθαι, καίεσθαι.
>
> Aristotle, *Categories*, IV. See translation in Sect. 15.2.

15.1 Beyond Formal Semantics

15.1.1 La langue et la parole

Formal semantics provides clean grounds and well-mastered devices for bridging language and logic. Although debated, the assumption of such a link is common sense. There is obviously a connection – at least partial – between sentences and logical representations. However, there are more controversial issues. For instance, can the whole language be handled in terms of logical forms? Language practice, psychology, or pragmatics are not taken into account. These areas pertain to cognition: processes of symbolization, conceptualization, or understanding.

Bibliography on nonformal semantics is uncountable. Let us have a glimpse at it with Ferdinand de Saussure (1916), the founder of modern linguistics. Much of Saussure's work, but not exclusively, was devoted to the area of what we would call now real-world semantics. He first made the distinction between the cultural background of a community of people of a same language embodied in words and grammatical structures and physical messages of individuals expressed by the

P.M. Nugues, *Language Processing with Perl and Prolog*, Cognitive Technologies,
DOI 10.1007/978-3-642-41464-0_15, © Springer-Verlag Berlin Heidelberg 2014

means of a tongue. He called these two layers language and speech – *la langue et la parole*, in his words.

15.1.2 Language and the Structure of the World

Starting from the crucial distinction between *langue* and *parole*, Saussure went on to consider linguistic values of words, taking examples from various languages (Saussure 1916, Chap. 4). Comparing words to economic units, Saussure described them as structural units tied together into a net of relationships. These units would have no sense isolated, but taken together are the mediators between thought and the way individuals express themselves. Accepting Saussure's theory, languages are not only devices to communicate with others but also to seize and understand reality. This entails that the structure of knowledge and thought is deeply intermingled within a linguistic structure. And of course, to fit communication, this device has to be shared by a community of people.

In this chapter, we limit ourselves to some aspects on how a language and, more specifically, words relate to the structure of the world. Words of a specific tongue also embed a specific view of the universe. We believe that most concepts are common to all languages and can be structured in the same way. However, certain words cover concepts somewhat differently according to languages. In addition, the ambiguity they introduce is puzzling since it rarely corresponds from one language to another. We present techniques to structure a lexicon and to resolve ambiguity. Within this framework, we examine verb structures and case grammars that provide us with a way to loop back to sentence representation and to formal semantics.

15.2 Lexical Structures

15.2.1 Some Basic Terms and Concepts

To organize words, we must first have a clear idea of what they express. In dictionaries, this is given by definitions. **Definitions** are statements that explain the meaning of words or phrases. Some words have nearly the same definition and hence nearly the same meaning. They are said to be **synonyms**. In fact, perfect synonyms are rare if they even exist. We can relax the synonymy definition and restate it as: synonyms are words that have the same meaning in a specific context. Synonymy is then rather considered as a graded similarity of meaning. **Antonyms** are words with opposite meanings.

Contrary to synonymy, a same word – or string of characters – may have several meanings. It is then said to be ambiguous. Word ambiguity is commonly divided between **homonymy** (or **homography**) and **polysemy**:

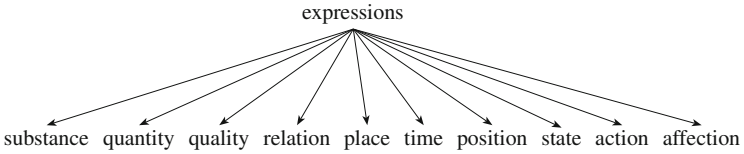


Fig. 15.1 Aristotle's ontology

- When words of a same spelling have completely unrelated meanings, such as for the strings *lot* in *a lot of* and *a parking lot*, they are said to be **homonyms** or **homographs**.
- When a word extends its meaning from concrete to abstract and to concepts tied by analogy, it is said to be **polysemous**. Consider the example of *tools* used in *computer tools* and in *carpenter tools*, where the latter is a concrete object and the former a computer program.

15.2.2 Ontological Organization

There are several ways to organize words within a lexicon. Most dictionaries for European languages sort words alphabetically. An obvious advantage of this method is to provide easy access to words. However, alphabetical organization is of little help when we want to process semantic properties. A more intuitive way is to organize words according to their meaning. The lexicon structure then corresponds to broad categories where we arrange and group the words. Such a classification certainly better reflects the structure of our knowledge of the world and is more adequate for semantic processing.

A first classification dates back to ancient Greek philosophy, when Aristotle established his famous division of words into ten main categories (Fig. 15.1). Such a lexicon structure, and beyond it, the representation of the world it entails, is often called an **ontology** in computational linguistics.

> Expressions, which are in no way composite, signify substance, quantity, quality, relation, place, time, position, state, action, or affection. To sketch my meaning roughly, examples of substance are 'man' or 'the horse', of quantity, such terms as 'two cubits long' or 'three cubits long', of quality, such attributes as 'white', 'grammatical'. 'Double', 'half', 'greater', fall under the category of relation; 'in the market place', 'in the Lyceum', under that of place; 'yesterday', 'last year', under that of time. 'Lying', 'sitting', are terms indicating position; 'shod', 'armed', state; 'to lance', 'to cauterize', action; 'to be lanced', 'to be cauterized', affection.
>
> Aristotle, *Categories*, IV. (trans. E. M. Edghill)

We can deepen the classification hierarchy. Aristotle's substance is what we could call now an entity. It includes *man* and *horse* as well as *meal* and *table*. It is easy to introduce further divisions between these words. To refine them, we

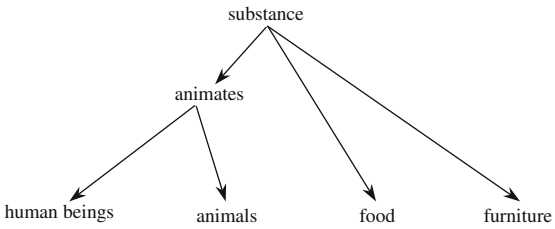


Fig. 15.2 Extending Aristotle's ontology

insert new nodes under the *substance* class. Figure 15.2 shows a symbolic tree distinguishing between *animates*, *human beings*, *animals*, *food*, and *furniture*. This tree representation – now ubiquitous – is traditionally attributed to Porphyry.

15.2.3 Lexical Classes and Relations

An ontological structure defines classes and relationships relative to each word of the lexicon. The most obvious way to group words within an ontological tree is to cut a branch under a word. The branch then contains the **hyponyms** of that word: more specific and specialized terms. For instance, hyponyms of *animals* are *mammals*, *carnivores*, *felines*, or *cats*. We can go the reverse direction, from specific to more general, and abstract heading to the root of the tree. Thus we get the **hypernyms** of a word. Hypernyms of *hedgehogs* are *insectivores*, *mammals*, *animals*, and *substance*.

It is easy to express hypernymy and hyponymy using Prolog facts. Let us define the `is_a/2` predicate to connect two concepts. We can represent the hierarchy of the previous paragraph as:

```
%% is_a(?Word, ?Hypernym)

is_a(hedgehog, insectivore).
is_a(cat, feline).
is_a(feline, carnivore).
is_a(insectivore, mammal).
is_a(carnivore, mammal).
is_a(mammal, animal).
is_a(animal, animate_being).
```

Hypernymy and hyponymy are reversed relationships and are both transitive. This can trivially be expressed in Prolog:

```
hypernym(X, Y) :- is_a(X, Y).
hypernym(X, Y) :- is_a(X, Z), hypernym(Z, Y).

hyponym(X, Y) :- hypernym(X, Y).
```

Beyond the tree structure, we can enrich relationships and link parts to the whole. *Feet*, *legs*, *hands*, *arms*, *chest*, and *head* are parts of *human beings*. This relation is called **meronymy**. Meronymy is also transitive. That is, if *nose*, *mouth*, *brain* are meronyms of *head*, they are also meronyms of *human beings*. Again it is easy to encode this relation using Prolog facts. Let us use the `has_a/2` predicate:

```
%% has_a(?Word, ?Meronym).

has_a(human_being, foot).
has_a(human_being, leg).
has_a(human_being, hand).
has_a(human_being, arm).
has_a(human_being, chest).
has_a(human_being, head).
has_a(head, nose).
has_a(head, mouth).
has_a(head, brain).
```

The opposite of meronymy is called **holonymy**.

15.2.4 Semantic Networks

We can generalize the organization of words and knowledge and extend it to any kind of relationships that may link two concepts. Words are figured as a set of nodes, and relationships are labeled arcs that connect them. This representation is called a semantic network (Quillian 1967).

Figure 15.3 shows an extension of Fig. 15.2 where we have added the relations *eat* and *possess*. As we see, the graph contains two *eat* links: the first one between *carnivores* and *meat*, and the second one between *animates* and *food*. Once a semantic net has been designed, we search relations between two concepts, climbing up from specific to general. Inheritance enables us then to assign relations `eat(X, meat)` to nodes `X` under *carnivores*, and `eat(Y, food)` to other nodes `Y` under *animates*.

Inheritance makes the design of a semantic network easier; therefore the core structure of the graph remains centered on hypernymy, that is, “is a” links. Other properties come as a supplement to it. There are then many ways to augment a net. Design decisions depend on the application. The verbs linking words representing the agent of an action and its object are among common and useful arcs.

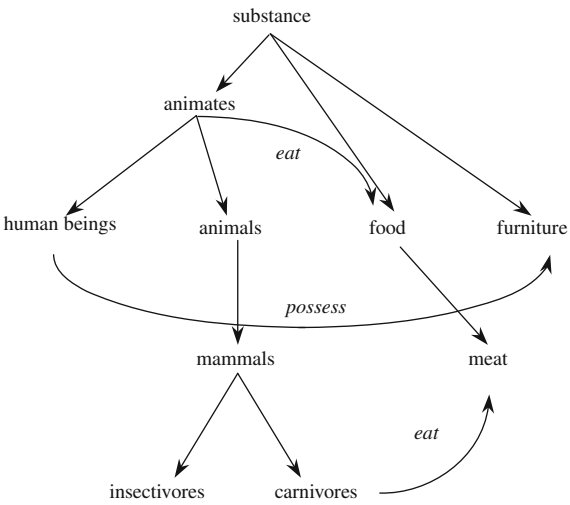


Fig. 15.3 A semantic network

15.3 Building a Lexicon

Dictionaries – or lexicons – are repositories of a language's words. They are organized as a set of entries – the words – containing one or more senses. Current dictionaries attempt to itemize all the senses of words and typically contain more than 50,000 entries. Others are focused on specific domains. Dictionaries associate words or senses with grammatical models and definitions. Grammatical models such as the part of speech or a verb's conjugation class indicate morphological and syntactic properties of words; this enables their lemmatization and parsing. Models can also extend to semantic and pragmatic classifications. Many dictionaries cross-reference words using synonyms and give usage examples to show how a word is used in context.

As we could have guessed, wide-coverage lexical databases are central to most natural language processing applications. Instead of creating a new base from scratch, many computerized dictionaries have been derived from existing paper lexicons and transcribed in a computer-readable format, which are called machine-readable dictionaries (MRDs). Computerized dictionaries often take the structure of their paper counterparts and are organized as a set of entries corresponding to word senses with their syntactical model, semantic annotations, and definition.

Learner or nonnative speaker dictionaries are often preferred as primary resources to derive lexical databases. They describe precisely pronunciation and syntactical features, such as a verb's subcategory or an inflection paradigm, while other dictionaries sometimes take it for granted by native speakers. Some dictionaries also tie words to specialized domains with labels such as: anatomy, computer science, linguistics, etc., or to general semantic codes: life, body, people, food, etc. Finally, most learner's dictionaries define each entry with a controlled

vocabulary limited to two to three thousand words. This ensures a consistency in definitions, ease of understanding, and avoids circular – looping – definitions.

General lexicographic sources for English include the *Longman Dictionary of Contemporary English* (LDOCE) (Procter 1978), the *Oxford Advanced Learner's Dictionary* (OALD) (Hornby 1995), the *Collins Cobuild English Language Dictionary* (COBUILD) (Sinclair 1987) or the *Cambridge International Dictionary of English* (CIDE) (Procter 1995). Among them, the computerized version of the LDOCE gained the largest popularity within the academic computational linguistics community.

15.3.1 The Lexicon and Word Senses

As we saw in Chap. 1, many words are ambiguous, that is, a same string of letters has more than one meaning. Most dictionaries arrange homonyms that have clearly different meanings under different entries. The OALD (1995 edition) lists three entries for *bank*: two nouns, *organization* and *raised ground*, and a verb *turn*. Polysemy, which refers to meaning variations within a same entry, is subtler. Dictionaries divide entries into submeanings with more or less precision according to the dictionary. These are the senses of a word. Let us take the example of the sentence

The patron ordered a meal

to realize concretely what word senses are. We will annotate each word of the sentence with its correct sense, and we will use definitions of the OALD to carry out this operation.

In the sentence, there are three content words: *patron*, *order*, and *meal*. For each of these words, the OALD lists more than one sense. *Patron* has one main entry for which the dictionary makes out two meanings:

1. A person who gives money or support to a person, an organization, a cause or an activity
2. A customer of a shop, restaurant, theater

Order has two entries. The first one is a noun and the other is a verb for which the OALD details four sub-meanings:

1. To give an order to somebody
2. To request somebody to supply or make goods, etc.
3. To request somebody to bring food, drink, etc. in a hotel, restaurant, etc.
4. To put something in order

And finally, *meal* has two entries – two homographs – one as in *oatmeal*, and the other being divided into two submeanings:

1. An occasion where food is eaten
2. The food eaten on such occasion

Table 15.1 Sense ambiguity in the sentence *The patron ordered a meal*

Words	Definitions	OALD sense numbers
The patron	**Correct sense:**	
	A customer of a shop, restaurant, theater	**1.2**
	Alternate sense:	
	A person who gives money or support to a person, an organization, a cause or an activity	**1.1**
ordered	**Correct sense:**	
	To request somebody to bring food, drink, etc. in a hotel, restaurant etc.	**2.3**
	Alternate senses:	
	To give an order to somebody	**2.1**
	To request somebody to supply or make goods, etc.	**2.2**
	To put something in order	**2.4**
a meal	**Correct sense:**	
	The food eaten on such occasion	**1.2**
	Alternate sense:	
	An occasion where food is eaten	**1.1**

Table 15.2 Some verb constructions

English	*depend* + *on* + object noun group
	I like + verb-*ing* (gerund)
	require + verb-*ing* (gerund)
French	*dépendre* + *de* + object noun group
	Ça me plaît de + infinitive
	demander + *de* + infinitive
German	*hängen* + *von* + dative noun group + *ab*
	es gefällt mir + *zu* + infinitive
	verlangen + accusative noun group

That is, with such a simple sentence, we already have 16 choices ($2 \times 4 \times 2$; Table 15.1).

Classically, senses of a word are numbered relatively to a specific dictionary using the entry number and then the sense number within the entry. So *requesting somebody to bring food, drink, etc. in a hotel, restaurant, etc.*, which is the 3rd sense of the 2nd entry of *order* in the OALD is denoted **order (2.3)**. The proper sense sequence of *The patron ordered a meal* is then *patron* (1.2) *order* (2.3) *meal* (1.2).

15.3.2 Verb Models

Dictionaries contain information on words' pronunciations, parts of speech, declension, and conjugation models. Some enrich their annotations with more precise syntactic structures such as the verb construction. In effect, most verbs constrain their subject, object, or adjuncts into a relatively rigid construction (Table 15.2).

Some dictionaries such as the OALD or the LDOCE provide the reader with this argument structure information. They include the traditional transitive and intransitive verb distinction, but descriptions go further. The OALD itemized 28 different types of verb patterns. Intransitive verbs, for example, are subdivided into four categories:

- **V** are verbs used alone.
- **Vpr** are verbs followed by a prepositional phrase.
- **Vadv** are verbs followed by an adverb.
- **Vp** are verbs followed by a particle.

A verb entry contains one or more of these models to indicate possible constructions.

Some dictionaries refine verb patterns with semantic classes. They indicate precisely the ontological type of the subject, direct object, indirect object, and sometimes adjuncts. Verbs with different argument types will be mapped onto as many lexical senses. For instance, Rich and Knight (1991) quote three kinds of *wanting*:

1. Wanting something to happen
2. Wanting an object
3. Wanting a person

We can map the 2nd construction onto a DCG rule specifying it in its arguments:

```
%% word(+POS, +Construction, +Subject, +Object)

word(verb, transitive, persons, phys_objects) --> [want].
```

Argument types enforce constraints, making sure that the subject is a person and that the object is a physical object. These are called **selectional restrictions**. They may help parsing by reducing syntactic ambiguity.

The LDOCE lists selectional restrictions of frequent verbs that give the expected semantic type of their subject and objects. It uses semantic classes such as inanimate, human, plant, vehicle, etc. The Collins Robert French–English dictionary (Atkins 1996) is another example of a dictionary that includes such ontological information with a large coverage.

15.3.3 Definitions

The main function of dictionaries is to provide the user with definitions, that is, short texts describing words. The typical definition of a noun first classifies it in a *genus proximum* or superclass using a hypernym. Then, it describes in which way the noun is specific using attributes to differentiate it from other members of the superclass. This part of the definition is called the *differentia specifica*. Examples from the OALD include (general in bold and specific underlined):

bank (1.1): ***a land*** *sloping up along each side of a canal or a river*.
hedgehog: ***a small animal*** *with stiff spines covering its back*.
waiter: ***a person*** *employed to serve customers at their table in a restaurant, etc*.

from *Le Robert Micro* (Rey 1988)

bord (1.1): ***contour, limite, extrémité*** *d'une surface*.
hérisson (1.1): ***petit mammifère*** *au corps recouvert de piquants, qui se nourrit essentiellement d'insectes*.
serveur (1.1): ***personne*** *qui sert les clients dans un café, un restaurant*.

and from *Der kleine Wahrig* (Wahrig 1978)

Ufer (1.1): ***Rand*** *eines Gewässers, Gestade*.
Igel (1.1): ***ein kleines insektfressendes Säugetier*** *mit kurzgedrungenem Körper und auf dem Rücken aufrichtbaren Stacheln*.
Ober (1.2) –> Kellner: ***Angestellter*** *in einer Gaststätte zum Bedienen der Gäste*.

15.4 An Example of Exhaustive Lexical Organization: WordNet

WordNet (Fellbaum 1998; Miller 1995) is a lexical database of English. It is probably the most successful attempt to organize word information with a computer. It has served as a research model for other languages such as Dutch, German, Italian, and Spanish. A key to this success is WordNet's coverage – it contains more than 120,000 words – and its liberal availability online: users can download it under the form of Prolog facts from its home at Princeton University.

WordNet arranges words or word forms along with word meanings into a lexical matrix (Fig. 15.4). The lexical matrix addresses both synonymy and polysemy. A horizontal line defines a set of synonymous words – a *synset* in WordNet's parlance. A column shows the different meanings of a word form. In Fig. 15.4, F_1 and F_2 are synonyms (both have meaning M_1) and F_2 is polysemous (it has meanings M_1 and M_2). Synsets are the core of WordNet. They represent concepts and knowledge that they map onto words.

From synonymy and synsets, WordNet sets other semantic relations between words, taking their part of speech into account. WordNet creators found this property relevant, citing cognitive investigations: when people have to associate words spontaneously, they prefer consistently to group words with the same part of speech rather than words that have a different one.

WordNet considers open-class words: nouns, verbs, adjectives, and adverbs. It has set aside function words. According to classes, the organization and relationships between words are somewhat different. However, semantic relations remain based on synsets and thus are valid for any word of a synset.

Word meanings	**Word forms**				
	F_1	F_2	...	...	F_n
M_1	$E_{1,1}$	$E_{1,2}$			
M_2		$E_{2,2}$			
M_m					$E_{m,n}$

Fig. 15.4 The lexical matrix (Miller et al. 1993)

{*act, action, activity*}	{*food*}	{*possession*}
{*animal, fauna*}	{*group, collection*}	{*process*}
{*artifact*}	{*location, place*}	{*quantity, amount*}
{*attribute*}	{*motive*}	{*relation*}
{*body, corpus*}	{*natural object*}	{*shape*}
{*cognition, knowledge*}	{*natural phenomenon*}	{*state, condition*}
{*communication*}	{*person, human being*}	{*substance*}
{*event, happening*}	{*plant, flora*}	{*time*}
{*feeling, emotion*}		

Fig. 15.5 WordNet's 25 semantic primes

{*entity, something*}	{*state*}	{*group, grouping*}
{*psychological feature*}	{*event*}	{*possession*}
{*abstraction*}	{*act, human action, human activity*}	{*phenomenon*}

Fig. 15.6 Nouns' top nodes

15.4.1 Nouns

WordNet singles out 25 primitive concepts or **semantic primes** (Fig. 15.5), and it partitions the noun set accordingly. Within each of the corresponding topics, WordNet uses a hypernymic organization and arranges nouns under the form of a hierarchical lexical tree. WordNet contains 95,000 nouns.

In addition to the 25 base domains, WordNet adds top divisions (Fig. 15.6). This enables it to gather some classes and to link them to a single node. Figure 15.7 shows the hierarchy leading to {*thing, entity*}.

To picture the word hierarchy and synsets with an example, let us take *meal*. It has two senses in WordNet:

1. *meal, repast* – (the food served and eaten at one time)
2. *meal* – (coarsely ground foodstuff; especially seeds of various cereal grasses or pulse)

For sense 1, synonyms are *nutriment*, *nourishment*, *sustenance*, *aliment*, *alimentation*, and *victuals*; and hypernyms are (from the word up to the root):

- *nutriment, nourishment, sustenance, aliment, alimentation, victuals* – (a source of nourishment)

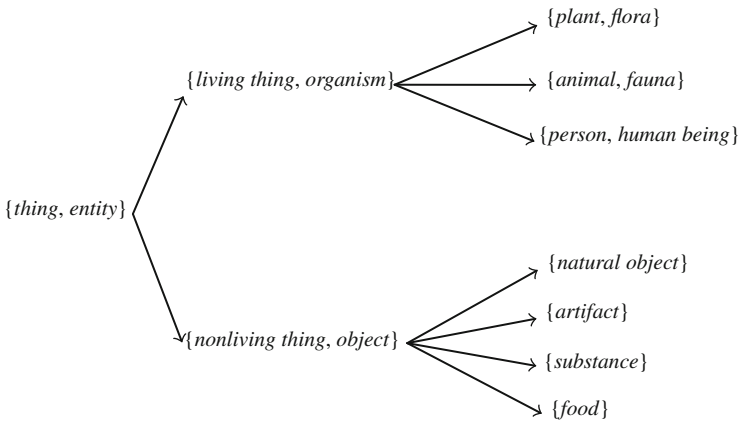


Fig. 15.7 {*thing, entity*} node of WordNet's hierarchy

- *food, nutrient* – (any substance that can be metabolized by an organism to give energy and build tissue)
- *substance, matter* – (that which has mass and occupies space; "an atom is the smallest indivisible unit of matter")
- *object, physical object* – (a physical (tangible and visible) entity; "it was full of rackets, balls, and other objects")
- *entity, something* – (anything having existence (living or nonliving))

15.4.2 Adjectives

WordNet divides adjectives into two general classes: descriptive and relational, and into a more specific one: color adjectives. WordNet contains 20,000 adjectives.

Descriptive adjectives modify a noun and qualify one of its attributes. Examples include *hot* and *cold*, as in *hot meal* and *cold meal*, where *hot* and *cold* both describe the temperature attribute of *meal*. Another example is *heavy* and *light*, which give a value to the weight attribute of a noun (more precisely to the object it represents). As for other words, adjectives are grouped into synsets, and for each adjective synset, there is a link to the attribute it describes.

In addition to synonymy, WordNet uses antonymy as a core concept to organize descriptive adjectives. It clusters all of them around bipolar couples: word–antonym together with their respective synsets. *Hot* and *cold* or *wet* and *dry* are typical couples of antonyms, and WordNet enumerates 2,500 of them.

Antonymy relation, however, is not valid for all the members of a synset. *Torrid* is a synonym of *hot* but it cannot be considered as an antonym of *cold*. To cope with this, WordNet makes a distinction between bipolar antonymy and opposite concepts – or indirect antonyms. There is no direct antonym for *torrid*, but using its synset, WordNet can link it indirectly to *cold* via *hot*.

Table 15.3 Name and description of verb files provided with the WordNet 1.6 distribution

File	Description
Body	Verbs of grooming, dressing, and bodily care
Change	Verbs of size, temperature change, intensifying, etc.
Cognition	Verbs of thinking, judging, analyzing, doubting
Communication	Verbs of telling, asking, ordering, singing
Competition	Verbs of fighting and athletic activities
Consumption	Verbs of eating and drinking
Contact	Verbs of touching, hitting, tying, digging
Creation	Verbs of sewing, baking, painting, performing
Emotion	Verbs of feeling
Motion	Verbs of walking, flying, swimming
Perception	Verbs of seeing, hearing, feeling
Possession	Verbs of buying, selling, owning
Social	Verbs of political and social activities and events
Stative	Verbs of being, having, spatial relations
Weather	Verbs of raining, snowing, thawing, thundering

Relational adjectives (**pertainyms**) such as *fraternal*, *contextual*, or *dental* are modified nouns and behave much like them on the semantic side, although they have the syntactic properties of adjectives. WordNet encodes them with a reference to their related noun: *fraternal* with *fraternity* or *brother*, *contextual* with *context*, and *dental* with *teeth* or *dentistry*. As opposed to descriptive adjectives, WordNet does not associate them to an attribute.

15.4.3 Verbs

WordNet partitions verbs into 15 categories. Fourteen of these categories are semantic domains: bodily functions and care, change, cognition, communication, competition, consumption, contact, creation, emotion, motion, perception, possession, social interaction, and weather. A last part contains verbs referring to states: verbs of being, having, and spatial relations (Table 15.3). WordNet has a total of 10,300 verbs.

The first relation WordNet sets between verbs is synonymy, as for other words. However, synonymy is more delicate to delimit because verb meanings are quite sensitive to the context. That is, two verbs with apparently the same meaning, such as *rise* and *ascend*, do not occur with the same type of subject. This is a general case, and most verbs are selective with the type of their nominal arguments: subject, object, or adjunct. Moreover, as a verb often has no lexical synonym, WordNet encodes synsets with a small explanation – a gloss. For example, the verb *order* has nine senses whose sense 2 is represented by the synset {*order, make a request for something*}. *Bring* has 11 senses, and sense 1 is the synset {*bring, convey, take, take something or somebody with oneself somewhere*}.

Then, WordNet organizes verbs according to principles similar to hyponymy and meronymy for nouns. However, it cannot apply these principles directly because they do not match exactly that of nominals. WordNet replaces them respectively with troponymy and entailment.

WordNet designers found the `is_a` relationship not relevant or clumsy for verbs (Fellbaum 1998, p. 79):

> *to amble* is a kind of *to walk* is not a felicitous sentence.

To name specializations of more generic verbs, they coined the word **troponyms**. *Amble* is then a troponym of *walk*. This is roughly a kind of verbal hyponymy related to a manner, a cause, or an intensity that makes the description of an action more precise. Since *tropos* is a Greek word for manner or fashion, it enables us to rephrase the hierarchical relation between *amble* and *walk* as *to amble is to walk in a particular manner*.

The second principle of verb organization is **entailment** – or implication – as *he is snoring* implies *he is sleeping*, or *she is limping* implies *she is walking*. Relations between verbs in these examples are not of the same kind. The latter is related to troponymy: *limping* is a specialization or an extension of *walking*. The former is an inclusion: the action of *snoring* is always included in an action of *sleeping*. In total, WordNet makes out four kinds of entailments. In addition to extension and inclusion, the two other entailments are backward presupposition – an action must have been preceded by another one, as with the pair *succeed/try* – and cause – an action leads to another one, as with *give/have*.

15.5 Automatic Word Sense Disambiguation

Although ambiguity is ubiquitous in texts, native speakers recognize the proper sense of a word intuitively. In the beginning of computational linguistics, some people declared it a human faculty impossible to reproduce automatically. This is no longer the case. There have been considerable improvements recently, and researchers have good reason to believe that a computer will be able to discriminate among word senses. Here we will present an overview of techniques to carry out word sense disambiguation, that, alone or combined, show promising results.

15.5.1 Senses as Tags

Let us again consider the sentence *The patron ordered a meal*. Solving ambiguity has an obvious definition. It consists in linking a word with its correct sense entry in a dictionary. We can recast this as a tagging problem. We regard a dictionary as a sense inventory and senses as a finite set of labels that we call semantic tags. The number of tags per word ranges from one to more than a dozen. Ambiguous words

can receive several tags, and disambiguation consists in retaining one single tag – a unique sense – per word. Semantic tagging applies most frequently to open-class words: nouns, verbs, adjectives, and adverbs.

Compared to part-of-speech tagging, a major contrast comes from the tagset. Dictionaries have somewhat different classifications of word senses and there is no complete consensus on them. According to authors, sense division is also finer or coarser. Applications can even use sets of word senses specifically designed for them. If broad categories of senses are common across most dictionaries and can be transcoded, there always will be some cases with a lack of agreement. The dissimilarity in classification and how we perceive senses can be measured by asking two or more different people to annotate a same text. The agreement between manual annotators – or **interannotator agreement** – is usually significantly lower for semantic tagging than, say, for part-of-speech annotation. There is no definitive solution to it, however. We must be aware of it and live with it.

We can carry out semantic tagging using techniques similar to those that we have used with parts of speech. Namely, we can resort to numerical or symbolic techniques. Numerical techniques attempt to optimize a sequence of semantic tags using statistics from a hand-annotated corpus. Symbolic techniques apply constraints to discard wrong semantic readings and retain the good ones.

SemCor (Landes et al. 1998) is a frequently used resource to train systems for English. It comes as a freely available corpus in which all the words are annotated with the WordNet nomenclature.

15.5.2 Associating a Word with a Context

The basic idea of most disambiguation techniques is to use the context of a word (Wilks et al. 1996, Chap. 11). The noun *bank*, for example, has two major senses[1] that will probably appear in clear-cut contexts. Sense one (**bank1**) resorts to finance and money; sense 2 (**bank2**) pertains to riversides and sloping ground. Context may be given by the words of the sentence or of the paragraph where the word occurs. This means that, depending on the words surrounding *bank* or what the text is about, a reader can select one of its two senses.

Some finer and more local relations such as the order of two words or the grammatical relations may also give the context. Disambiguating *meal* in *The patron ordered a meal* requires such considerations, because the two senses of this word belong to the same topic:

1. An occasion where food is eaten
2. The food eaten on such occasion

[1] OALD lists a third sense of *bank* as being a row of similar objects.

15.5.3 Guessing the Topic

The idea of this technique is first to define a limited number of topics, that is, a list of general areas, to attach a topic to each sense of words, and then to guess the topic (or topics) of a paragraph or of a sentence. This technique implies that correct word senses will make the paragraph topic converge and enable us to discard senses attached to other topics. To make disambiguation possible, topics must, of course, be different along with each sense of a word.

According to applications, topics may come from dictionaries that flag some words with broad classifications – subject tags. For instance, the LDOCE categorizes words with 300 subject codes or domains that we can use as topics: agriculture, business, economics, engineering, etc. These tags usually annotate more specialized words.[2] Topics could also be a small set of hypernyms drawn from a comprehensive lexical database. For instance, using WordNet, **bank1** could be attached to financial institution (finance or institution), and **bank2** to slope:

```
%% topic(?Word, ?OALD_Sense, ?Topic).

topic(bank, bank1, [finance, institution]).
topic(bank, bank2, [slope]).
```

The disambiguation algorithm operates on a context that corresponds to a sequence of words such as a paragraph, a fixed number of sentences, or a fixed number of words, from 10 to 100, where the topic is supposed to be stable. A procedure annotates the words in the window with the possible subject tags when they are available. It yields possible sense sequences. The algorithm then retains the sense sequence that has the maximum of subject tags in common. A variation of this algorithm annotates nouns only. This method is referred to as a **bag-of-words** approach because it does not take the word order into account.

15.5.4 Naïve Bayes

The naïve Bayes classifier is an alternate statistical strategy that uses the bag-of-words approach. It also computes the sense of a word given its context. For a polysemous word w with n senses s_1, s_2, ..., s_n, the context C is defined as the sequence of words surrounding it: $w_{-m}, w_{-m+1}, \ldots, w_{-1}, w, w_1, \ldots, w_{m-1}, w_m$. The optimal sense $\hat{s}$ corresponds to $\arg\max_{s_i, 1 \leq i \leq n} P(s_i|C)$.

Using Bayes' rule, we have:

[2]LDOCE annotates the rest of the nonspecialized words with another set of semantic codes: the key concepts.

$$\hat{s} = \arg\max_{s_i, 1 \le i \le n} P(s_i) P(C|s_i),$$
$$= \arg\max_{s_i, 1 \le i \le n} P(s_i) P(w_{-m}, w_{-m+1}, \ldots, w_{-1}, w_1, \ldots, w_{m-1}, w_m | s_i).$$

And using the bag-of-words assumption, we replace

$$P(w_{-m}, w_{-m+1}, \ldots, w_{-1}, w_1, \ldots, w_{m-1}, w_m | s_i)$$

with the product of probabilities:

$$\prod_{j=-m, j\neq 0}^{m} P(w_j | s_i).$$

This enables us to compute the optimal sense:

$$\hat{s} = \arg\max_{s_i, 1 \le i \le n} P(s_i) \prod_{j=-m, j\neq 0}^{m} P(w_j | s_i),$$

where $P(s_i)$ and $P(w_j|s_i)$ are both estimated from hand-annotated corpora.

15.5.5 Using Constraints on Verbs

As we saw, most verb arguments have a structure and a semantic type that is relatively rigid. Another set of disambiguation techniques exploits these properties and takes verb constructions and local relations into account. We start here from clauses, and for each one we detect the verb group and noun groups. The idea is to apply the selectional restrictions imposed by the verb group to its depending noun groups and thus reject wrong senses.

This technique needs a group detector and a shallow parser to identify the verbs' subject and object. The sense tagger operates on headwords, that is, here on the main noun and the main verb of each group. The tagger goes through the verbs that it annotates with their possible semantic constructions. It also annotates nouns with their possible senses. Finally, for each verb sense, the tagger retains subject and object senses that agree with the selectional restrictions.

Although this technique sets aside some parts of sentences, such as adjuncts, it reduces ambiguity and can be used with a combination of other techniques. In contrast to the previous technique, it has a more local viewpoint.

In addition, we can operate a disambiguation within groups using other selectional restrictions on adjectives and adverbs. We need to extend the description of adjectives with features giving the semantic type of the noun they expect to modify.

Adverbs also have to include their modifier type. As an example, the word *mean* can have the properties of being an adjective and of qualifying only persons:

```
%% word(+Category, +Qualify)

word(adjective, persons) --> [mean].
```

15.5.6 Using Dictionary Definitions

We saw that using the naïve Bayes approach to tag senses in unrestricted texts requires an immense hand-annotation effort. It is possible to avoid it using unsupervised methods. Unsupervised methods have no training step or are trained on raw texts. These techniques are very appealing, especially in word sense disambiguation, because they avoid the need for human labor to annotate the words.

Wilks and Stevenson (1997) described an algorithm that only uses word definitions from general dictionaries as semantic resource. Their method was inspired by a paper by Lesk (1986).

The algorithm tags each word with all its possible senses listed in the dictionary and links each sense with its definition in a dictionary. It first applies constraints on parts of speech and then identifies the context using the definitions: it selects senses whose definitions overlap best within the range of a window of N words, a sentence, or a paragraph. This is made easier with dictionaries, such as the LDOCE, whose definitions are written using a controlled defining vocabulary. Simplified main steps of the program are:

1. A name recognition module identifies the proper nouns of the text.
2. A lemmatization module transforms each word into its canonical form. It associates each content word with its set of possible senses listed in the dictionary and with the corresponding textual definitions. Words occurring in definitions are also lemmatized.
3. A part-of-speech tagger annotates each word with its part of speech. At this step, the program can discard some senses because they have grammatical categories different from that of the words in the sentence.
4. The algorithm then computes the definition overlap for each sequence of possible senses. The overlap function considers a sequence of senses and their textual definition – one definition per word. The algorithm concatenates definitions of this sequence and counts the occurrences of each definition word: n. Each definition word brings a score of $n - 1$. So, if a definition word appears once, it will contribute nothing to the function; if it appears twice, it will contribute 1, and so on. Then, the algorithm adds up the counts and associates this score to the sense sequence.
5. The algorithm retains the sequence that has the maximum overlap, which is the largest number of definition words in common.

Wilks and Stevenson (1997) improved this algorithm using topics as defined in Sect. 15.5.3. Basically, they compute an overlap function for topics within the range of a paragraph:

6. The algorithm annotates nouns of a paragraph with possible subject tags when available. It retains the sequence that has the maximum of subject tags in common. This computation is similar to that of step 4.
7. The results of steps 4 and 6 are combined in a simplistic way. When both tags do not correspond, the first one in the dictionary entry list is retained. This is based on the assumption that entries are ordered by frequency of occurrence.

Step 4 of this algorithm can lead to very intensive computations. If a sentence has 15 words with 6 senses each, it leads to $6^{15} \sim 4.7\ 10^{11}$ intersections. Wilks and Stevenson used simulated annealing to approximate the function. See also Wilks et al. (1996, Chap. 11).

15.5.7 An Unsupervised Algorithm to Tag Senses

Yarowsky (1995) proposed a slightly supervised and effective algorithm based on two assumptions on sense distribution:

- Nearby words provide strong clues to the sense of a word. This means that a word has **one sense per collocation**.
- The sense of a word is consistent within any given document. This means that a word has **one sense per discourse**.

The algorithm is basically a classifier. Given a polysemous word w with n senses $s_1, s_2, \ldots, s_n$ and a set of examples of the word surrounded by the neighboring words, the algorithm assigns each example a class corresponding to one of the senses. Each word in the examples is defined by a set of features, which are, as for naïve Bayes, the surrounding words. The algorithm starts from a few manually annotated examples that serve as a seed set to derive incrementally a sequence of classifiers for the remaining unlabeled examples. It uses an objective function that measures the performance of the classification. The algorithm is repeated until it has classified all the examples.

The algorithm has an initialization step and two loops. It extracts the set of all the examples of word w with the surrounding words from the training corpus. It results in N contexts $c_1, \ldots, c_N$ of, say, ten words, centered around w. These examples will be the input. In his original article, Yarowsky used the word *plant* and its two main senses $s_1 = \textit{living}$ and $s_2 = \textit{factory}$. The algorithm gradually annotates all the examples of the corpus with one of the two senses. It produces a sequence of annotated corpora $\mathit{Corpus}(0), \mathit{Corpus}(1), \ldots, \mathit{Corpus}(n)$, and builds classifiers that correspond to the sets of collocations of the first sense, Coll_1^k, and of the second one, Coll_2^k. $\mathit{Corpus}(0)$ is the original, unannotated set of examples.

1. **Initialization**. This step manually identifies initial collocations, and the first sense classifier tags the examples whose context contains one of the collocations with the corresponding sense label. Yarowsky used the words *life* for the first sense, $Coll_1^1 = \{life\}$, and *manufacturing* for the second one, $Coll_2^1 = \{manufacturing\}$. Both words enabled the disambiguation of 2% of the examples in $Corpus(1)$.
2. **Outer Loop**. This loop uses the "one sense per collocation" principle. It identifies the examples where the intersection of the context and one of the collocation sets is nonempty: $c_k \cap Coll_i^j \neq \emptyset$ with $1 \leq k \leq N$, $i = 1, 2$, and j is the iteration index of the loop. It annotates the corresponding examples with the sense s_i. It results in $Corpus(j)$. In Yarowsky's paper, contexts of *plant* that contained one word of the first set were tagged with the first sense, and others that contained one word of the second set were tagged with the second sense. The algorithm applies optionally the "one sense per discourse" constraint.
   - **Inner Loop**. The objective function determines for each sense other collocations that partition the training data $Corpus(j)$ and ranks them by the purity of the distribution. It builds new sets of classifiers $Coll_i^{j+1}$ with collocations where the objective function is above a certain threshold. This step identifies *cell*, *microscopic*, *animal*, and *species* as collocates of the first sense $Coll_1^{j+1} = \{life, cell, microscopic, animal, species\}$ and *equipment*, *employee*, and *automate* as collocates of the second sense: $Coll_2^{j+1} = \{manufacturing, equipment, employee, automate\}$.
3. Repeat the outer loop until it converges (the partition is stable).

The algorithm identifies collocations with an objective function that determines the "strongest feature." It uses the log-likelihood ratio that is defined for a word w with two senses as $\log \frac{P(Sense_1|w_k)}{P(Sense_2|w_k)}$. It ranks the resulting values depending on w_k for all w_k members of the contexts $w_{-m}, w_{-m+1}, \ldots, w_{-1}, w, w_1, \ldots, w_{m-1}, w_m$, where the collocations the most strongly tied to a specific sense show the largest values, either positive or negative.

The "one sense per collocation" principle implies that counts of 0 are frequent. In another paper, Yarowsky (1996) describes techniques to smooth data. Once the collocation sets have been built, the resulting classifiers can be applied to other corpora.

15.5.8 Senses and Languages

Word senses do not correspond in a straightforward way across languages. In a famous comparison, Hjelmslev (1943) exemplified it with the values of French words *arbre* 'tree', *bois* 'wood', and *forêt* 'forest' and their mapping onto German and Danish scales (Figure 15.8). He went on and remarked that the word covering the material sense in French (*bois*) and in Danish (*træ*) could also have the plant

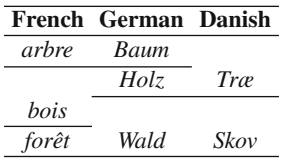


Fig. 15.8 Values of *arbre*, *bois*, and *forêt* in German and Danish

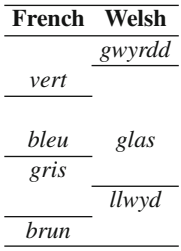


Fig. 15.9 Color values in French and Welsh

sense but in different ways: a group of trees in French, a single tree in Danish. In a more striking example, Hjelmslev cited color naming that is roughly common to European languages with the exception of Celtic languages such as Welsh or Breton, which does not makes the same distinction between blue and green (Figure 15.9).

There are many other examples where one word in English can be rendered by more words in French or German, or the reverse. Finding the equivalent word from one language to another often requires identifying its correct sense in both languages. It is no great surprise that word sense disambiguation was attempted first within the context of automatic machine translation projects.

This raises some questions about the proper granularity of sense division for a translation application. In some cases, sense division that is available in monolingual dictionaries is not sufficient and must be split within as many senses as there are in both languages combined. In other cases, all senses of one word correspond from one language to another. Therefore their distinction is not necessary and the senses can be merged. This problem is still wide open and is beyond the scope of this book.

15.6 Case Grammars

15.6.1 Cases in Latin

Some languages, like Latin, Russian, and to a lesser extent German, indicate grammatical functions in a sentence by a set of inflections: the cases. Basically, Latin cases are relative to the verb, and a case is assigned to each noun group: the noun and its depending adjectives. Latin has six cases that we can roughly associate to a semantic property:

- **Nominative** marks the subject of the sentence.
- **Accusative** indicates the object of the verb.

- **Dative** describes the beneficiary of a gift or of an action. It corresponds to the indirect object of the verb.
- **Genitive** describes the possession. As opposed to other cases, it is relative to a noun that the word in the genitive modifies or qualifies.
- **Ablative** describes the manner, the instrument, or the cause of an action. It corresponds to the adjunct function.
- **Vocative** is used to name and to address a god or a person.
- **Locative** is a seventh and an archaic case. It indicates the location of the speaker in some particular expressions.

Latin, like Russian, has quite a flexible word order. That is, we can arrange words of a sentence in different manners without modifying its meaning. The subject can appear at the beginning as well as at the end of a sentence. It has no specific location as in English or in French.

A flexible word order makes cases necessary for a sentence to be understandable. They indicate functions of groups: *who did what to whom, when, and where* and hence the arguments of a verb. Searching the subject, for example, corresponds to searching the noun phrase at the nominative case. Let us apply these principles to parse the following example:

Servus	*senatoris*	*domino*	*januam*	*clave*	*aperit*
Slave	senator	master	door	key	opens

Aperit is the verb in the third-person singular of present and means open (*aperire*). It is the predicate relative to which nouns will be the arguments. Each Latin noun has a model of inflection, also called a declension, five in total. *Servus* follows the second declension and means the slave. It is in the nominative case and hence is the subject of the sentence. *Senatoris*, third declension, is the genitive case of *senator* and is the noun complement of *servus*. *Domino*, second declension, means master and is the dative of *dominus*. It corresponds to the indirect object of the verb. *Januam*, first declension, is the accusative of *janua* – door – and is the object. Finally, *clave*, third declension, is the ablative of *clavis* – the key – and the instrument of the action. Once we have identified cases, we can safely translate the sentence as:

> The slave of the senator opens the door to the master with a key.

Cases are also useful to discover what goes with what such as an adjective and its head noun. Both will have the same case even if the noun group is fragmented within the sentence.

15.6.2 Cases and Thematic Roles

Case grammars stem from the idea that each verb – or each verb sense – has a finite number of possible cases. Case grammars rest on syntactic and semantic

Table 15.4 Examples of case frames (Fillmore 1968, p. 27)

Sentences	Case frames
The door opened	[O = door, (I), (A)]
John opened the door	[O = door, (I), (A) = John]
The wind opened the door	[O = door, (I), (A) = wind]
John opened the door with a chisel	[O = door, (I) = chisel, (A) = John]

observations of languages like Latin and offer a framework to represent sentences. Hjelmslev (1935–1937), and more recently, Fillmore (1968) are known to have posited that cases were universal and limited to a handful. Because of declensions, cases are obvious to those who learned Latin. However, it is somewhat hidden to speakers of English or French only. That is probably why, compared to compositionality, the acceptance of the case theory and its transposition to English or French has been slower.

Surveying a set of languages ranging from Estonian to Walapai, Fillmore percolated a dozen core cases, or **thematic roles**. A first classification led him to define (Fillmore 1968, p. 24):

- **Agentive** (A) – the case of the instigator of the action, which is typically animate
- **Instrumental** (I) – the case of the force or object, typically inanimate, causing the event
- **Dative** (D) – the case of the entity typically animate affected by the action
- **Factitive** (F) – the case of the object or being resulting from the event
- **Locative** (L) – the case of the identifying the place of the event or the orientation of the action
- **Objective** (O) – the most general case indicating the entity that is acted upon or that changes

As an example, Fillmore (1968, p. 27) attached to the verb *open* a frame containing an objective case that always occurs in the sentence, and optional instrumental and agentive cases denoted in parentheses: [O, (I), (A)]. This frame enables us to represent sentences in Table 15.4. One must note that the objective case, here filled with *the door*, sometimes corresponds to the grammatical subject and sometimes to the grammatical object.

To be complete and represent our Latin sentence, we add a dative case:

The slave of the senator opens the door to the master with a key.
[O = the door, (I) = a key, (A) = the slave of the senator, (D) = the master]

Later a multitude of authors proposed extensions to these cases. Most general and useful are:

- **Source** – the place from which something moves
- **Goal** – the place to which something moves
- **Beneficiary** – the being, typically animate, on whose behalf the event occurred
- **Time** – the time at which the event occurred

Table 15.5 *Bring* cases with constraints

Case	Type		Value
Agentive	Animate	(Obligatory)	*The waiter*
Objective (or theme)		(Obligatory)	*the meal*
Dative	Animate	(Optional)	*the patron*
Time		(Obligatory)	past

Over the time, Fillmore himself slightly changed the structure and name of his cases. Here is a more abstract classification of cases together with their description.

- **Agent** – primary animate energy source
- **Experiencer** – psychological locus of an experience
- **Theme** – primary moving object
- **Patient** – object which undergoes a change
- **Source** – starting point of a motion or change
- **Goal** – destination, target of a motion
- **Location** – location of an object or event
- **Path** – trajectory of a motion, between source and goal
- **Content** – content of an event of feeling, thinking, speaking, etc.

Some verbs do not fit into this case scheme, in spite of its generality. Fillmore again cited some of them such as the verb set *buy*, *sell*, *pay*, *spend*, *charge*, etc., whose cases are the quadruplet **buyer**, **seller**, **goods**, **money**, and the set *replace*, *substitute*, *swap*, etc., whose cases are **old**, **new**, **position**, **causer**. In addition, some applications may require other more specific cases.

15.6.3 Parsing with Cases

Parsing with the case grammar formalism transforms a sentence – or a part of it – into a kind of logical form: the frame. The predicate is the main verb, and its arguments represent the cases (or the roles). The parsing process merely maps noun groups or other features such as the tense or adverbs onto the cases. According to the verbs, some cases will be obligatory, such as the agent for most verbs. They will be assigned with exactly one argument. Others cases will be optional. They will be assigned with at most one value. In addition, cases are constrained by an ontological type. Table 15.5 shows a representation of the sentence

The waiter brought the meal to the patron

which links noun groups and the verb tense to cases.

We can relate verbs cases to Tesnière's *actants* and *circonstants* (1966), which are idiosyncratic patterns of verbs encapsulated into a predicate argument structure. Tesnière first made a distinction between the typical cases of a verb – *actants*

– and its optional modifiers – *circonstants*. A verb attracts a definite number of actants corresponding to its **valence**. Drawing on the semantic side, cases fit well an ontology of nouns and lead to subcategories of verb patterns. The agent, or the subject, of verb *eat* is generally animate. The instrument of *open* should comply with the instrument ontological subtree. These semantic properties related to verb cases are another viewpoint on sectional restrictions.

15.6.4 Semantic Grammars

Originally, parsing with a case grammar was carried out using a combination of techniques: shallow parsing and "expectations" on verb arguments. First, the parser detects the verb group and its depending noun groups or noun phrases. Then, the parser fills the cases according to "markers": topological relations, ontological compatibility (selectional restrictions), prepositions, tense, and for German, syntactic cases.

In many circumstances, we can assimilate the Agent to the subject of a sentence and the Theme to the object. Languages like English and French have a rather rigid word order in a sentence, and functions correspond to a specific location relative to a the verb. The subject is generally the first noun phrase; the object comes after the verb. In German, they are inflected respectively with the nominative and accusative cases.

Adjuncts are more mobile, and a combination of constraints on prepositions and selectional restrictions can be productive to fill modifier cases such as Source, Goal, and Instrument. Prepositions such as *from*, *to*, or *into* often indicate a Source and a Goal. We can add a double-check and match them to location classes such as places, cities, countries, etc. Other prepositions are more ambiguous, such as *by* in English, *pour* in French, and *auf* in German. Ontological categories come first as conditions to carry out the parse. They enable us to attach noun groups to classes and to choose a case complying with the selectional restrictions of the verb.

Phrase-structure rules can help us implement a limited system to process cases. It suffices to replace parts of speech and phrase categories with ontological classes in rules. This leads to **semantic grammars** dedicated to specific applications, such as this one describing a piece of the real and gory life of animals:

```
sentence --> np_insectivores, ingest, np_crawling_insects.
np_insectivores --> det, insectivores.
np_crawling_insects --> det, crawling_insects.

insectivores --> [mole].
insectivores --> [hedgehog].
ingest --> [devoured].
ingest --> [ate].
crawling_insects --> [worms].
crawling_insects --> [caterpillars].
det --> [the].
```

Rules describe prototypic situations, and parsing checks the compatibility of the types in the sentence. They produce a semantic parse tree.

Semantic grammars were once popular because they were easy to implement. However, they are limited to one application. Changing context or simply modifying it often requires a complete redesign of the rules.

15.7 Extending Case Grammars

15.7.1 FrameNet

The FrameNet research project started from Fillmore's theory on case grammars (1968). Reflecting on it, Fillmore noticed how difficult (impossible?) it was to work out a small set of generic cases applicable to all the verbs. He then altered his original ideas to form a new theory on **frame semantics** (Fillmore 1976). With frame semantics, Fillmore no longer considers universal cases but a set of frames resembling predicate–argument structures, where each frame is specific to a class of verbs. Frames are supposed to represent prototypical conceptual structures shared by a language community, i.e., here English.

FrameNet is a concrete outcome of the frame semantics theory. It aims at describing the frame properties of all the English verbs as well as some nouns and adjectives, and at annotating them in a large corpus. Like WordNet, FrameNet takes the shape of an extensive lexical database, which associates a word sense to a frame with a set frame elements (FEs). FrameNet also links the frames to annotations in the 100-million word British National Corpus.

Ruppenhofer et al. (2010) list Revenge as an example of frame, which features five frame elements: Avenger, Punishment, Offender, Injury, and Injured_party. The Revenge frame serves as a semantic model to 15 lexical units, i.e., verb, noun, or adjective senses:

> *avenge.v, avenger.n, get back (at).v, get_even.v, retaliate.v, retaliation.n, retribution.n, retributive.a, retributory.a, revenge.n, revenge.v, revengeful.a, revenger.n, vengeance.n, vengeful.a*, and *vindictive.a*

where the *.v* suffix denotes a verb, *.n* a noun, and *.a* an adjective.

Once the frame was defined, the FrameNet team annotated the corresponding lexical units in sentences extracted from its corpus. The annotation identifies one lexical unit per sentence, which is the **target**, and brackets its frame elements as in these examples from Ruppenhofer et al. (2010, Chapter 3):

1. [$_{\langle Avenger \rangle}$ His brothers] **avenged** [$_{\langle Injured\_party \rangle}$ him].
2. With this, [$_{\langle Avenger \rangle}$ El Cid] at once **avenged** [$_{\langle Injury \rangle}$ the death of his son].
3. [$_{\langle Avenger \rangle}$ Hook] tries to **avenge** [$_{\langle Injured\_party \rangle}$ himself] [$_{\langle Offender \rangle}$ on Peter Pan] [$_{\langle Punishment \rangle}$ by becoming a second and better father].

Table 15.6 The valence patterns of *avenge* in the sentences above and their three levels of annotations: frame element (FE), phrase type (PT), and grammatical function (GF)

Sent. 1	*avenge*	FE	Avenger	Injured_party		
		PT	NP	NP		
		GF	Ext	Object		
Sent. 2	*avenge*	FE	Avenger	Injury		
		PT	NP	NP		
		GF	Ext	Obj		
Sent. 3	*avenge*	FE	Avenger	Injured_party	Offender	Punishment
		PT	NP	NP	PP	PPing
		GF	Ext	Obj	Comp	Comp

Each frame element contains semantic and grammatical information split into three levels of annotation. The first level is the name of the semantic role. The second and third ones describe how a frame element is realized in the sentence: the phrase syntactic category and its grammatical function. The phrase syntactic category, i.e., noun phrases, prepositional phrases, and so on, is called the phrase type (PT). FrameNet uses a small set of grammatical functions (GFs), which are specific to the target's part of speech (i.e., verbs, adjectives, prepositions, and nouns). For the verbs, FrameNet defines four GFs: Subject, Object (Obj), Complement (Comp), and Modifier (Mod), i.e., modifying adverbs ended by *-ly* or indicating manner. FrameNet renames the subjects as external arguments (Ext).

Table 15.6 shows the three-level annotation of the sentences above. Altogether, these levels form a **valence group**. Each sentence shows a **valence pattern**, a specific set of valence groups.

15.7.2 The Proposition Bank

The Proposition Bank, or Propbank (Palmer et al. 2005), has goals similar to FrameNet, although its initial focus was more on the annotation of all the propositions of a large body of text than on the thorough description of a lexicon of predicates and arguments. Propbank started from the syntactic annotation of the Penn Treebank that we saw in Sect. 11.3 and added a layer of predicate–argument structures to it for all the verbs in the treebank. The idea was that this annotation could be used to train statistical classifiers and then build more easily semantic parsers.

Propbank models the observed predicate–argument structures of each verb with a specific role set. The verb *accept*, for example, has one sense, denoted *accept.01*, and four roles represented by the arguments Arg0, Arg1, Arg2, and Arg3:

Roleset: **accept.01**, *take willingly*

Roles:
Arg0: acceptor

Arg1: thing accepted
Arg2: accepted-from
Arg3: attribute

While *accept.01* has four arguments, most verbs in Propbank have only two, Arg0 and Arg1, or sometimes three, Arg2. The maximum is of six arguments for certain verbs of motion. The thematic roles are specific to each verb sense, but they follow some rules: Arg0 and Arg1 generally correspond to the agent and patient of Fillmore's case grammar. The other arguments have a relatively variable role: Arg2 is usually either an instrument, a benefactive, or an attribute, Arg3, a starting point, Arg4, an ending point, and Arg5, a direction.

Using the roles defined for *accept*, Propbank annotates the Penn Treebank sentence (Palmer et al. 2005):

He wouldn't accept anything of value from those he was writing about.

with one predicate and five arguments:

[$_{\text{Arg}_0}$ He] [$_{\text{ArgM-MOD}}$ would] [$_{\text{ArgM-NEG}}$ n't] **accept** [$_{\text{Arg}_1}$ anything of value] [$_{\text{Arg}_2}$ from those he was writing about] .

In addition to the verb-specific roles numbered from Arg0 to Arg5, the annotation includes 13 possible semantic adjuncts or modifiers, such as the time or the location, that can apply to any verb (Table 15.7). In the previous sentence, we had two of these modifiers: a modal, *would* and a negation, *n't*.

A verb in Propbank can have two or more senses when its description requires different role sets. The verb *kick* has six senses numbered from *kick.01*, *kick.02* to *kick.06*, where the two first senses correspond respectively to *drive or impel with the foot* and *begin*. Each of these senses has its own role set as here for sense 1 and 2:

Roleset: **kick.01**, *drive or impel with the foot*

Roles:
Arg0: kicker
Arg1: thing kicked
Arg2: instrument (defaults to foot)

Roleset: **kick.02**, *begin*

Roles:
Arg0: causer of beginning
Arg1: thing beginning

In the sentence:

John tried to kick the football, but Mary pulled it away at the last moment.

to kick the football is an infinitive clause, where *kick* has sense 01. Although *kick* has no grammatical subject here, we can consider that *John*, the subject of *tried*, also plays the role of subject for *kick*. We saw in Sect. 11.2.3 that this phenomenon is called a movement in Chomsky's grammar using the analogy that the subject would have moved from *kick* to *try*. The Penn Treebank marks what would be the

Table 15.7 The semantic adjuncts or modifiers in Propbank

AM-DIR	Directionals	AM-REC	Reciprocals	AM-ADV	Adverbials
AM-LOC	Locatives	AM-PRD	Secondary predication	AM-MOD	Modals
AM-MNR	Manner	AM-PNC	Purpose	AM-NEG	Negation
AM-TMP	Time	AM-CAU	Cause		
AM-EXT	Extent	AM-DIS	Discourse		

original subject position of *kick* as a trace, here *trace*-1, and Propbank reuses it at the semantic level:

[$_{\text{Arg}_0}$ John-1] tried [$_{\text{Arg}_0}$ *trace*-1] to **kick** [$_{\text{Arg}_1}$ the football], but Mary pulled it away at the last moment.

In the sentence:

While program trades swiftly kicked in, a "circuit breaker" that halted trading in stock futures in Chicago made some program trading impossible.

kick has sense 02 and we use the corresponding role set to annotate it as:

While [$_{\text{Arg}_1}$ program trades] [$_{\text{ArgM-MNR}}$ swiftly] **kicked in**, a "circuit breaker" that halted trading in stock futures in Chicago made some program trading impossible.

While Propbank annotates only verb predicates, a parallel project, Nombank (Meyers et al. 2004) annotated the nominal predicates. Nombank tried when possible to follow the Propbank nomenclature and transpose the roles from the verbs to the nouns. For example, *acceptance.01* is modeled after *accept.01*:

Roleset: **acceptance.01**, *take willingly*, source verb-accept.01

Roles:
- Arg0: acceptor
- Arg1: thing accepted
- Arg2: accepted-from
- Arg3: attribute

and the phrase *government acceptance of its bid for control of Jaguar* is annotated as:

[$_{\text{Arg}_0}$ government] **acceptance** [$_{\text{Arg}_1}$ of its bid for control of Jaguar]

15.7.3 Annotation of Syntactic and Semantic Dependencies

In 2008 and 2009, the conference on natural language learning (CoNLL) organized an evaluation of parsers for syntactic and semantic dependencies. The corpus annotation builds on the CoNLL tabular format that we saw for the parts of speech and morphology in Sect. 6.6 and for syntax in Sect. 11.9.2, and adds columns to the right to represent the semantic roles.

Table 15.8 The semantic annotation of a sentence simplified from CoNLL 2008 and 2009

Lexical and morphological level					Syntactic level		Semantic level		
ID	Form	Lemma	POS	Feats	Head	Deprel	Sense	APred1	APred2
1	The	the	DT	_	4	NMOD	_	_	_
2	luxury	luxury	NN	_	3	NMOD	_	A1	_
3	auto	auto	NN	_	4	NMOD	_	A1	_
4	maker	maker	NN	_	7	SBJ	maker.01	A0	A0
5	last	last	JJ	_	6	NMOD	_	_	_
6	year	year	NN	_	7	TMP	_	_	AM-TMP
7	sold	sell	VBD	_	0	ROOT	sell.01	_	_
8	1,214	1,214	CD	_	9	NMOD	_	_	_
9	cars	car	NNS	_	7	OBJ	_	_	A1
10	in	in	IN	_	7	LOC	_	_	AM-LOC
11	the	the	DT	_	12	NMOD	_	_	_
12	U.S.	u.s.	NNP	_	10	PMOD	_	_	_

Table 15.8 shows the annotation of the sentence:

The luxury auto maker last year sold 1,214 cars in the U.S.

simplified from Surdeanu et al. (2008). This annotation goes from the words to semantics, and we can roughly divide it into three main levels:

1. The lexical and morphological level with the columns ID, Form, Lemma, POS, Feats corresponding respectively to the word index, the word form, the part of speech, and the grammatical features. The annotation in these columns is similar to the one we saw in Sect. 6.6.
2. The syntactic level with the columns Head and Deprel that show the dependencies and the grammatical functions as in Sect. 11.9.2. And finally,
3. The semantic level with the columns Sense, APred1, and APred2 that show the predicates and the arguments:
   - The sense column marks the verbal and nominal predicates using the Propbank and Nombank role sets.
   - The remaining columns, from APred1 to APred*n*, contain the argument labels for the each semantic predicate. There are as many APred columns as there are predicates, and the column order follows the predicate order in the sense column, i.e., the first argument column, APred1, corresponds to the first predicate in the sense column, and so on.

The sentence in Table 15.8 has two predicates: one verb, sell.01, and one noun, maker.01. The CoNLL format associates two argument columns to these two predicates. The nominal predicate maker.01 occurs first in the sentence and is associated with the first argument column, APred1; the verbal predicate sell.01 is associated with APred2. Had the sentence contained five predicates, we would have had five argument columns.

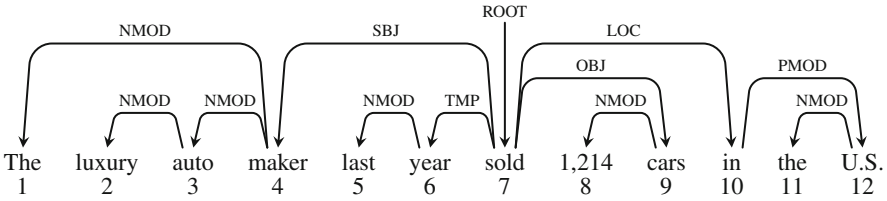


Fig. 15.10 The syntactic dependencies

	The	luxury	auto	maker	last	year	sold	1,214	cars	in	the	U.S.
maker.01		A1		A0								
sell.01	A0				AM-TMP			A1		AM-LOC		

Fig. 15.11 The semantic analysis in the form of segments

For each predicate, the arguments in the corresponding column mark the head of the semantic roles. Table 15.8 shows that sell.01 is associated with the roles A0 and A1 as well as with two adjuncts: AM-TMP and AM-LOC, whose respective heads are *maker*, *cars*, *year*, and *in*, and where the Propbank role set for sell.01 is:

Roleset: **sell.01**, *commerce: seller*

Roles:
- Arg0: seller
- Arg1: thing sold
- Arg2: buyer
- Arg3: price paid
- Arg4: benefactive

We can derive the complete semantic roles using the syntactic dependencies in Fig. 15.10 and the subtrees defined by the descendant nodes of each semantic head. We call such a subtree a yield and the resulting constituent, a yield string, if the graph is projective. Using the yields, we can extract the roles associated with *sell* in the sentence:

A0: *The luxury auto maker*
A1: *1,214 cars*
AM-TMP: *last year*
AM-LOC: *in the U.S.*

The creation of segments from heads may involve more complex cases when the graph is nonprojective or when the yield strings are overlapping. Figure 15.11 shows a graphical representation of the roles using the segment creation algorithm described by Johansson and Nugues (2008a) and the MATE tools semantic processing pipeline (Björkelund et al. 2010).

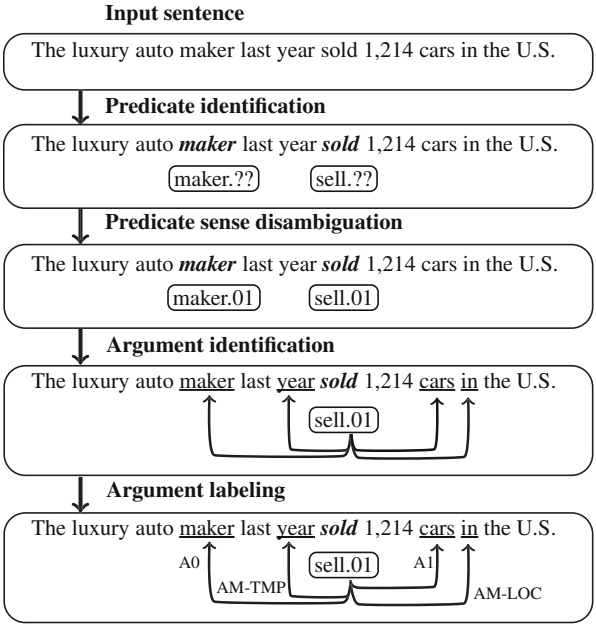


Fig. 15.12 The sequence of classifiers: predicate identification, predicate sense disambiguation, argument identification, and argument labeling

15.7.4 A Statistical Method to Identify Semantic Roles

We saw that it was possible to develop a case parser using manually written rules. However, such parsers require much labor, testing, and debugging, and have an unavoidably limited coverage. We introduce now a statistical technique to identify semantic roles for unrestricted text. In our description, we will follow the CoNLL 2008 annotation that uses the Propbank and Nombank roles and Johansson and Nugues (2008b) that reported the best figures in the evaluation.

Almost all the semantic role labelers, also called semantic parsers, start with a parsing step, which takes a sentence as input and produces either dependencies or constituents. As for CoNLL 2008, we will use dependency structures as input to the semantic analyzer. This analyzer then consists of a sequence of classifiers that identifies the predicates in the sentence, the sense of each predicate, for each predicate, its arguments in the form of head words, and finally the label of each argument. Figure 15.12 shows this sequence on the sentence *The luxury auto maker last year sold 1,214 cars in the U.S.* for the predicate *sell*.

Table 15.9 The features used for the predicate identification and disambiguation steps. The extracted values correspond to the word *sold* in the sentence *The luxury auto maker last year sold 1,214 cars in the U.S.*

Feature	Value
PREDFORM	sold
PREDLEMMA	sell
PREDHEADFORM	ROOT
PREDHEADPOS	ROOT
PREDDEPREL	ROOT
CHILDFORMSET	{maker, year, cars, in}
CHILDPOSSET	{NN, NNS, IN}
CHILDDEPSET	{SBJ, TMP, OBJ, LOC}
DEPSUBCAT	SBJ+TMP+OBJ+LOC
CHILDFORMDEPSET	{maker+SBJ, year+TMP, cars+OBJ, in+LOC}
CHILDPOSDEPSET	{NN+SBJ, NN+TMP, NNS+OBJ, IN+LOC}

Predicate Identification

This step considers all the words in the input sentence and decides if a word is a predicate or not using a binary classifier. In Fig. 15.12, the classifier identified *maker* and *sell* as predicates. To make the decision, the classifier extracts features from the input words and the dependency structures of the training set and trains a model. The classifier then uses this model to identify the predicates of unannotated sentences.

In CoNLL 2008, Johansson and Nugues (2008b) used logistic regression and trained two classifiers, one for the verbs and one for the nouns, with the following features to decide whether a word is a predicate or not:

PREDFORM, PREDLEMMA: Lexical form and lemma of the word;
PREDHEADFORM, PREDHEADPOS: Form and part of speech of the head node of the word;
PREDDEPREL: Dependency relation between the word and its head;
CHILDFORMSET, CHILDPOSSET, CHILDDEPSET: The sets of forms, parts of speech, and dependency relations of the dependents of the word;
DEPSUBCAT: Subcategorization frame: the concatenation of the dependency relations between the word and its dependents;
CHILDFORMDEPSET: The sets of pairs of forms and dependency relations of the dependents of the word;
CHILDPOSDEPSET: The sets of pairs of parts of speech and dependency relations of the dependents of the word.

Table 15.9 shows the features and their values extracted for the word *sold* in the sentence *The luxury auto maker last year sold 1,214 cars in the U.S.* These values can be computed from the CoNLL columns produced by the parsing step and shown in Table 15.8 and in Fig. 15.10.

Predicate Sense Disambiguation

This step assigns a sense number to each predicate after it has been identified. In Fig. 15.12, the classifier assigned the sense 01 to both *maker* and *sell*. As the sense number has no consistent meaning across the lemmas in Propbank and Nombank, it is necessary to train one classifier per lemma. For a given predicate, the training procedure collects all its occurrences in the training corpus and the classes are the observed sense numbers for this predicate. The classifier uses logistic regression and the same feature set as for the predicate identification step (Table 15.9).

Argument Identification

This step identifies the head words of all the arguments of a predicate. For a given predicate, we apply a binary classifier to each word in the sentence from left to right to decide if it is an argument head word or not. In Fig. 15.12, the classifier found that the argument head words of *sell* were *maker*, *year*, *cars*, and *in*. Then, using the dependency graph, we use the yield (subgraph) of these head words to derive the complete argument strings. This procedure is applied to all the predicates in the sentence and results in unlabeled propositions, where each predicate is associated with its set of arguments.

To train the classifier, we collect all the predicate–word pairs contained in each sentence of the annotated corpus and we extract feature vectors from these pairs. Given a sentence and a predicate, we mark a predicate–word pair as positive, if the word is an argument head word of the predicate, or negative otherwise. The feature set used to identify the arguments includes features extracted from the predicate and that are identical to those from the predicate identification step, the predicate lemma and its sense, for instance, sell.01, as well as features extracted from the candidate argument:

ARGFORM, ARGPOS: The form and the part of speech of the argument word;
ARGDEPREL: The dependency relation between the argument word and its head;
LEFTFORM, RIGHTFORM: The form of the leftmost, respectively rightmost, dependent of the argument word;
LEFTPOS, RIGHTPOS: The part of speech of the leftmost, respectively rightmost, dependent of the argument word;
LEFTSIBLINGFORM, RIGHTSIBLINGFORM: The form of the left, respectively right, sibling of the argument word;
LEFTSIBLINGPOS, RIGHTSIBLINGPOS: The part of speech of the left, respectively right, sibling of the argument word;
DEPRELPATH: The concatenation of dependency labels and link direction when moving from the predicate to the argument word. The dependency path from *maker* to *cars* in Fig. 15.10 is SBJ↑OBJ↓;

Table 15.10 The features used for the argument identification step in addition to those extracted from the predicate. The values correspond to the word *maker* as argument to the predicate *sell* in the sentence *The luxury auto maker last year sold 1,214 cars in the U.S.*

Feature	Value
ARGFORM	maker
ARGPOS	NN
ARGDEPREL	SBJ
LEFTFORM	The
RIGHTFORM	null
LEFTPOS	DT
RIGHTPOS	null
LEFTSIBLINGFORM	null
RIGHTSIBLINGFORM	cars
LEFTSIBLINGPOS	null
RIGHTSIBLINGPOS	NNS
DEPRELPATH	SBJ↓
POSPATH	VBD↓NN
POSITION	before

POSPATH: The concatenation of parts of speech and link direction when moving from the predicate to the argument word. The part-of-speech path from *maker* to *cars* in Fig. 15.10 is NN↑VBD↓NNS;

POSITION: The position of the argument word relative to the predicate, before, on, or after.

Table 15.10 shows the features and their values extracted for the argument *maker* with respect to the predicate *sell*. As Propbank and Nombank have different properties, training two classifiers, one for the verbs and one for the nouns, should produce better results.

The feature extraction procedure will create many more negative examples than positive ones in the training corpus. Many of these negative examples have no chance to be an argument, because of their part of speech or their path to the predicate. It is possible to balance the numbers by restricting the words in the training set that are the direct dependents of the predicate or the direct dependents of the predicate ancestors in the dependency graph.

Argument Labeling

This final step labels the argument head words of a proposition. For a given predicate and its arguments obtained from the previous step, the labeling step uses a multiclass classifier to assign the arguments with a label from A0 to A5 or those in Table 15.7. In Fig. 15.12, the classifier labels *maker* with A0, *year* with AM-TMP, *cars* with A1, and *in* with AM-LOC.

The training procedure is similar to that of the argument identification step except that the classes are the different argument labels collected from a semantically-annotated corpus. The features are similar too.

Evaluation

The evaluation of semantic dependency parsers resembles that of dependency parsing defined in Sect. 13.2. In CoNLL 2008, Surdeanu et al. (2008) divided the semantic dependencies into two types:

1. Those from the virtual ROOT word to the predicates, where the label is the predicate sense. These dependencies are created by the predicate identification and disambiguation steps.
2. The dependencies from the predicates to their arguments, where the labels are the argument labels. They are created by the argument identification and labeling steps.

The scoring procedure compares the semantic dependencies produced by the automatic parser with those of a manually-annotated test set and computes the precision and recall of the semantic links. The final CoNLL 2008 score combines these labeled precision and recall with a syntactic score and computes the harmonic mean of them. See Surdeanu et al. (2008) for the details.

Feature Selection

The features used in the classifiers are essential to the performance of a semantic parser. Contrary to intuition, a larger set of features does not necessarily lead to improved figures and may even degrade them. The features are also sensitive to the language: a feature set effective for English may yield poor results for Chinese or German.

There are two main procedures to carry out the selection of a feature set from a finite pool of N potential features: forward selection and backward elimination. Both techniques build an optimal set using a greedy search procedure, where they incrementally add, respectively remove, features:

1. A forward selection starts with an empty set and adds one feature at each iteration. At iteration n, the set has n features, and the search procedure generates $N - n$ successor sets of $n + 1$ features. The procedure evaluates these successor sets and keeps the one that yields the best performance. The search stops when there is no improvement in the performance.
2. A backward elimination starts with a feature set consisting of the complete pool of N features and removes one feature at each iteration. At iteration n of the procedure, the feature set consists of $N - n$ features. Its successors are the $N - n$ sets generated by removing one feature. The backward elimination evaluates these sets and keeps the one that leads to the largest improvement.

The feature selection is carried out for each step of the processing pipeline shown in Fig. 15.12 and produces one different feature set per step in the pipeline. To avoid a possible overfit, classification models should be trained and evaluated using a crossvalidation. Björkelund et al. (2009) give examples of feature sets for seven languages obtained using a forward selection.

Table 15.11 A dialogue with the EVAR system

Turn	Utterance
EVAR	*Guten Tag, hier ist die automatische Intercity-Auskunft.*
	'Hello, this is the automatic Intercity information system.'
User	*Ich will nach Hamburg fahren*
	'I want to go to Hamburg'
EVAR	*Sie wollen nach Hamburg fahren?*
	'You want to go to Hamburg?'
	Wo wollen Sie abfahren?
	'Where do you want to start?'
User	*In Erlangen*
EVAR	*Sie wollen von Erlangen abfahren?*
	'You want to start from Erlangen?'
	An welchem Tag?
	'Which day?'
User	*Morgen früh*
	'Tomorrow morning'
EVAR	*Sie wollen am Donnerstag den 26. Juni zwischen 6 und 12 Uhr fahren?*
	'You want to travel on Thursday 26th of June between 6 and 12?'
User	*Genau*
	'Yes'
EVAR	*Ich habe mehrere Verbindungen gefunden.*
	'I found connections.'
	Nach Hamburg HBF fahren sie ab Erlangen um 11 Uhr 1, Ankunft in Hamburg HBF...
	'To Hamburg, you can start from Erlangen at 11.01, arrival in Hamburg...'
	Soll ich die Verbindungen wiederholen?
	'Shall I repeat?'
User	*Nein Danke*
EVAR	*Möchten Sie weitere Auskünfte?*
	'Would you like more information?'
User	*Nein*
EVAR	*Auf wiederhören*
	'Good bye'

15.8 An Example of Case Grammar Application: EVAR

The EVAR application (Mast et al. 1994) is a spoken dialogue system that answers inquiries about train connections in Germany. EVAR uses a dedicated ontology to represent the domain and a case grammar to process the utterances. It is able to process exchanges like the one in Table 15.11.

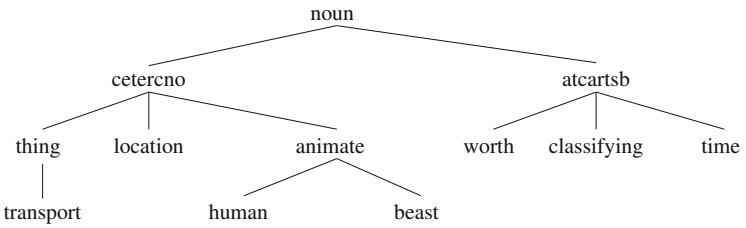


Fig. 15.13 Semantic classes in the EVAR application (After Mast et al. (1994))

15.8.1 EVAR's Ontology and Syntactic Classes

EVAR organizes nouns as a classical ontological tree restricted to the railway domain in Germany (Fig. 15.13). For instance, *train* is linked to "transport," *Hamburg* to "location," etc.

15.8.2 Cases in EVAR

EVAR uses a partial parser to detect syntactic groups, notably noun groups, prepositional groups, infinitive groups, verb groups, and time and date expressions. It represents utterances with a case grammar tied to the train domain and uses the ontology in Fig. 15.13. The case system is relatively detailed – it is said to be fine grained – and consists of about 30 cases associated to verbs and also to nouns and adjectives. Table 15.12 shows some examples of case frames together with their constraints.

Sagerer (1990) gives the full description of the semantic cases related to EVAR.

15.9 Further Reading

Although it may not serve for immediate applications, Saussure's *Cours de linguistique générale* (1916) offers a fundamental background introduction on links between language and thought. Also of interest is Hjelmslev's *Prolegomena to a Theory of Language* (1943), which provides a complement to Saussure's views. A good introduction to the classical texts is the historical presentation on linguistics by Harris and Taylor (1997).

Electric Words by Wilks et al. (1996) provides an account on semantics that focuses on computerized dictionaries. It contains many mistakes, however. Mel'čuk et al. (1995) propose a detailed dictionary model for French. Fellbaum (1998) gives an in-depth description of WordNet and its design. The WordNet lexical

Table 15.12 Some verbs and nouns with their cases in the EVAR system (After Mast et al. (1994))

Word senses	Examples and cases
Fahren1.1	*Der Zug fährt von Hamburg nach München*
	'The train is going from Hamburg to Munich'
	Instrument: noun group (nominative), Transport, obligatory
	Source: prepositional group (Origin), Location, optional
	Goal: prepositional group (Direction), Location, optional
Fahren1.2	*Ich fahre mit dem Zug von Hamburg nach München*
	'I am going by train from Hamburg to Munich'
	Agent: noun group (nominative), Animate, obligatory
	Instrument: prepositional group (prep=mit), Transport, optional
	Source: prepositional group (Origin), Location, optional
	Goal: prepositional group (Direction), Location, optional
Abfahrt1.1	*Die Abfahrt des Zuges von Hamburg nach München*
	'The departure of the train at Hamburg for Munich'
	Object: noun group (genitive), Transport, optional
	Location: prepositional group (Place), Location, optional
	Time: prepositional group (Moment), Time, optional
Verbindung1.5	*Eine Verbindung von Hamburg nach München*
	'A connection from Hamburg to Munich'
	Source: prepositional group (Origin), Location, optional
	Goal: prepositional group (Direction), Location, optional

database is regularly updated and its content is available for download from http://wordnet.princeton.edu/. Pustejovsky (1995) presents an alternate viewpoint on lexical structure. Dutoit (1992) describes another model that governed the implementation of Dicologique, a lexical database in French of size comparable to WordNet.

Literature on word sense disambiguation is countless. The reader can find a starting point in a dedicated special issue of *Computational Linguistics*, especially in its *Introduction* (Ide and Véronis 1998), which lists more than 300 references! Word sense disambiguation has made considerable progress recently. The SENSEVAL workshops, renamed SEMEVAL from 2007, benchmark competing systems for a variety of languages. Proceedings are available from the ACL Anthology (http://www.aclweb.org/anthology/).

In a classical text, Fillmore (1968) gives the rationale behind case grammars. Jackendoff (1990) gives another detailed description of cases for English. Later, Fillmore started the FrameNet project that itemizes the frame elements of all the verbs. Its description and content is available for download from http://framenet.icsi.berkeley.edu/. The Proposition bank or Propbank is a similar project aimed at annotating the Penn Treebank with semantic data (Kingsbury et al. 2002). The Unified Verb Index is an attempt to merge all the English verb entries of three different lexicons, Verbnet, Propbank, and Framenet, in a single index: http://verbs.colorado.edu/verb-index/.

Automatic role labeling using statistical techniques has received considerable interest recently and was the theme of four conferences on Computational Natural Language Learning (CoNLL 2004, CoNLL 2005, CoNLL 2008, and CoNLL 2009). Annotated data, descriptions, and performance of the competing systems are available from the conference web pages: http://www.lsi.upc.edu/~srlconll/st04/st04.html, http://www.lsi.upc.edu/~srlconll/st05/st05.html, http://barcelona.research.yahoo.net/dokuwiki/doku.php?id=conll 2008:start, and http://ufal.mff.cuni.cz/conll2009-st/.

Exercises

15.1. Implement the complete semantic net of Fig. 15.3.

15.2. Implement a graph search that finds entities linked by properties using inheritance, and test it using the `eat/2` relation.

15.3. Annotate each word of the following sentences with their possible senses:

The waiter brought the starter to the customers.
Le serveur a apporté l'entrée aux clients
Der Ober hat die Vorspeise zum Kunden gebracht.

You may use any dictionary.

15.4. Write verb syntactical models corresponding to senses of *order* in Table 15.1.

15.5. Write selectional restrictions corresponding to senses of *order* in Table 15.1.

15.6. Take a dozen or so words and, using their definitions, build the corresponding ontological tree.

15.7. According to WordNet, *bring* entails *come, come up* (move toward, travel toward something or somebody or approach something or somebody). Classify this type of entailment as coextensiveness, proper inclusion, backward presupposition, or cause.

15.8. Implement the word sense disambiguation algorithm outlined in Sect. 15.5.3.

- Write a Prolog program that produces all sense sequences of a given sentence. Implement the lexical database representing possible senses of *patron*, *ordered*, and *meal*, and test the program with *The patron ordered the meal*.
- Find topics associated with senses of words *patron*, *order*, and *meal*. Set these topics under the form of Prolog facts.
- Write a Prolog program that collects all the topics associated with a sense sequence.
- What is the main condition for the algorithm to produce good results?

15.9. Disambiguate by hand the senses of words in the sentence: *the patron ordered the meal* using word definitions and the algorithm of Sect. 15.5.6. You may use any dictionary.

15.10. Program the algorithm of Exercise 15.9.

Chapter 16
Discourse

16.1 Introduction

The grammatical concepts we have seen so far apply mostly to isolated words, phrases, or sentences. Texts and conversations, either full or partial, are out of their scope. Yet to us, human readers, writers, and speakers, language goes beyond the simple sentence. It is now time to describe models and processing techniques to deal with a succession of sentences. Although analyzing texts or conversations often requires syntactic and semantic treatments, it goes further. In this chapter, we shall make an excursion to the discourse side, that is, paragraphs, texts, and documents. In the next chapter, we shall consider dialogue, that is, a spoken or written interaction between a user and a machine.

Most basically, a discourse is made of **referring expressions**, i.e., words or phrases that refer to real – or possibly imaginary – things: the **discourse entities** or **discourse referents**. A first objective of discourse processing techniques is then to identify and track sets of referring expressions – phrases or words – along with sentences and to relate them to entities – real-world objects. A symmetrical operation is to associate entities with their **mentions** in a text.

A discourse normally links the entities together, via their mentions, to address topics, issues throughout the sentences, paragraphs, chapters such as, for instance, the quality of food in restaurants, the life of hedgehogs and toads, and so on. At a local level, i.e., within a single sentence, grammatical functions such as the subject, the verb, and the object provide a model of relations between entities. A model of discourse should extend and elaborate relations that apply not to an isolated sentence but to a sequence and hence to the entities that this sequence of sentences covers.

Models of discourse structures are still a subject of controversy. As for semantics, discourse has spurred many theories, and it seems relatively far off to produce a synthesis of them. In consequence, we will merely adopt a bottom-up and pragmatic approach. We will start from what can be a shallow-level processing of discourse and application examples; we will then introduce theories, namely centering, rhetoric, and temporal organization, which provide hints for a discourse structure.

P.M. Nugues, *Language Processing with Perl and Prolog*, Cognitive Technologies,
DOI 10.1007/978-3-642-41464-0_16, © Springer-Verlag Berlin Heidelberg 2014

16.2 Discourse: A Minimalist Definition

16.2.1 A Description of Discourse

Intuitively what defines a discourse, and what differentiates it from unstructured pieces of text, is its coherence. A discourse is a set of more or less explicit topics addressed in a sequence of sentences: what the discourse is about at a given time. Of course, there can be digressions, parentheses, interruptions, etc., but these are understood as exceptions in the flow of a normal discourse. Distinctive qualities of a discourse are clarity, expressiveness, or articulation, which all relate to the ease of identification of discourse topics and their logical treatment. Discourse coherence ideally takes the shape of a succession of stable subjects (or contexts) that are chained rationally along with the flow of sentences.

More formally, we describe a discourse as a sequence of utterances or segments, $S_1, S_2, S_3, \ldots, S_n$, so that each of these segments is mapped onto a stationary context. Segments are related to sentences, but they are not equivalent. A segment can span one or more sentences, and conversely a sentence can also contain several segments. Segments can be produced by a unique source, which is the case in most texts, or by more interacting participants, in the case of a dialogue.

16.2.2 Discourse Entities

Discourse entities – or discourse referents – are the real, abstract, or imaginary objects introduced by the discourse. Usually they are not directly accessible to a language processing system because it would require sensors to "see" or "feel" them. In a language like Prolog, discourse entities are represented as a set of facts stored in a database.

Referring expressions are mentions of the discourse entities along with the text. Table 16.1 shows entities and references of sentences adapted from Suri and McCoy (1994):

1. *Susan drives a Ferrari*
2. *She drives too fast*
3. *Lyn races her on weekends*
4. *She often beats her*
5. *She wins a lot of trophies*

Discourse entities are normally stable – constant – over a segment, and we can use them to delimit a segment's boundaries. That is, once we have identified the entities, we can delimit the segment boundaries. Let us come back to our example. There are two sets of relatively stable entities that we can relate to two segments. The first one is about Susan and her car. It consists of sentences 1 and 2. The second one is about Susan and Lyn, and it extends from 3 to 6 (Table 16.2).

Table 16.1 Discourse entities and mentions (or referring expressions)

Mentions (or referring expressions)	Discourse entities (or referents)	Logic properties
Susan, she, her	`'Susan'`	`'Susan'`
Lyn, she	`'Lyn'`	`'Lyn'`
A Ferrari	`X`	`ferrari(X)`
A lot of trophies	`E`	`E` $\subset$ `{X | trophy(X)}`

Table 16.2 Context segmentation

Contexts	Sentences	Entities
C1	1. *Susan drives a Ferrari*	Susan, Ferrari
	2. *She drives too fast*	
C2	3. *Lyn races her on weekends*	Lyn, Susan, trophies
	4. *She often beats her*	
	5. *She wins a lot of trophies*	

16.3 References: An Application-Oriented View

As a starting point of discourse processing, we will focus on referring expressions, i.e., words or phrases that correspond to the discourse entities. This treatment can be done fairly independently without any comprehensive treatment of the text. In addition, the identification of discourse entities is interesting in itself and has an industrial significance in applications such as information extraction.

In this section, we will take examples from the Message Understanding Conferences (MUCs) that we already saw in Chap. 10. We will learn how to track the entities along with sentences and detect sets of phrases or words that refer to the same thing in a sentence, a paragraph, or a text.

16.3.1 References and Noun Phrases

In MUC, information extraction consists in converting a text under the form of a file card. Cards are predefined templates whose entries are formatted tabular slots that represent the information to be extracted: persons, events, or things. For each text, information extraction systems have to generate a corresponding card whose slots are filled with the appropriate entities.

Detecting – generating – the entities is a fundamental step of information extraction; a system could not fill the templates properly otherwise. To carry it out, the basic idea is that references to real-world objects are equivalent to noun groups or noun phrases of the text. So, detecting the entities comes down to recognizing the nominal expressions.

Table 16.3 References in the sentence: *Garcia Alvarado, 56, was killed when a bomb placed by urban guerrillas on his vehicle exploded as it came to a halt at an intersection in downtown San Salvador*

Entities	Noun groups
Entity 1	*Garcia Alvarado*
Entity 2	*a bomb*
Entity 3	*urban guerrillas*
Entity 4	*his vehicle*
Entity 5	*it*
Entity 6	*a halt*
Entity 7	*an intersection*
Entity 8	*downtown*
Entity 9	*San Salvador*

To realize in concrete terms what this means, let us take an example from Hobbs et al. (1997) and identify the entities. We just have to bracket the noun groups and to assign them with a number that we increment with each new group:

> [$_{entity1}$ Garcia Alvarado], 56, was killed when [$_{entity2}$ a bomb] placed by [$_{entity3}$ urban guerrillas] on [$_{entity4}$ his vehicle] exploded as [$_{entity5}$ it] came to [$_{entity6}$ a halt] at [$_{entity7}$ an intersection] in [$_{entity8}$ downtown] [$_{entity9}$ San Salvador].

We have detected nine nominal expressions and hence nine candidates to be references that we represent in Table 16.3.

Typical discourse analyzers integrate modules into an architecture that they apply on each sentence. Depending on applications, they use a full-fledged parser or a combination of part-of-speech tagger, group detector, semantic role identifier, or ontological classifier. Here, we could have easily created these entities automatically with the help of a noun group detector. A few lines more of Prolog to our noun group detector (Chap. 10) would have numbered and added each noun group to an entity database.

16.3.2 Names and Named Entities

In unrestricted texts, in addition to common nouns, many references correspond to names (or proper nouns). Names refer *inter alia* to:

Persons: Mrs. Smith, François Arouet, Dottore Graziani, Wolfgang A. Mozart, H.C. Andersen, Sammy Davis, Jr.

Companies or organizations: IBM Corp., Fiat SpA, BT Limited, Banque Nationale de Paris, Siemens AG, United Nations, Nations unies

Countries, nations, or provinces: England, France, Deutschland, Romagna, Vlaanderen

Cities or geographical places: Paris, The Hague, Berlin, le Mont Blanc, la Città del Vaticano, the English Channel, la Manche, der Rhein.

While entities coincide with unique things from the real world, named entities correspond to persons, locations, organizations, trade marks, etc., that can we can identify in a unique way by their name. In addition, although numerical expressions and dates are sometimes called named entities, the most common acceptation is that they are restricted to proper nouns (names).

16.3.3 Finding Names – Proper Nouns

Name recognition is central to a proper reference processing and uses the techniques we described in Chap. 10. They are similar to noun group detection, or chunking, and can be carried out with rules or statistical methods. Name recognition frequently uses a dedicated index of names: a gazetteer. Such name gazetteers can sometimes be downloaded from the Internet; however, for many applications, they have to be compiled manually or bought from specialized companies.

Name indexes are rarely complete or up-to-date. Peoples' names particularly are tricky and may sometimes be confused with common names. The same can be said of names of companies, which are created every day, and those of countries, which appear and disappear with revolutions and wars. If we admit that there will be names missing in the database, we have to design the rules or the features of a statistical system so that they cope with them.

Rules

Guessing a person's name is often done through titles and capitalization. Here are a few rules of thumb to reflect this:

- A first name, a possible initial, and a surname that is two strings of characters with a capitalized first letter followed by lowercase letters:
  Robert Merryhill, Brigitte Joyard, Max Hübnisch
  A possible enhancement is to try to match the first string to common first names.
- A title, possible first names or initials and a surname where common titles have to be itemized:
  Sir Robert Merryhill, Dr. B. K. Joyard, Herr Hübnisch
- A person's name and a suffix:
  R. Merryhill Sr., Louis XXII, Herr Hübnisch d. med.

We can implement a rule-based name recognition system with local DCG rules and a word spotting program to match titles and first names as in Sect. 10.4. We can test the character case with a short piece of Prolog code or regular expressions.

Table 16.4 Features used for named-entity recognition by Zhang and Johnson (2003). The features are extracted from the tokens in a window of three tokens centered at the current token unless specified

Features
Tokens in a window of five tokens centered at the current token: $w_{i-2}, w_{i-1}, w_i, w_{i+1}, w_{i+2}$
Initial capitalization of the tokens in a window of five tokens
Booleans describing the word type: initial capitalization, all capitalization, all digitals, or digitals containing punctuations
Parts of speech of the tokens
Set of chunk tags of the chunk at the current token
Token prefix and suffix
Previous predicted NER tags
Conjunction of the previous NER tag and the current token
Presence in gazetteers of locations, persons, organizations, and other named entities

Machine-Learning Techniques

Named-entity recognition (NER) with machine learning consists usually of a pipeline of classifiers that carry out tokenization, POS tagging, and possibly lemmatization and chunking. Then, the NER step is roughly identical to chunking that we saw in Sect. 10.7, except that it uses a different feature set.

Table 16.4 outlines a relatively simple feature set used by Zhang and Johnson (2003) in CoNLL-2003 (Tjong Kim Sang and De Meulder 2003). It represents a good starting point to build a NER classifier.

16.3.4 Disambiguation of Named Entities

In a document or a collection of documents, writers may use different names to refer to a same named entity. Think of *Ferdinand de Saussure*, for instance, and the variants *Saussure* or *de Saussure*. Conversely, a name like *Cambridge* may refer to different cities: Cambridge, England; Cambridge, Massachusetts; or Cambridge, Ontario.

To avoid confusion, a solution is to assign named entities with a unique identifier. If the entities are known persons, cities, countries, or organizations, etc., DBpedia, Yago, or Wikidata are widely used nomenclatures based on the articles in Wikipedia (Sect. 14.11.3). They provide unique identifiers to notable entities, something like a Social Security number for US residents. Geonames is a similar comprehensive register for geographical names.

DBpedia assigns *Ferdinand de Saussure* with the identifier (URI): http://dbpedia.org/resource/Ferdinand_de_Saussure, while Wikidata uses http://www.wikidata.org/wiki/Q13230. Such unique names can then be used across programs and applications. In addition, when the properties and the values of

the corresponding entities are encoded using the RDF format, they can be accessed and linked across different repositories.

As we saw, strings mentioning an entity may be ambiguous and refer to two or more persons, places, things, etc. The name *Saussure* matches at least 11 different persons in the English version of Wikipedia. Henri de Saussure, father of Ferdinand, and René de Saussure, his brother, are two of them. Henri has the http://dbpedia.org/resource/Henri_Louis_Frédéric_de_Saussure identifier in DBpedia and http://www.wikidata.org/wiki/Q123776 in Wikidata. Finding the correct entity behind a name is referred to as named-entity disambiguation.

Bunescu and Paşca (2006) proposed a supervised classification method to disambiguate named entities based on the vector space model and ranking support vector machines. We introduced the vector space model in Sect. 5.12 and support vector machines in Sect. 4.8.

Given an ambiguous named-entity mention in a document and entity candidates in the form of Wikipedia articles, Bunescu and Paşca (2006) represent each mention–entity pair with a feature vector. This means that if the system finds 11 entity candidates for the mention, they build 11 vectors. Each vector consists of two components:

1. The first dimension is the cosine similarity between the document, restricted to a window of 55 words centered on the mention, and the Wikipedia page corresponding to the entity. This similarity is the dot product of the bag-of-word vectors that uses $tf \times idf$ as term weight.
2. The remaining dimensions are occupied by copies of the bag-of-word vector representing the document, also restricted to a window of 55 words, with as many copies as there are categories in Wikipedia, History, Science, Politics, etc. If the vocabulary used in Wikipedia has V unique words and there are C categories, the vector dimension will then be of $V \times C$. For each category, the vector is copied as is if the entity belongs to the category or copied and set to 0 if not.

Bunescu and Paşca (2006) built a training set of positive and negative examples from the Wikipedia text, where they used the link markup to create a gold annotation. A link in Wikipedia is encoded with two parts separated by a vertical bar: the link itself, the page we go when we click on the link, and the text that shows in the page. The Wikipedia code:

```
[[Ferdinand de Saussure|Saussure]]
```

shows the word *Saussure* and takes us to the *Ferdinand de Saussure* page. It reflects the association between the mention *Saussure* and the entity Ferdinand de Saussure in a document.

Relying on the work of the Wikipedia editors, these links enable us to extract mention–entity pairs in their context: a positive pair here: (Saussure, Ferdinand de Saussure), and negative ones: (Saussure, Henri de Saussure), (Saussure, René de Saussure), etc.

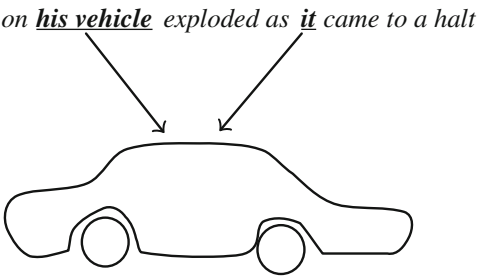


Fig. 16.1 Coreferencing an entity with a noun group and a pronoun

16.4 Coreference

16.4.1 Anaphora

In the example shown in Table 16.3, we numbered eight objects corresponding to noun groups and one corresponding to the pronoun *it*. Such a pronoun is generally related to a previous expression in the text and depends on this expression to be interpreted. Here, the reader can easily guess that *it* and the noun group *his vehicle* designate the same entity. This means that entities 5 and 4 in Table 16.3 are equal and that nominal expressions *his vehicle* and *it* **corefer** to a same thing (Fig. 16.1).

The pair of expressions *his vehicle* and *it* form an **anaphora** where the first reference to the object – *his vehicle* – is the **antecedent** and subsequent references – here *it* – are **anaphors**. Antecedent and anaphors are then a set of references to a same entity in a text. The antecedent acts as the semantic source of the set and enables the understanding of the anaphors (Tesnière 1966).

Third-person and relative pronouns (*he/she/it/who*) are typical examples of anaphors. In addition to them, anaphora uses demonstrative pronouns such as *this*/*that* in *He did that*, or location adverbs such as *here*/*there* in *He was there*. Demonstrative pronouns (or adjectives) can be used as determiners as in *this vehicle*. The possessive pronouns (or adjectives) are similar and also denote an anaphora as *his* in *his vehicle* that is tied to the vehicle's possessor: *Garcia Alvarado*.

While normally anaphors have their antecedent before they occur, there are sometimes examples of forward references or **cataphora**. For example, in the sentence:

> I just wanted to touch **it**, this stupid animal.

It refers to the *stupid animal*. It has been shown that in most structured discourses, cataphoras occur in the same sentence. However, this is not always the case, and sometimes the referent is never mentioned, either because it is obvious given the context, because it is unknown, or for some other reasons:

> They have stolen my bicycle.

16.4.2 Solving Coreferences in an Example

Although we had no difficulty recognizing the identity of the two expressions, *his vehicle* and *it* in the example above, coreference resolution is not as straightforward as it may appear at a first sight. Let us come back to our example in Table 16.3 to show this, and let us make our method explicit to outline an algorithm. We will first admit that coreferences of a pronoun are always located before the pronoun occurs in the text. Then, in the example, in addition to *his vehicle* (entity 4), pronoun *it* (entity 5) has four possible candidates: *it* could be *Garcia Alvarado* (entity 1), *a bomb* (entity 2), or *urban guerrillas* (entity 3).

We can rule out entities 1 and 3 from the coreference set because they do not match the pronoun's number or gender. If entity 5 had been *Garcia Alvarado* – a man – the pronoun would have been *he*, and if it had been *urban guerrillas* – a plural – the pronoun would have been *they*. The noun group *A bomb* is more difficult to discard. We do not retain it because of a semantic incompatibility. Selectional restrictions of the verb *came* likely require that its subject is a vehicle or a person.

We saw examples of anaphora where a same entity is specified by a noun and a pronoun. Pairs of references can also consist of nouns or noun groups. They can simply be a repetition of identical expressions, (*the vehicle*, *the vehicle*). Sometimes there might be a different determiner, a different denomination, synonyms, or aliases to refer to a same thing. For instance, in an economic wire, we can first have *Bayerische Motoren Werke*, then *BMW*, and finally *the German automaker*.

Coreference is a far-reaching concept that can prove very complex. The definition of anaphora may also vary: most authors restrain anaphors to be pronouns and certain types of adverbs. Others extend it to noun phrases, either definite or not. In the rest of the text, we will make no distinction, and we will define coreference resolution or coreference recognition as the retrieval of sets of references to identical entities in a text – what we have just come to do. We will also keep the terms antecedent and anaphor to refer to the first and second term of a coreferring pair, even if the anaphor is not a pronoun.

16.4.3 The MUC Coreference Annotation

Before we explain general methods to solve coreferences, let us first examine an annotation scheme proposed in the sixth and seventh Message Understanding Conferences (MUC-6 and MUC-7) to tag them. While there are various mark-up models, this one, based on XML tags, is widely public and can be considered as a standard. In addition, as the MUC's final objective is to extract information, these tags have an application interest.

The annotation of references and coreferences in a text consists first of identifying of the referring expressions and then assigning a unique label to expressions referring to a same entity. Hirschman and Chinchor (1997) proposed to annotate

nominal expressions, that is nouns, noun phrases, and pronouns, here considered as referring expressions, and their antecedents, with the XML-defined COREF element. COREF has five possible attributes: ID, REF, TYPE, MIN, and STAT.

ID is an arbitrary integer that assigns a unique number to each nominal expression of the text. REF is an optional integer that links a nominal expression to a coreferring antecedent. REF value is then the ID of its antecedent. From Hirschman and Chinchor's annotated examples, the text

```
<COREF ID="100">Lawson Mardon Group Ltd.</COREF> said <COREF ID="101"
TYPE="IDENT" REF="100">it</COREF>
```

indicates that *Lawson Mardon Group Ltd.* and *it* are assigned respectively with ID 100 and 101, and that *it* refers to the same entity as *Lawson Mardon Group Ltd.* through REF="100".

In the MUC competitions, coreference is defined as symmetric and transitive, that is, if A is coreferential with B, the reverse is also true. And if A is coreferential with B, and B is coreferential with C, then A is coreferential with C. Such a coreference set then forms an equivalence class called a **coreference chain**. This is stated with the TYPE attribute that specifies the link between the anaphor and its antecedent: "IDENT" is the only possible value of the attribute, and it indicates that coreferences are identical. One may imagine other types of coreference such as part, subset, etc.

Other attributes are MIN and STAT. Some denominations may have a variable length and yet refer to the same entity, such as *Queen Elizabeth of England* and *Queen Elizabeth*. In a text where the denomination appears in full, a coreference analyzer could bracket both. The COREF tag MIN indicates the minimum valid string. From Hirschman and Chinchor's guidelines,

```
<COREF ID="100" MIN="Haden MacLellan PLC">Haden MacLellan PLC of
Surrey, England</COREF> ... <COREF ID="101" TYPE= "IDENT"
REF="100">Haden MacLellan</COREF>
```

indicates that *Haden MacLellan PLC of Surrey, England* and *Haden MacLellan PLC* are both valid bracketing.

Finally, STAT ("status") means that the annotation is optional. It is used when coreference is tricky or doubtful. The only value for this attribute is OPT ("optional"). From Hirschman and Chinchor's guidelines,

```
<COREF ID="102" MIN="Board of Education">Our Board of Educa-
tion</COREF> budget is just too high, the Mayor said. <COREF ID="103"
STAT="OPT" TYPE="IDENT" REF="102">Livingston Street</COREF> has lost
control.
```

indicates that *Board of Education* and *Livingston Street* refers to the same entity, but that it can bewilder the reader and the annotation is left optional.

16.4.4 The CoNLL Coreference Annotation

Format

In 2011 and 2012, the CoNLL shared task assessed coreference resolution (Pradhan et al. 2011). The CoNLL 2011 corpus consists of a set of documents in English, where the coreference chains are marked in each document, while CoNLL 2012 extends the languages to Chinese and Arabic. The CoNLL format contains syntactic and semantic annotations. It uses columns to describe the index, words, lemmas, parts of speech, parse trees, semantic types, and predicate–argument structures. The coreference chains are given in the last column.

Table 16.5 shows the simplified annotation of the short text:

> "Vandenberg and Rayburn are heroes of mine," Mr. Boren says, referring as well to Sam Rayburn, the Democratic House speaker who cooperated with President Eisenhower. "They allowed this country to be credible. I really want to see that happen again."

where the sentences are separated by a blank line.

The CoNLL coreference format is similar to other CoNLL formats that we saw in other chapters of this book. The main differences are:

- The first column contains a document identifier, which corresponds to the scope of coreference solving. While coreferences could be annotated across multiple documents, this not the case here.
- The parse bit column contains the parse trees of the sentences. It uses constituents and the Penn Treebank annotation. For each word in the table, the parse bit starts with the first open parenthesis in the parse tree and consists of the phrase symbols and parentheses up to the word. The word and part-of-speech leaves are replaced with a star (*) and followed by possible parentheses closing the constituent. Using this column, we can extract all the noun phrases of a document, and thus the mentions.
- The goal of coreference solving is to predict the coreference chains in the last column from the other columns given as input.

Mentions and Chains

The entity or event mentions in the CoNLL corpus are described with a parenthesis structure, where each entity is assigned a number unique in the document.

Representing a coreference chain as:

$$CorefChain(x) = \{Mention_1^x, Mention_2^x, \ldots, Mention_n^x\},$$

Table 16.5 Simplified annotation of three sentences in the CoNLL 2011 corpus (After Pradhan et al. (2011))

Document	Index	Word	POS	Parse bit	Type	Chain
wsj_0771	0	“	“	(TOP(S(S*	*	–
wsj_0771	1	Vandenberg	NNP	(NP*	(PERSON)	(8|(0)
wsj_0771	2	and	CC	*	*	–
wsj_0771	3	Rayburn	NNP	*)	(PERSON)	(23)|8)
wsj_0771	4	are	VBP	(VP*	*	–
wsj_0771	5	heroes	NNS	(NP(NP*)	*	–
wsj_0771	6	of	IN	(PP*	*	–
wsj_0771	7	mine	NN	(NP*))))	*	(15)
wsj_0771	8	,	,	*	*	–
wsj_0771	9	”	”	*)	*	–
wsj_0771	10	Mr.	NNP	(NP*	*	(15
wsj_0771	11	Boren	NNP	*)	(PERSON)	15)
wsj_0771	12	says	VBZ	(VP*	*	–
wsj_0771	13	,	,	*	*	–
wsj_0771	14	referring	VBG	(S(VP*	*	–
wsj_0771	15	as	RB	(ADVP*	*	–
wsj_0771	16	well	RB	*)	*	–
wsj_0771	17	to	IN	(PP*	*	–
wsj_0771	18	Sam	NNP	(NP(NP*	(PERSON*	(23
wsj_0771	19	Rayburn	NNP	*)	*)	–
wsj_0771	20	,	,	*	*	–
wsj_0771	21	the	DT	(NP(NP*	*	–
wsj_0771	22	Democratic	JJ	*	(NORP)	–
wsj_0771	23	House	NNP	*	(ORG)	–
wsj_0771	24	speaker	NN	*)	*	–
wsj_0771	25	who	WP	(SBAR(WHNP*)	*	–
wsj_0771	26	cooperated	VBD	(S(VP*	*	–
wsj_0771	27	with	IN	(PP*	*	–
wsj_0771	28	President	NNP	(NP*	*	–
wsj_0771	29	Eisenhower	NNP	*))))))))))	(PERSON)	23)
wsj_0771	30	.	.	*))	*	–
wsj_0771	0	“	“	(TOP(S*	*	–
wsj_0771	1	They	PRP	(NP*)	*	(8)
wsj_0771	2	allowed	VBD	(VP*	*	–
wsj_0771	3	this	DT	(S(NP*	*	(6
wsj_0771	4	country	NN	*)	*	6)
wsj_0771	5	to	TO	(VP*	*	–
wsj_0771	6	be	VB	(VP*	*	(16)
wsj_0771	7	credible	JJ	(ADJP*)))))	*	–
wsj_0771	8	.	.	*))	*	–
wsj_0771	0	I	PRP	(TOP(S(NP*)	*	(15)
wsj_0771	1	really	RB	(ADVP*)	*	–
wsj_0771	2	want	VBP	(VP*	*	–
wsj_0771	3	to	TO	(S(VP*	*	–

(continued)

Table 16.5 (continued)

Document	Index	Word	POS	Parse bit	Type	Chain
wsj_0771	4	see	VB	(VP*	*	–
wsj_0771	5	that	IN	(SBAR*	*	(16)
wsj_0771	6	happen	VBP	(S(VP*	*	–
wsj_0771	7	again	RB	(ADVP*))))))))	*	–
wsj_0771	8	.	.	*	*	–
wsj_0771	9	”	”	*))	*	–

where x denotes an entity or an event, and $Mention_i^x$, its ith mention of it in a document, the coreference chains in Table 16.5 are:

$$
\begin{aligned}
CorefChain(0) &= \{Vandenberg\},\\
CorefChain(6) &= \{this\ country\},\\
CorefChain(8) &= \{Vandenberg\ and\ Rayburn, They\},\\
CorefChain(15) &= \{mine, Mr.\ Boren, I\},\\
CorefChain(16) &= \{be, that\},\\
CorefChain(23) &= \{Rayburn, Sam\ Rayburn\backslash,\ the\ Democratic\ House\ speaker\\
&\qquad who\ cooperated\ with\ President\ Eisenhower\},
\end{aligned}
$$

To carry out the annotation, the annotators extracted all the noun phrases and pronouns and created the coreference chains from them. The mentions use the maximal span of a noun phrase, i.e., given a head noun, the NP constituent that includes all its direct and indirect dependents. In the example above, both noun phrases and mentions:

Sam Rayburn

and

Sam Rayburn, the Democratic House speaker who cooperated with President Eisenhower

refer to the same person. The annotators kept only the latter phrase as it spans all the dependents. They did not mark the singletons: entities or events mentioned only once in a text.

The annotators could also include verbs as part of coreference chains, although they accounted for less than 2 % of the mentions in the corpus. The next two sentences give an example of a verb–noun coreference (Pradhan et al. 2011):

Sales of passenger cars **grew** 22%. **The strong growth** followed year-to-year increases.

In Table 16.5, *be* is marked as a mention and corefers with *that*.

16.5 References: A More Formal View

16.5.1 Generating Discourse Entities: The Existential Quantifier

In Chap. 14, we introduced a logical notation to represent nominal expressions that differs from that of the previous section. If we take the formal semantics viewpoint, a sentence such as:

A patron ordered a meal.

exposes two new terms: *a patron* and *a meal*. These entities are tied to indefinite noun phrases and hence to logical forms headed by the existential quantifier ∃:

$$\exists x, patron(x)$$
$$\exists y, meal(y)$$

A discourse interpretation program should reflect them in a Prolog database and augment the database with the corresponding semantic facts:

```
patron(patron#3).
meal(meal#15).
```

We generate the entities by creating new constants – new atoms – making sure that they have a unique name, here `patron#3` or `meal#15`. Then, we can add them in the database under the form of facts using the `asserta/1` built-in predicate.

New entities are only a part of the whole logical set because the complete semantic representation of the sentence is:

```
a(X, patron(X), a(Y, meal(Y), ordered(X, Y)))
```

To be consistent with this representation, we must also add the predicate `ordered(Subject, Object)` to link the two new entities. We carry this out by asserting a last fact:

```
ordered(patron#3, meal#15).
```

16.5.2 Retrieving Discourse Entities: Definite Descriptions

While indefinite noun phrases introduce new entities, definite ones usually refer to entities created previously. A possible subsequent sentence in the discourse could be:

The patron ate the meal,

which should not create new entities. This simply declares that the patron already mentioned ate the meal he ordered. Such definite noun phrases are then anaphors.

The logic interpretation of definite descriptions usually translates as:

$$\exists!x, patron(x)$$
$$\exists!y, meal(y)$$

where properties are quantified with $\exists!$ meaning that x and y are unique. To reflect this in the Prolog database, we could identify x and y among the entities previously created and then assert the new fact:

```
ate(patron#3, meal#15).
```

An alternate processing of the `ate/2` relation – and probably a more alert one – is to first create new atoms, that is, new names:

```
patron(patron#5).
meal(meal#17).
```

to link them with `ate/2`:

```
ate(patron#5, meal#17).
```

and to assert later that some names are identical:

```
equals(patron#3, patron#5).
equals(meal#15, meal#17).
```

This method is precisely the coreference recognition that we described previously. Besides, proceeding in two steps enables a division of work. While a first task generates all potential entities, a second one resolves coreferences using techniques that we review in Sect. 16.6.

16.5.3 Generating Discourse Entities: The Universal Quantifier

We saw that determiners can also correspond to the universal quantifier $\forall$. An example of such a sentence is:

Every patron ordered a meal.

Its corresponding logic representation is:

$$\forall x, patron(x) \Rightarrow \exists y, meal(y), ordered(x, y)$$

Table 16.6 A Skolem function

X	Y	**Skolem function values**
pierre	cassoulet#2	f(pierre) = cassoulet#2
charlotte	pytt_i_panna#4	f(charlotte) = pytt_i_panna#4
dave	yorkshire_pudding#4	f(dave) = yorkshire_pudding#4

or in a predicate form:

```
all(X, patron(X), a(Y, meal(Y), ordered(X, Y))).
```

In such a logical form, each value of X should be mapped onto a specific value of Y: each patron has eaten his/her own and unique meal. A definition in extension of this sentence – that is, the list of all the facts it encompasses – could be:

Pierre ordered a cassoulet,
Charlotte ordered a pytt i panna, and
Dave ordered a Yorkshire pudding.

Doing so, we have defined a function linking each value of X with a unique value of Y, that is, *Pierre* with a specific *cassoulet*, *Charlotte* with a *pytt i panna*, and *Dave* with a *Yorkshire pudding*. In logic, this is called a Skolem function (Table 16.6).

Our Skolem function has eliminated variable y and the existential quantifier. It has replaced them by $f(x)$ in the logical form:

$$\forall x, patron(x) \Rightarrow ordered(x, f(x))$$

or

```
all(X, patron(X), a(f(X), meal(f(X)), ordered(X, f(X))))
```

More generally, Skolemization handles logical formulas with universally quantified variables, $x_1, x_2, \ldots, x_n$, and a variable existentially quantified y on its left-hand side:

$$\forall x_1, \forall x_2, \ldots, \forall x_n, \exists y, pred(x_1, x_2, \ldots, x_n, y)$$

It substitutes y by a function of the universally quantified variables:

$$y = f(x_1, x_2, \ldots, x_n)$$

yielding unique values for each n-tuplet $(x_1, x_2, \ldots, x_n)$.

Skolemization results in a new formula, where variable y has disappeared:

$$\forall x_1, \forall x_2, \ldots, \forall x_n, pred(x_1, x_2, \ldots, x_n, f(x_1, x_2, \ldots, x_n))$$

and where $f(x_1, x_2, \ldots, x_n)$ is called a Skolem function.

16.6 Solving Coreferences

Although coreferences to a same object are frequently ambiguous, they generally raise no understanding problem to a human reader, with the exception of poorly written texts. However, they represent a tricky issue for a machine. The field has long been dominated by complex linguistic theories that are difficult to implement and to process. Fortunately, as with partial parsing, the MUCs, and later CoNLL, have focused research on concrete problems and robust algorithms that revolutionized coreference resolution.

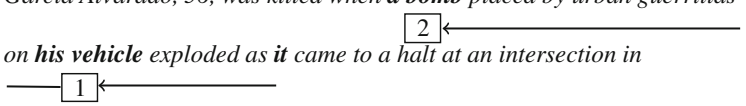
In the next sections, we describe algorithms to automatically resolve coreferences. We first introduce systems based on manually written rules and then describe an efficient machine-learning approach. Even if coreference algorithms do not reach the performance of POS taggers or noun group detectors, they have greatly improved recently and can now be applied to unrestricted texts.

16.6.1 A Simplistic Method: Using Syntactic and Semantic Compatibility

A basic rule that links an anaphor and its antecedent is that their number and gender are identical. This yields the idea of a simplistic method to resolve anaphoric pronouns. The algorithm first collects a list of all the mentions. When an anaphor occurs, the antecedent is searched backward in this list. We set aside cataphoras here. The resolution retains the first antecedent it finds in the list – the most recent one – that agrees in gender and number.

This method may seem naïve, but in fact, most of the time the first antecedent occurring in the sentence or in the previous one with matching gender and number is the good one. This **recency** principle has been observed in many experimental studies. The method ranks properly potential antecedents of *it* in the sentence:

Garcia Alvarado, 56, was killed when ***a bomb*** *placed by urban guerrillas*
on ***his vehicle*** *exploded as* ***it*** *came to a halt at an intersection in*
downtown San Salvador



We can extend this resolution method to find antecedents of definite noun phrases. The recency principle remains the same, but in addition to syntactic features such as gender and number, we add semantic constraints. We search the antecedent of a definite noun phrase, considered as an anaphor, among the entities that are semantically compatible. Compatibility takes the form of:

- The identity – identical noun groups indicate a same reference
- A direct ontological link between mentions – generalization or specialization as in *a car* and *the vehicle*, or
- Compatible modifiers – adjectives and complements of the head noun as in *car*, *white car* or *police car*, but not in *police car* and *ambulance*

Huls et al. (1995) report that such a method identifies pronoun anaphor coreferences with an accuracy of 95 %. Although this figure would probably degrade in some cases, it proves the power and effectiveness of this very simple model. The existence of gender for nouns in French and in German makes the search probably more accurate in these languages.

16.6.2 Solving Coreferences with Shallow Grammatical Information

Kameyama (1997) proposed an algorithm using manually written rules that produced good results in the MUC contest for the coreference resolution task. Here is a slightly modified version of her algorithm. It operates on pronouns and definite noun groups only. It sets aside others such as indefinite and possessive noun groups.

The algorithm first extracts all nominal expressions of the text to form the set of mentions. Then, it scans these expressions in left-to-right order, and for each pronoun or definite noun group, it collects preceding nominal expressions – the potential antecedents – within a definite span of a couple of sentences. The exact window size depends on the type of referring expression:

- The entire MUC text preceding the current expression for proper nouns.
- Narrower for definite noun phrases. Kameyama suggests ten sentences.
- Even narrower for pronouns. Again, Kameyama suggests three sentences.
- The current sentence for reflexive pronouns.

The algorithm applies constraints on the collected nominal expressions to check the compatibility between the current entity E and possible antecedents:

- Number and gender consistency: both must coincide. In some cases, such as with organizations, plural pronouns may denote a singular antecedent.
- Ontological consistency: type of E must be equal to the type of the antecedent or subsume it. For instance, *the automaker* is a valid antecedent of *the company*, but not the reverse.
- Modifier consistency: modifiers such as adjectives must not contradict such as in *the British company* and *the French company*.

Then, among possible candidates, the algorithm retains the one whose **salience** is the highest. This salience is based on the prominence of certain elements in a sentence, such as subjects over objects, and on obliteration with time (or recency).

It has its origin in a rough model of human memory. Memory tends to privilege recent facts or some rhetoric or syntactic forms. A linear ordering of candidates approximates salience in English because subjects have a relatively rigid location in front of the sentence. Kameyama's salience ranks candidates from:

1. The preceding part of the same sentence in left–right order (subject salience)
2. The immediately preceding sentence in left–right order (subject salience)
3. Other preceding sentences within the window in right–left order (recency)

In addition, the algorithm improves the name recognition with aliases. Companies are often designated by full names, partial names, and acronyms to avoid repetitions. For example, consider *Digital Equipment Corporation*, *Digital*, *DEC*. An improvement to coreference recognition is to identify full names with substrings of them and their acronyms.

16.6.3 Salience in a Multimodal Context

EDWARD (Huls et al. 1995) is a model that extends salience to a gesture designation of entities. EDWARD is part of a system that is intended to control a graphical user interface made of windows containing icons that represent files. The interface accepts natural language and mouse commands to designate objects, that is, to name them and to point at them. This combination of modes of interaction is called **multimodality**.

The multimodal salience model keeps the idea of recency in language. The subject of the sentence is also supposed to be retained better than its object, and an object better than an adjunct. In addition, the model integrates a graphical salience and a possible interaction. It takes into account the visibility of entities and pointing gestures. Syntactic properties of an entity are called linguistic context factors, and visual ones are called perceptual context factors. All factors: subject, object, visibility, interaction, and so on, are given a numerical value. A pointed object has the highest possible mark.

The model uses a time sliding window that spans a sentence. It creates the discourse entities of the current window and assigns them a weight corresponding to their contextual importance. Computation of an entity's weight simply sums up all the factors attached to it. An entity salience is then mapped onto a number: its weight. Then the window is moved to the next sentence, and each factor weight attached to each entity is decremented by one. An entity mentioned for the first time and in the position of an object has a context factor weight – a salience – of 3. The next sentence, its worth will be 2, then 1, and finally 0 (Table 16.7).

The model sequentially processes the noun phrases of a sentence. To determine coreferring expressions of the current noun phrase, the model selects all entities semantically compatible with it that have been mentioned before. The one that has the highest salience value among them is retained as a coreference. Both salience

Table 16.7 Context factors (simplified) according to Huls et al. (1995). Note that a subject appears twice in the context factor list: as a subject and as a major constituent

Context factors (CF)	Objects in Scope	Successive weights
Linguistic CFs		
Major-constituent referents CF	Referents of subject, (in)direct object, and modifier	[3, 2, 1, 0]
Subject referent CF	Referent of the subject phrase	[2, 1, 0]
Nested-term referent CF	Referents of the noun phrase modifiers (e.g., prepositional phrase, relative clause)	[1, 0]
Perceptual CFs		
Visible referent CF	Referents visible in the current viewpoint. Typically icons visible in a window	[1, . . . , 1, 0]
Selected referent CF	Referents selected in the model world. Typically icons selected – highlighted – with the mouse or by a natural language command	[2, . . . , 2, 0]
Indicated referent CF	Referents indicated by a pointing gesture. Typically an icon currently being pointed at with a mouse	[30, 1, 0]

Table 16.8 Computation of the salience value (SV) of *Lyn*, *Susan*, and *Ferrari*

	SV of Susan	SV of Lyn	SV of Ferrari
Initial values	0	0	0
Susan drives a Ferrari	$3+2=5$ major + subject	0 major	3
Decay after completion	$3-1+2-1=3$		$3-1=2$
She drives too fast	$3+3+2=8$ existing + major + subject	0	2
Decay after completion	$3-1-1+2-1-1+3-1+2-1=4$		$3-1-1=1$
Lyn races her on weekends	$4+3=7$ existing + major	$3+2=5$ major + subject	1
Decay after completion	$3-1-1-1+3-1-1+2-1-1+3-1=3$	$3-1+2-1=3$	$3-1-1-1=0$
She often beats her	$3+3+2=8$ existing + major + subject	$3+3=6$ existing + major	0

values are then added: the factor brought by the current phrase and the accumulated salience of its coreference. All entities are assigned values that are used to interpret the next sentence. Then, the decay algorithm is applied and the window is moved to the next sentence.

Table 16.8 shows a processing example. It indicates the salience values of *Lyn*, *Susan*, and *Ferrari*. In case of ambiguous reference, the system would ask the user to indicate which candidate is the right one.

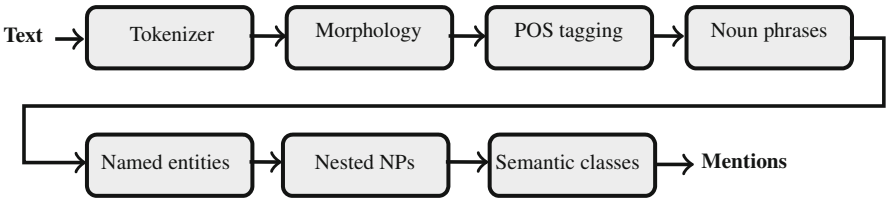


Fig. 16.2 Pipeline to extract the mentions

16.6.4 Using a Machine-Learning Technique to Resolve Coreferences

Algorithms we have seen so far are based on manually engineered rules. This strategy requires a good deal of expertise and considerable clerical work to test and debug the rules. In this section, we introduce a machine-learning approach where the coreference solver uses a classifier automatically trained from a hand-annotated corpus (Soon et al. 2001).

The coreference solver is a decision tree. It considers pairs of mentions, noun phrases and pronouns, (NP_i, NP_j), where each pair is represented by a feature vector of 12 parameters. The solver first extracts pairs of mentions and computes feature vectors for each pair. It then takes the set of mention pairs as input and decides for each pair whether it corefers or not. Using the transitivity property, it identifies all the coreference chains in the text.

The ID3 learning algorithm (Quinlan 1986) (Sect 4.3.2) automatically induces the decision tree from annotated texts using the MUC annotation standard (Sect. 16.4.3).

Mention Detection

To create the set of mentions, the solver extracts all the noun phrases and pronouns from the text. Using a POS tagger, a constituent parser, and the Penn Treebank annotation, as in Table 16.5, this would correspond to the `NP` constituents and words with the `PRP` and `PRP$` tags.

Instead, Soon et al. (2001) used a partial parser, a chunker, to detect the mentions, probably because fast and reliable constituent parsers were still uncommon at the time they created their algorithm. They applied a processing pipeline similar to what we have seen in information extraction: tokenization, morphological processing, POS tagging, noun phrase identification, named entity recognition, nested noun phrase extraction, and semantic class determination (Fig. 16.2).

The four first modules are generic to many language processing applications. The named entities module follows the MUC style and extracts organization, person,

location, date, time, money, and percent entities. When a noun phrase and a named entity overlap, they are merged to form a single noun phrase.

The Nested NPs module is a work-around to a constituent parser to detect mentions inside noun phrases. It has two roles:

1. It extracts the possessive pronouns from the noun phrases as in *his long-term strategy* that results in two mentions: *his* and *his long-term strategy.*
2. It also extracts modifier nouns in nominal compounds, as in *wage reductions*, to generate two mentions: *wage* and *wage reduction*.

In a modern coreference solver, the Nested NPs module would be replaced by a parser that would automatically bracket the embedded noun phrases.

Features

As input, the coreference solver takes a pair of extracted mentions (NP_i, NP_j), where NP_i is before NP_j in the text. The solver considers NP_i as a potential antecedent and NP_j as an anaphor and classifies the pair as positive if both NPs corefer, or negative if they do not. Soon et al. (2001) described each pair by a feature vector of 12 parameters that correspond to positional, grammatical, semantic, and lexical properties:

- Positional feature:
  1. Distance (DIST): This feature is the distance between the two mentions measured in sentences: 0, 1, 2, 3, ... The distance is 0 when the mentions are in the same sentence.
- Grammatical features:
  2. i-Pronoun (I_PRONOUN): Is NP_i a pronoun, i.e., personal, reflexive, or possessive pronoun? Possible values are true or false.
  3. j-Pronoun (J_PRONOUN): Is NP_j a pronoun? Possible values are true or false.
  4. Definite noun phrase (DEF_*NP*): Is NP_j a definite noun phrase, i.e., that starts with *the*? Possible values are true or false.
  5. Demonstrative noun phrase (DEM_*NP*): Is NP_j a demonstrative noun phrase, i.e., that starts with *this, that, these, those*? Possible values are true or false.
  6. Number agreement (NUMBER): Do NP_i and NP_j agree in number? Possible values are true or false.
  7. Gender agreement (GENDER): Do NP_i and NP_j agree in gender? Possible values are true, false, or unknown.
  8. Both proper nouns (PROPER_NOUN): Are NP_i and NP_j both proper nouns? Proper nouns are determined using capitalization. Possible values are true or false.
  9. Appositive (APPOSITIVE): Is NP_j an apposition to NP_i, as *the chairman of Microsoft* in *Bill Gates, the chairman of Microsoft*, ...

Table 16.9 Feature vector of the mention pair: NP_i = *Frank Newman* and NP_j = *vice chairman* (After Soon et al. (2001))

Feature type	Feature	Value	Comments
Positional	DIST	0	NP_i and NP_j are the same sentence
Grammatical	I_PRONOUN	–	NP_i is not a pronoun
	J_PRONOUN	–	NP_j is not a pronoun
	DEF_NP	–	NP_j is not a definite NP
	DEM_NP	–	NP_j is not a demonstrative NP
	NUMBER	+	NP_i and NP_j are both singular
	GENDER	1	NP_i and NP_j are both males (false = 0, true = 1, unknown = 2)
	PROPER_NOUN	–	Only NP_i is a proper noun
	APPOSITIVE	+	NP_j is an apposition to NP_i
Semantic	SEMCLASS	1	NP_i and NP_j are both persons (false = 0, true = 1, unknown = 2)
	ALIAS	–	NP_j is not an alias of NP_i
Lexical	STR_MATCH	–	NP_i and NP_j do not match

- Semantic features:

10. Semantic class agreement (SEMCLASS): Do NP_i and NP_j have the same semantic class? Possible values are true, false, or unknown. Classes are organized as a small ontology with two main parts, person and object, themselves divided respectively into male and female, and organization, location, date, time, money, and percent. The head nouns of the NPs are linked to this ontology using the WordNet hierarchy.
11. Alias (ALIAS): Are NP_i and NP_j aliases, for instance, *IBM* and *International Business Machines*? Possible values are true or false.

- Lexical feature:

12. String match (STR_MATCH): Are NP_i and NP_j equal after removing articles and demonstratives from both mentions? Possible values are true or false.

Table 16.9 shows an example of feature vector for the pair *Frank Newman* and *vice chairman* excerpted from the next sentence (Soon et al. 2001):

> Separately, Clinton transition official said that *Frank Newman*, 50, *vice chairman* and chief financial officer of BankAmerica Corp., is expected to be nominated as assistant Treasury secretary for domestic finance.

The set proposed by Soon et al. (2001) mainly consists of Boolean features. As with chunking or parsing, it is possible to add word values to the feature vector and train lexicalized models. Most lexicalized features are based on the head nouns of the mentions or other dependency links that we extract with percolation rules as those shown in Table 11.15 or with a dependency parser.

Lexicalized models improve the performance of coreference solving. Björkelund and Nugues (2011) showed that the feature bigram consisting of the head noun of

the antecedent and the head noun of the anaphor was one of the most significant. However, lexicalized models require more annotated data than those based on Booleans or parts of speech.

Training Examples

The training procedure extracts positive and negative examples from the annotated corpus:

- It generates the positive examples using pairs of adjacent coreferring mentions. If $NP_{a1} - NP_{a2} - NP_{a3} - NP_{a4}$ is a coreference chain in a text, the positive examples correspond to the set of pairs:

$$\{(NP_{a1}, NP_{a2}), (NP_{a2}, NP_{a3}), (NP_{a3}, NP_{a4})\},$$

  where the first mention is always considered to be the antecedent and the second one the anaphor.
- To create the negative examples, the training procedure considers the same adjacent pairs antecedent, anaphor (NP_i, NP_j), and the mentions intervening between them $NP_{i+1}, NP_{i+2}, \ldots, NP_{j-1}$. For each positive pair (NP_i, NP_j), the training procedure generates negative pairs, which consist of one intervening mention and the anaphor NP_j:

$$\{(NP_{i+1}, NP_j), (NP_{i+2}, NP_j), \ldots, (NP_{j-1}, NP_j)\}.$$

  The intervening mentions can either be part of another coreference chain or not.

As classifier, Soon et al. (2001) induced a decision tree from the examples with the ID3 algorithm. It is possible to use other binary classifiers such as logistic regression or support vector machines.

Extracting the Coreference Chains

Once the classifier has been trained, it is applied to the mentions in a text to identify the coreference chains. The engine first extracts all the mentions in the text. It traverses the text from left to right from the second mention. For each current NP_j, the algorithm considers every NP_i before it as a possible antecedent. It then proceeds from right to left and submits the pairs (NP_i, NP_j) to the classifier until it reaches an antecedent or the start of the text.

The algorithm is as follows:

1. Let $NP_1, NP_2, \ldots, NP_N$ be the mentions.
2. For $j = 2$ to N.

(a) For each NP_j, generate all the pairs (NP_i, NP_j), where $i < j$.
(b) Compute the feature vector of each pair (NP_i, NP_j).
(c) For $i = j - 1$ to 1, submit the pair (NP_i, NP_j) to the classifier until a positive pair is found or the beginning of the text is reached.
(d) If a noun phrase returns positive, NP_j has an antecedent and is part of the corresponding coreference chain.

16.6.5 More Complex Phenomena: Ellipses

An **ellipsis** is the absence of certain words or phrases normally necessary to build a sentence. Ellipses occur frequently in the discourse to avoid tedious repetitions. For instance, the sequence:

I want to have information on caterpillars. And also on hedgehogs.

features a second sentence whose subject and verb are missing. The complete sentence would be:

I want to have information on hedgehogs.

Here the speaker avoids saying twice the same thing. Ellipses also occur with clauses linked by conjunctions where a phrase or a word is omitted as in the sentence:

I saw a hedgehog walking on the grass and another sleeping.

Everyone, however, can understand that it substitutes the complete sentence:

I saw a hedgehog walking on the grass and I saw another hedgehog sleeping.

Ellipses are rather difficult to handle. In many cases, however, maintaining a history of all the discourse's referents can help retrieve an omitted referent or verb. A referent missing in a sentence can be searched backward in the history and replaced with an adequate previous one.

16.7 Centering: A Theory on Discourse Structure

Of the many theories on discourse structure, Grosz and Sidner's (1986) has been very influential in the computational linguistics community. Grosz and Sidner modeled a discourse as being a composite of three components:

- The linguistic structure of the actual sequence of utterances in the discourse
- A structure of intentions
- An attentional state

Grosz and Sidner's first assumption is that the linguistic structure of a discourse is made of segments. They substantiated this claim using psychological studies

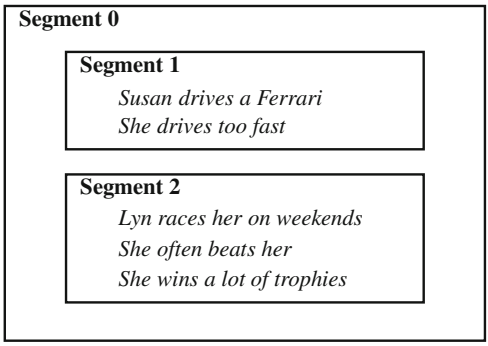


Fig. 16.3 The embedded structure of discourse. Segment 0 covers the five sentences and spans segment 1 (1 and 2) and segment 2 (3–5)

showing a relative agreement among individuals over the segmentation of a text: given a text, individuals tend to fractionate it in a same way. Segments have a nonstrict embedded (hierarchical) organization (Fig. 16.3). It is roughly comparable to that of the phrase structure decomposition of a sentence. Segment boundaries are often delimited by clues and **cue phrases**, also called markers, that indicate transitions.

The intentional structure is what underlies a discourse. It is the key to how segments are arranged and their internal coherence. It has global and local components. From a global viewpoint, intention relates to the **discourse purpose**, which is the main objective of the discourse and why it takes place. Within each segment there is a **discourse segment purpose** that is local and that contributes to the main purpose. Discourse segment purposes are often easier to determine than the overall discourse intention.

The attentional state is the dynamic set of objects, relations, and properties along with the discourse. The attentional state is closely related to segments. For each of them there is a focus space made of salient entities, properties of entities, and relations between entities, that is, predicates describing or linking the entities. The attentional state also contains the discourse segment purpose.

While Grosz and Sidner's general model may prove difficult to implement, Grosz et al. (1995) derived a simpler concept of **centering** from it. Centering retains the idea of segment, defined as a set of utterances, along which a limited number of dynamic centers turn up. Centers are the "useful" entities of an utterance that link it to other utterances of a segment. Since centers are a subset of entities, they are easier to detect than the intention or the whole attentional state. They provide a tentative model to explain discourse coherence and coreference organization.

Centers of an utterance are split into a set of **forward-looking centers** and a unique **backward-looking center**, except for the first utterance of the segment, which has no backward-looking center:

- The backward-looking center, or simply the center, is the entity that connects the current utterance with the previous one and hence with one of the previous forward-looking centers. It is often a pronoun.

- Forward-looking centers are roughly the other discourse entities of a segment. More precisely, they are limited to entities serving to link the utterance to other utterances.

Forward-looking centers can be ordered according to syntactic, semantic, and pragmatic factors, and the first one has a great chance to become the backward-looking center of the next utterance. As examples, centers in Table 16.1 are:

- In sentence 1, *Susan* and *Ferrari* are the discourse entities and forward-looking centers.
- In sentence 2, *she* is the backward-looking center because it connects the utterance with the previous one.
- In sentence 3, *Lyn* and *weekends* are the forward-looking centers; *her* is the backward-looking center.

16.8 Discourse and Rhetoric

Rhetoric also offers means to explain discourse coherence. Although rhetoric has a very long tradition dating from ancient times, modern linguists have tended to neglect it, favoring other models or methods. Recently however, interest has again increased. Modern rhetorical studies offer new grounds to describe and explain argumentation. Modeling argumentation complements parts of human discourse that cannot only be explained in terms of formal logic or arbitrary beliefs. The *Traité de l'argumentation* by Perelman and Olbrechts-Tyteca (1976) is a prominent example of this trend.

On a parallel road, computational linguistics also rediscovered rhetoric. Most of the renaissance in this community is due to influential papers on rhetorical structure theory (RST) by Mann and Thompson (1987, 1988). This section provides a short introduction to ancient rhetoric and then describes RST.

16.8.1 Ancient Rhetoric: An Outline

Rhetoric was studied in most schools of ancient Greece and Rome, and in universities in the Middle Ages. Rhetoric was then viewed as a way to define how best to compose ideas in a discourse, to make it attractive, to convince and persuade an audience. It was considered as a kind of discourse strategy defining the optimal arrangement or planning of arguments according to the type of audience, of speech case, etc.

According to the ancient rhetoric school, the production of discourse had to be organized around five canons – invention, arrangement, style, memory, and delivery.

- Invention (*inventio*) is related to the ideas or facts contained in a discourse: what to say or to write are the first things to identify to make a discourse exist. According to ancient Greeks, a key to invention was to answer the right questions in the right order.
- Arrangement (*dispositio*) is the discourse construction for which general patterns have been proposed. According to Cicero, a discourse should feature an introduction (*exordium*), a narrative (*narratio*) where the orator sets forth the issues of the problem, a proposition (*propositio*) where s/he states her/his arguments for the case, a refutation (*refutatio*), where s/he gives counterarguments, a confirmation (*confirmatio*) where s/he reinforces her/his arguments, and finally a conclusion (*peroratio*).
- Style (*elocutio*) concerns the transcription and the editing of ideas into words and sentences. Rules of style suggested to privilege clarity – use plain words and conform to a correct grammar. This was a guarantee to be understood by everybody. Style was also a literary art where efficiency mattered most. It was divided into three categories whose goals were to emote (*movere*), to explain (*docere*), or to please (*delectare*) according to the desired effect on the audience.
- Memory (*memoria*) was essential so that the orator should retain what s/he had to say. The Ancients advised orators to sleep well, to be in good shape, to exercise memory by learning by heart, and to use images.
- Delivery (*actio*) concerned the uttering of the discourse: voice, tone, speed, and gestures.

Although current discourse strategies may not be the same as those designed and contrived in Athens or Sicily 2,500 years ago, if elucidated they give keys to a discourse structure. Later, the historical definition of rhetoric was sometimes superseded by a pejorative sense meaning empty political speeches or ranting.

16.8.2 Rhetorical Structure Theory

Rhetorical structure theory (RST) is a theory of text organization in terms of relations that occur in a text. As for Grosz and Sidner, RST identifies a hierarchical tree structure in texts. A text consists of nonoverlapping segments that define the tree nodes. These segments are termed by Mann and Thompson as "text spans." They correspond typically to one or more clauses. Text spans may be terminal or nonterminal nodes that are linked in the tree by relations.

Rhetorical relations are sorts of dependencies between two text spans termed the **nucleus** and the **satellite**, where the satellite brings some sort of support or explanation to the nucleus, which is the prominent issue. To illustrate this concept, let us take the example of the *Justify* relation from Mann and Thompson (1987, pp. 9–11): "A justify satellite is intended to increase the reader's readiness to accept the writer's right to present the nuclear material." In the short text:

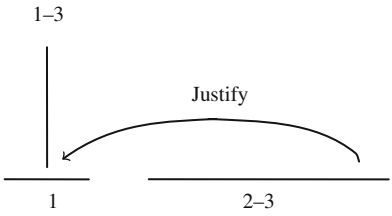


Fig. 16.4 The Justify relation

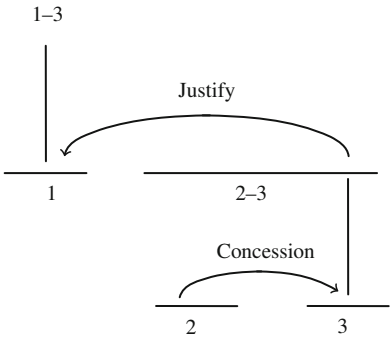


Fig. 16.5 More relations: Concession

1. *The next music day is scheduled for July 21 (Saturday), noon–midnight*
2. *I'll post more details later,*
3. *but this is a good time to reserve the place on your calendar.*

segments 2 and 3 justify segment 1, and they can be represented graphically by Fig. 16.4.

Segments can then be further subdivided using other relations, in the example a *Concession* (Fig. 16.5).

Relations are easy to represent in Prolog with facts `rhetorical_relation(relation_type, satellite, nucleus)`:

```
rhetorical_relation(justify, 3, 1).
rhetorical_relation(concession, 2, 3).
```

Another example is given by this odd text about dioxin (Mann and Thompson 1987, pp. 13–15):

1. *Concern that this material is harmful to health or the environment may be misplaced.*
2. *Although it is toxic to certain animals,*
3. *evidence is lacking that it has any serious long-term effect on human beings.*

which can be analyzed with relations *Elaboration* and *Concession* in Fig. 16.6. These relations are equivalent to the Prolog facts:

```
rhetorical_relation(elaboration, 3, 1).
rhetorical_relation(concession, 2, 3).
```

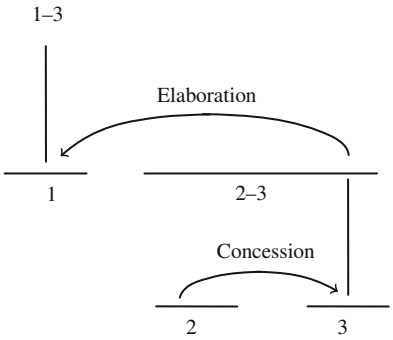


Fig. 16.6 Elaboration and Concession

Circumstance	Evidence	Otherwise
Solutionhood	Justify	Interpretation
Elaboration	Cause	Evaluation
Background	Antithesis	Restatement
Enablement	Concession	Summary
Motivation	Condition	

Fig. 16.7 RST rhetorical relations linking a nucleus and a satellite

Sequence Joint Contrast

Fig. 16.8 Relations linking two nuclei

16.8.3 Types of Relations

The total number and the type of rhetorical relation vary much among authors and even among papers written by their creators. Their number ranges from a dozen to several hundreds. As we saw in the previous section, most relations link a nucleus and a satellite. Figure 16.7 shows a slightly simplified list of them from Mann and Thompson (1987). In some instances, relations also link two nuclei. They are shown in Fig. 16.8.

16.8.4 Implementing Rhetorical Structure Theory

Mann and Thompson gave formal definitions of rhetorical relations using constraints on the satellite, the nucleus, and both. Table 16.10 shows constraints holding for *evidence*. In addition, a rhetorical relation entails consequences that are described by an effect: here, with *evidence*, the reader's belief of the nucleus is increased.

Table 16.10 The EVIDENCE relation (After Mann and Thompson (1987))

Relation name	EVIDENCE
Constraints on the nucleus N	The reader R might not believe to a degree satisfactory to the writer W
Constraints on the satellite S	The reader believes S or will find it credible
Constraints on the $N + S$ **combination**	R's comprehending S increases R's belief of N
The effect	R's belief of N is increased
Locus of the effect	N

Table 16.11 Examples of cue phrases and forms

Cues	**English**	**French**	**German**
Conjunctions	*Because, in fact, but, and*	*Car, en effet, puisque, et, mais*	*denn, und, aber,*
Adverbial forms	*In addition, for example*	*De plus, en particulier, particulièrement, par exemple*	*dazu, besonders, zum Beispiel*
Syntactic forms	Past participles: *given*	Present participles: *étant donné*	

Table 16.12 Typical orders for some relations

Satellite before nucleus		
	Antithesis	Condition
	Background	Justify
	Concession	Solutionhood
Nucleus before satellite		
	Elaboration	Evidence
	Enablement	Statement

Such constraints are difficult – if not impossible – to implement in a computer as is because they involve knowing the thoughts of the reader and the writer. However, rhetorical relations are often indicated by a handful of specific cue words or phrases. Mann and Thompson observe that a concession is often introduced by *although*, as in the dioxin text from the previous section, or *but*. A common workaround to detect a relation is then to analyze the surface structure made of these cue phrases. They may indicate the discourse transitions, segment boundaries, and the type of relations. Many cue phrases are conjunctions, adverbial forms, or syntactic patterns (Table 16.11). Mann and Thompson also observed that the nucleus and the satellite had typical topological orders (Table 16.12).

Recently, comprehensive works have itemized cue phrases and other constraints enabling the rhetorical parsing of a text. Marcu (1997) and Corston-Oliver (1998) are notable examples of this trend. As an example, Corston-Oliver (1998) recognizes the *Elaboration* relation with a set of necessary criteria that must hold between two clauses, clause 1 being the nucleus and clause 2 the satellite:

Table 16.13 Cues to recognize the *Elaboration* relation (After Corston-Oliver (1998, p. 129))

Cue	Score	Cue Name
Clause 1 is the main clause of a sentence (sentence i), and clause 2 is the main clause of a sentence (sentence j), and sentence i immediately precedes sentence j, and (a) clause 2 contains an elaboration conjunction (*also, for example*), or (b) clause 2 is in a coordinate structure whose parent contains an elaboration conjunction.	35	H24
Cue H24 applies, and clause 1 is the main clause of the first sentence in the excerpt.	15	H26
Clause 2 contains a predicate nominal whose head is in the set {*portion, component, member, type, kind, example, instance*}, or clause 2 contains a predicate whose head verb is in the set {*include, consist*}.	35	H41
Clauses 1 and 2 are not coordinated, and (a) clauses 1 and 2 exhibit subject continuity, or (b) clause 1 is passive and the head of the direct object of clause 1 and the head of the direct object of clause 2 have the same base form, or (c) clause 2 contains an elaboration conjunction.	10	H25
Cue H25 applies, and clause 2 contains a habitual adverb (*sometimes, usually*, ...).	17	H25a
Cue H25 applies, and the syntactic subject of clause 2 is the pronoun *some* or contains the modifier *some*.	10	H38

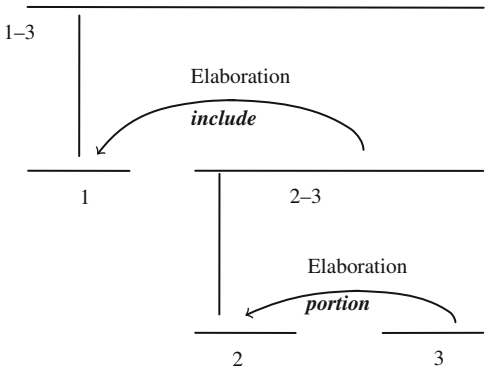


Fig. 16.9 Rhetorical structures

1. Clause 1 precedes clause 2.
2. Clause 1 is not subordinate to clause 2.
3. Clause 2 is not subordinate to clause 1.

and cues that are ranked according to an heuristic score (Table 16.13).

Corston-Oliver (1998) applied these cues to analyze the Microsoft *Encarta* encyclopedia. With the excerpt:

1. *A stem is a portion of a plant.*
2. *Subterranean stems include the rhizomes of the iris and the runners of the strawberry;*
3. *The potato is a portion of an underground stem.*

using cue H41, he could obtain the rhetoric structure shown in Fig. 16.9.

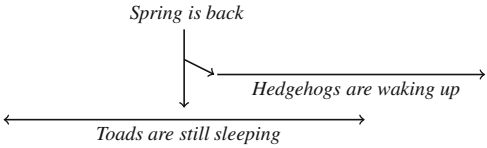


Fig. 16.10 Events

16.9 Events and Time

In most discourses, actions, events, or situations have a **temporal** context. This context is crucial to the correct representation of actions. It involves time, which is reflected by time expressions, such as adverbs or adjuncts, *now*, *tomorrow*, *in 5 minutes*, and verb **tenses**, such as present, past, or future.

Understanding temporal relations between events is difficult and may depend on the language. For instance, there is no exact correspondence for past and present tenses between French and English. In the next section, we will provide hints on theories about temporal modeling.

16.9.1 Events

Research on the representation of time, events, and temporal relations dates back to the beginning of logic. It resulted in an impressive number of formulations and models. A possible approach is to **reify** events, that is, to turn them into objects, to quantify them existentially, and to connect them to other objects using predicates based on action verbs and their modifiers (Davidson 1966). The sentence *John saw Mary in London on Tuesday* is then translated into the logical form:

$$\exists\varepsilon[saw(\varepsilon, John, Mary) \wedge place(\varepsilon, London) \wedge time(\varepsilon, Tuesday)],$$

where ε represents the event.

To represent the temporal context of an action sequence we can use a set of predicates. Consider:

> Spring is back. Hedgehogs are waking up. Toads are still sleeping.

There are obviously three actions or events described here. These events are located in time around a reference point defined by the return of spring. From this point, the hedgehogs' waking up process extends onwards while the toads' sleeping process overlaps it (Fig. 16.10). Events have a different duration: the first sentence merely describes a single time point whereas the two last processes are defined inside intervals.

Let us denote `e1`, `e2`, and `e3` the events in Fig. 16.10, and let us portray them in Prolog. In addition, let us use the agent semantic role that we borrow from the case grammars. We have a first representation:

```
event(e1).
is_back(e1).
agent(e1, spring).
...
event(e2).
waking_up(e2).
agent(e2, hedgehogs).
...

event(e3)
sleeping(e3).
agent(e3, toads).
```

16.9.2 Event Types

Events are closely related to sentence's main verbs, and different classifications have been proposed to associate a verb with a type of event. Vendler's (1967) for English, Gosselin (1996) for French, and others came to a consensus to divide verbs into four categories, denoting:

- A state – a permanent property or a usual situation (e.g., *be*, *have*, *know*, *think*).
- An achievement – a state change, a transition, occurring at single moment (e.g., *find, realize, learn*).
- An activity – a continuous process taking place over a period of time (e.g., *work, read, sleep*). In English, activities often use the present perfect, *-ing*.
- An accomplishment – an activity with a definite endpoint completed by a result (e.g., *write a book, eat an apple*).

Some authors have associated events to verbs only. It is safer, however, to take verb phrases – predicates – and even subjects into account to link events to Vendler's categories (Table 16.14). Compare *The water ran*, which is an activity in the past, and *The hurdlers ran* (in a competition), which depicts an achievement.

16.9.3 Temporal Representation of Events

Let us now try to represent processes in a temporal chronology. In the example in Fig. 16.10, the only process that has a definite location is `e1`. It is associated to a calendar period: *spring*. Other processes are then relative to it. As for these sentences, in most discourses it is impossible to map all processes onto an absolute time. Instead, we will represent them using relative, and sometimes partial, temporal relations.

Table 16.14 Vendler's verb categories

	English	French	German
State	*The cat is sick* *I like chocolate*	*Le chat est malade* *J'aime le chocolat*	*Die Katze ist krank* *Ich esse Schokolade gern*
Activity	*She works for a company* *He is writing a book*	*Elle travaille pour une entreprise* *Il écrit un livre*	*Sie arbeitet für eine Firma* *Er schreibt ein Buch*
Accomplishment	*He wrote a book* *The dormouse ate the pears*	*il a écrit un livre* *Le loir a mangé les poires*	*Er hat ein Buch geschrieben* *Die Haselmaus hat die Birnen gegessen*
Achievement	*The sun set* *I realized I was wrong*	*Le soleil s'est couché* *Je me suis rendu compte que j'avais tort*	*Die Sonne ist untergegangen* *Ich habe eingesehen, ich nicht recht hatte*

Simplifying things, we will suppose that time has a linear ordering and that each event is located in time: it has a certain beginning and a certain end. This would not be true if we had considered conditional statements. Temporal relations associate processes to time intervals and set links, constraints between them. We will adopt here a model proposed by Allen (1983); Allen's (1984), whose 13 relations are listed in Table 16.15.

Using Allen's representation, relations `before(e1, e2)`, `after(e2, e1)`, and `contains(e3, e1)` depict temporal constraints on events `e1`, `e2`, and `e3` in Sect. 16.9.1. Temporal relations result in constraints between all processes that enable a total or partial ordering of them.

16.9.4 Events and Tenses

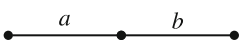
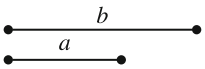
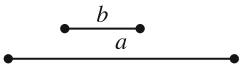
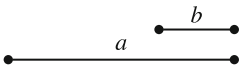
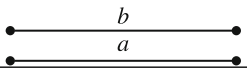
As we saw, event modeling results in time intervals and in relations between them. From event examples in Fig. 16.10, we can define two new temporal facts:

- Instantaneous events, which are punctual and marking a transition
- Situations, which have a duration – true over an interval

Relations as well as events or situations are not accessible directly. As for rhetorical relations or segment boundaries, we need cues or markers to track them. In the example above, we have mapped events onto verbs. This hints at detection and description methods. Although there is no definitive solution on how to detect events, many techniques rely on verbs and verb phrases to act as markers.

A first cue to create and locate an event is the verb tense. A sentence sequence defines a linear sequence of enunciation events. A basic distinction is between the

Table 16.15 Allen's temporal relations

#	Relations	Graphical representations
1.	`before(a, b)`	
2.	`after(b, a)`	
3.	`meets(a, b)`	
4.	`met_by(b, a)`	
5.	`overlaps(a, b)`	
6.	`overlapped_by(b, a)`	
7.	`starts(a, b)`	
8.	`started_by(b, a)`	
9.	`during(b, a)`	
10.	`contains(a, b)`	
11.	`finishes(b, a)`	
12.	`finished_by(a, b)`	
13.	`equals(a, b)`	

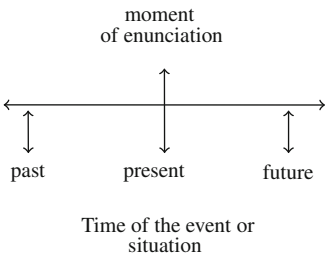


Fig. 16.11 Ideal time: past, present, and future

moment of the enunciation and the time of the event (or situation). Figure 16.11 represents a kind of ideal time.

The sentence

Ernest the hedgehog ate a caterpillar

creates two events; one corresponds to the processes described the sentence, `e1`, and the other, `e2`, to the time of speech. Both events are linked by the relation

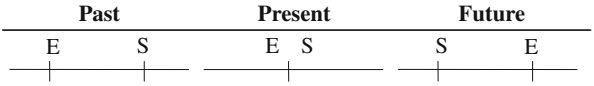


Fig. 16.12 Ideal tenses

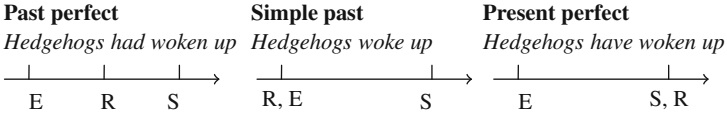


Fig. 16.13 Event, reference, and speech for some English tenses

`before(e1, e2)`. We could have refined the model with a beginning `e1b` and an end `e1e` of *Ernest*'s dinner. New relations would be:

```
before(e1b, e1e).
before(e1b, e2).
before(e1e, e2).
```

Using a verb classification and tenses helps determine the events location or situation boundaries. We may also rely on time adverbs and time adjuncts such as *for five minutes*, *tomorrow*, etc.

The 'ideal' representation, however, is not sufficient to describe many narrative phenomena where the writer/reader viewpoint is moved relatively to temporal events. Reichenbach (1947) elaborated a more complex representation to take this viewpoint into account. Basically, verb tenses are mapped onto a triple representing on a linear scale the point of the event or situation denoted E, the point of speech denoted S, and a point of reference denoted R. The reference corresponds to a sort of writer/reader viewpoint.

Let us first consider the time of speech and the event. It is clear to the reader that an event described by basic tenses, past, present, and future, is respectively before, coinciding, and after the point of speech (Fig. 16.12).

Reichenbach's tense model introduces the third point to position events relatively in the past or in the future. Consider the past sentence

Hedgehogs had already woken up when the sun set.

Two events are described, the hedgehogs' waking up, `ewu`, and the sunset, `ess`. Among the two events, the speaker viewpoint is focused by the clause *Hedgehogs had already woken up*: then, the action takes place. This point where the speaker moves to relate the story is the point of reference of the narrative, and the event is before it (Fig. 16.13). The point of reference of the first process enables us to locate the second one relatively to it and to order them in a sequence.

Some tenses describe a time stretch of the event, as for the French *imparfait* compared to the *passé composé* (Fig. 16.14), or continuous tenses of English (Fig. 16.15).

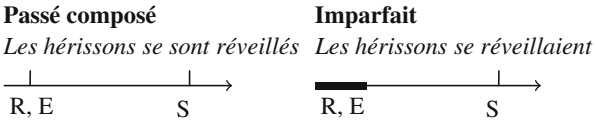


Fig. 16.14 French imparfait and passé composé

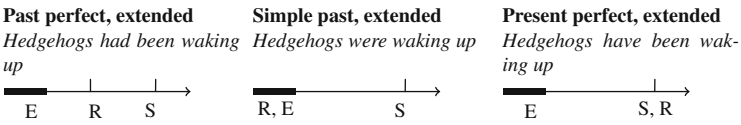


Fig. 16.15 Some English tenses involving a stretch of time

16.10 TimeML, an Annotation Scheme for Time and Events

Several schemes have been proposed to annotate temporal information in texts. Many of them were incompatible or incomplete, and in an effort to reconcile and unify the field, Ingria and Pustejovsky (2004) introduced the XML-based Time Markup Language (TimeML). TimeML is a specification language whose goal is to capture most aspects of temporal relations between events in discourses. It is based on Allen's (1984) relations and inspired by Vendler's (1967) classification of verbs.

TimeML defines the XML elements `TIMEX3` to annotate time expressions (*at four o'clock*), `EVENT` to annotate the events (*he slept*), and "signals." The `SIGNAL` tag marks words or phrases indicating a temporal relation. It includes function words such as *later* and *not* (*he did not sleep*). TimeML also features elements to connect entities using different types of links, most notably temporal links, `TLINK`s, that describe the temporal relation holding between events or between an event and a time.

TimeML elements have attributes. For instance, events have a tense, an aspect, and a class. The seven possible classes denote the type of event, whether it is a `STATE`, an instantaneous event (`OCCURRENCE`), etc.

The sentence

> All 75 people on board the Aeroflot Airbus died when it ploughed into a Siberian mountain in March 1994

is marked up as follows (Ingria and Pustejovsky 2004):

```
All 75 people
<EVENT eid="e7" class="STATE">on board</EVENT>
<MAKEINSTANCE eiid="ei7" eventID="e7" tense="NONE"
aspect="NONE"/>
<TLINK eventInstanceID="ei7" relatedToEvent="ei5"
relType="INCLUDES"/>
the Aeroflot Airbus
```

```
<EVENT eid="e5" class="OCCURRENCE" >died</EVENT>
<MAKEINSTANCE eiid="ei5" eventID="e5" tense="PAST"
aspect="NONE"/>
<TLINK eventInstanceID="ei5" signalID="s2"
relatedToEvent="ei6" relType="IAFTER"/>
<SIGNAL sid="s2">when</SIGNAL>
it
<EVENT eid="e6" class="OCCURRENCE">ploughed</EVENT>
<MAKEINSTANCE eiid="ei6" eventID="e6" tense="PAST"
aspect="NONE"/>
<TLINK eventInstanceID="ei6" signalID="s3"
relatedToTime="t2" relType="IS_INCLUDED"/>
<TLINK eventInstanceID="ei6" relatedToEvent="ei4"
relType="IDENTITY"/>
into a Siberian mountain
<SIGNAL sid="s3">in</SIGNAL>
<TIMEX3 tid="t2" type="DATE" value="1994-04">March 1994
</TIMEX3>.
```

In the example, three events `e5`, *died*, `e6`, *ploughed*, and `e7`, *on board*, are annotated and instantiated using the `MAKEINSTANCE` tag. The text contains one time expression, *March 1994*, which is annotated using `TIMEX3`. The events and the time expressions are connected by two temporal links, `TLINK`. The first link specifies that the passengers died after the plane ploughed, using the `relatedToEvent` attribute. The second link specifies that event *ploughed* is included in *March 1994*. A third and last `TLINK` refers to an event, `e4`, mentioned in a previous, noncited sentence. The temporal signals *when* and *in* can also be relevant, and they are tagged with a `SIGNAL` tag.

16.11 Further Reading

Schiffrin (1994) and Coulthard (1985) give general introductions to discourse. Ducrot and Schaeffer (1995) and Simone (2007) provide shorter and very readable accounts. Tesnière (1966) gives an outstanding description of anaphora (Chap. 42) and anaphors (Chap. 43). Kamp and Reyle (1993) provide a thorough logical model of discourse that they called the *discourse representation theory* – DRT.

The *Message Understanding Conferences* (MUCs) spurred very pragmatic research on discourse, notably on named-entity recognition and coreference resolution. Although considered a relatively low-level component, named-entity recognition has proven essential to many discourse applications. The MUCs, followed by CoNLL 2002 and 2003, created a framework to develop NER systems consisting of annotated data and a metric to evaluate the performance of systems. This gave birth to an uncountable number of papers and programs. The open-source *Stanford*

Named Entity Recognizer is one of them (Finkel et al. 2005) (http://nlp.stanford.edu/software/CRF-NER.shtml).

The MUCs also produced a coreference annotation scheme that became a standard and gave researchers the first tools to evaluate their algorithms. While the MUCs annotated a fairly small amount of data from unbalanced sources, OntoNotes is a subsequent project that tried to overcome these limitations. OntoNotes is a corpus of documents in Arabic, Chinese, and English totaling 2 million words and covering multiple domains: newswire, broadcast news, broadcast conversation, magazine, and web. It provided the material for the CoNLL 2011 and 2012 shared tasks (Pradhan et al. 2012, 2011).

Coreference solvers use either rules or statistical techniques. The open-source *Stanford Deterministic Coreference Resolution System* is a high-performance, rule-based solver for English (Raghunathan et al. 2010) (http://nlp.stanford.edu/software/dcoref.shtml). Soon et al. (2001) were first to develop a statistical system equaling the performance of those based on manually written rules; see Sect. 16.6.4. Most statistical solvers still use this algorithm with some variations. See the proceedings of the CoNLL 2011 and 2012 shared tasks for a description of improvements and multilingual feature sets. The evaluation of coreference solvers is still a matter of debate. CoNLL 2011 and 2012 used the average of three different metrics: MUC, B-CUBED, and CEAF. See Vilain et al. (1995), Bagga and Baldwin (1998), and Luo (2005) for their respective descriptions.

Linking words to real word entities has become a core step of semantic and discourse interpretation as underlined by the new motto of the search team at Google: *Things, not strings* (Singhal 2012). In Sect. 16.3.4, we introduced an algorithm to disambiguate entities. Cucerzan (2007) presents another disambiguation algorithm and Hoffart et al. (2011) describe AIDA, an open-source system available at http://www.mpi-inf.mpg.de/yago-naga/aida/.

Introductions to rhetoric include books by Corbett and Connors (1999), Reboul (1994), and Perelman and Olbrechts-Tyteca (1976). Annotated discourse treebanks include the *RST discourse treebank* (Carlson et al. 2003) and the *Penn discourse treebank* (PDTB) (Prasad et al. 2008). The PDTB site contains numerous papers on discourse parsing (http://www.seas.upenn.edu/~pdtb/).

Time processing in texts is still a developing subject. Reichenbach (1947) described a model for all the English tenses. Starting from this foundational work, Gosselin (1996) provided an account for French that he complemented with a process duration. He described rules and an implementation valid for French verbs. Ter Meulen (1995) describes another viewpoint on time modeling in English, while Gagnon and Lapalme (1996) produce an implementation of time processing for French based on the DRT. Johansson et al. (2005) describe how semantic role labeling and time processing can be used to generate animated 3D scenes from written texts.

Exercises

16.1. Choose a newspaper text of about ten lines. Underline the references and link coreferences.

16.2. Write DCG rules to detect noun groups and pronouns of Exercise 16.1 and collect the discourse entities in a Prolog list.

16.3. Write a grammar recognizing names (proper nouns) in English, French, or German. You will consider that a name consists of a title followed by a surname whose first letter is capitalized.

16.4. Choose a newspaper text of about ten lines. Collect noun groups and pronouns in a list with a noun group detector. Write a coreference solver in Prolog to associate each pronoun to all preceding noun groups.

16.5. Using the program written for Exercise 16.4, write a first predicate that retains the first preceding noun group – first potential antecedent – and a second predicate that retains the two first noun groups.

16.6. Implement the Kameyama algorithm of Sect. 16.6.2 in Prolog.

16.7. Implement the Soon et al. (2001) algorithm of Sect. 16.6.4.

16.8. Select a newspaper article and underline elliptical sentences or phrases.

16.9. Using the result of Exercise 16.8, describe rules that would enable you to resolve ellipses.

16.10. Select one page from a technical text and annotate clauses with rhetorical relations listed in Table 16.9.

16.11. Write rules using the model of Table 16.12 to recognize the EVIDENCE rhetorical relation.

16.12. Describe verb tenses in languages you know in terms of point of the event, of speech, and of reference, as in Sect. 16.9.4.

Chapter 17
Dialogue

τὸν αὐτὸν δὲ λόγον ἔχει ἥ τε τοῦ λόγου δύναμις πρὸς τὴν τῆς ψυχῆς τάξιν ἥ τε τῶν φαρμάκων τάξις πρὸς τὴν τῶν σωμάτων φύσιν. ὥσπερ γὰρ τῶν φαρμάκων ἄλλους ἄλλα χυμοὺς ἐκ τοῦ σώματος ἐξάγει, καὶ τὰ μὲν νόσου τὰ δὲ βίου παύει, οὕτω καὶ τῶν λόγων οἱ μὲν ἐλύπησαν, οἱ δὲ ἔτερψαν, οἱ δὲ ἐφόβησαν, οἱ δὲ εἰς θάρσος κατέστησαν τοὺς ἀκούοντας, οἱ δὲ πειθοῖ τινι κακῇ τὴν ψυχὴν ἐφαρμάκευσαν καὶ ἐξεγοήτευσαν.

Gorgias, *Encomium of Helen*, 14, See translation in Sect. 17.5.

17.1 Introduction

While discourse materialized in texts delivers static information, dialogue is dynamic and consists of two interacting discourses. Once written, a discourse content is unalterable and will remain as it is for its future readers. On the contrary, a dialogue enables exchange information flows, to complement and to merge them in a composition, which is not known in advance. Both dialoguing parties provide feedback, influence, or modify the final content along with the course of the conversation.

In this chapter, we will envision dialogue within the framework of an interface between a system and a user. Parties have information to transmit or to request using natural language. The dialogue purpose will be to make sure that the request is complete or the information has been well captured or delivered. Naturally, as for other discourse applications, a dialogue module is only a part of the whole system using language processing techniques we have described before.

P.M. Nugues, *Language Processing with Perl and Prolog*, Cognitive Technologies,
DOI 10.1007/978-3-642-41464-0_17, © Springer-Verlag Berlin Heidelberg 2014

17.2 Why a Dialogue?

The first role of a dialogue module as an interface is to manage the communication and to coordinate the turn-taking between the user and the system. It is also a kind of integration shell that calls other language processing components to analyze user utterances or to generate system answers. In addition, interaction and dialogue techniques can help linguistic analysis be more flexible and recover from failures.

We saw that coreferences are sometimes difficult to resolve. They provide an example of interaction usefulness. Instead of having an interactive system conjecture about an ambiguous pronoun, it might be safer for it to ask the user himself/herself to resolve the ambiguity. Two strategies are then possible: the system can infer a missing reference and ask the user for a confirmation. Or, in case of a more difficult ambiguity, it can ask the user to reformulate completely his/her sentence.

To summarize, dialogue systems can help manage a user's discourse to:

- Complement information when pieces are missing to understand a sentence or to carry out a command,
- Clarify some ambiguous words or constructions, or
- Confirm information or intention to manage errors when a certain failure rate is unavoidable, e.g., with speech recognition operating on naturally flowing speech.

17.3 Architecture of a Dialogue System

Many dialogue systems in commercial operation aim to provide information through telephones or web browsers. Such **speech servers** receive calls from users requesting information, often in a specific domain, and answer interactively to questions. Although many servers still interact with a user using touch-tone telephones, more and more they feature speech recognition and speech synthesis modules.

Figure 17.1 shows how speech processing is located within a language processing architecture, here to be a natural language interface to a database. In such systems, a speech recognition module transcribes the user's speech into a word stream. The character flow is then processed by a language engine dealing with syntax, semantics, dialogue, and finally by the back-end application program. A speech synthesizer converts resulting answers (strings of characters) into speech to the user.

Speech recognition and synthesis are domains in themselves that are outside the scope of this book. We will limit our discussion to the case of using speech application programming interfaces (API) to build a dialogue system. Given a speech signal, a speech recognition engine will carry out the transcription into words for us, and given a text, a speech synthesis engine will read it aloud.

Application programming interfaces are available from a variety of sources. See the section *Further Reading* (Sect. 17.8) for a short list of sources.

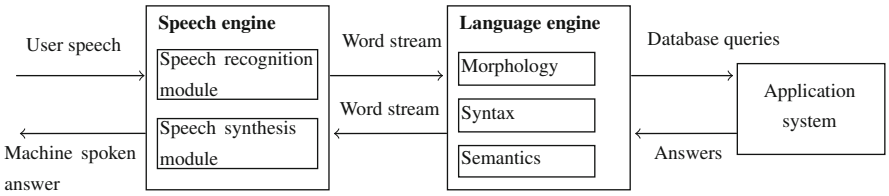


Fig. 17.1 Speech recognition and synthesis front ends

17.4 Simple Dialogue Systems

Speech recognition conditions are difficult with telephone servers since they have usually to handle a poor acoustic environment. It naturally comes at a price: recognition is prone to errors and the number of active words – words that the system can recognize at a given time – cannot be much more than a hundred on many systems. Speech recognition is then a bottleneck that limits the whole system performance.

For reasons of robustness and cost, operating components of real-world dialogue applications rarely correspond to the integration of classical linguistic layers: morphology, syntax, and semantics. The recognition itself does not attempt to produce the full stream of words but generally uses word-spotting techniques. Word spotting enables a word to be recognized within a short fragment of surrounding speech or noise. So a word will be correctly identified, even if you say *hmm* before or after it.

Because of word spotting, spoken systems do not stack a complete parsing after speech recognition. They focus on interesting words, meaningful according to context, and link them into information frames. These frames miss some words, but the important issue here is not to miss the overall meaning and to keep dialoguing with the user in real time and at a reasonable computational cost. Typical examples of such dialogue systems, elementary from a linguistic viewpoint, are based on automata.

17.4.1 Dialogue Systems Based on Automata

Dialogue systems based on finite-state automata have transitions triggered by a limited number of isolated words (Fig. 17.2). At each state, the automaton synthesizes a closed question: it proposes a finite choice of options. If the word recognition device has not understood a word, it loops onto the same state, asking for the user to repeat his/her command. If it corresponds to a legal transition, the automaton moves the user to another state.

As we said, speech recognition is not foolproof. The system avoids possible errors through a confirmation message while proceeding to a next state. On the leftmost edge of Fig. 17.2

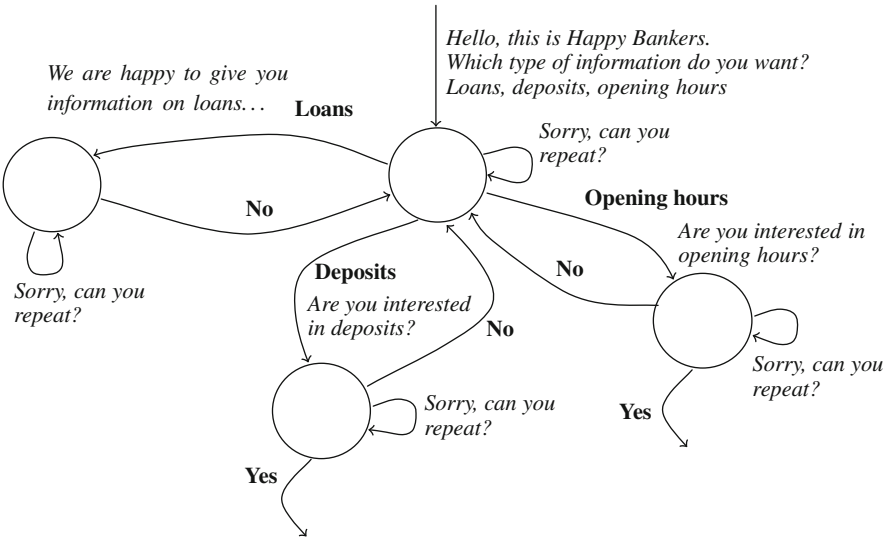


Fig. 17.2 An automaton for word-triggered dialogues

> We are happy to give you information on loans...

the system uses an implicit confirmation. The user can accept the ongoing transition either explicitly by saying *yes*, or implicitly by saying nothing. It can also contradict – only if necessary – and reject the transition by saying, for instance, *no* or

> No, I wanted to know about my account balance.

In this case, the user will regress to the previous state. Other edges as in the middle and rightmost transitions:

> Are you interested in deposits?

correspond to explicit confirmations. They require a mandatory answer – *yes* or *no* – from the user.

As a general design rule, confirmations should not be too numerous to be acceptable by users because they tend to be tedious if overused. The first strategy is certainly preferable.

17.4.2 Dialogue Modeling

The basic structure of dialogue automata is far from providing a natural interaction. However, it implements some fundamental characteristics shared by all dialogue systems. Interactions between the user and the system correspond to pairs: the system's turns and the user's turns. These pairs, which may be nested, have been extensively studied. Levinson (1983) proposed a classification of them according

Table 17.1 Classification of dialogue pairs

First member	Preferred second member	Dispreferred second member
Offer, Invitation	Acceptance	Refusal
Request	Compliance	Refusal
Assessment	Agreement	Disagreement
Question	Expected answer	Unexpected answer, no answer
Blame	Denial	Admission

Table 17.2 An exchange

Utt. no.	Turns	Utterances	Tags
1	S:	*Which type of information do you want: loans, deposits, opening hours?*	I_1
2	U:	*Loans*	R_1
3	S:	*We are happy to give you information on loans*	E_1

Table 17.3 An exchange with a nested evaluation exchange

Utt. no.	Turns	Utterances	Tags
1	S:	*Which type of information do you want: loans, deposits, opening hours?*	I_1
2	U:	*Deposits*	R_1
3	S:	*Are you interested in deposits?*	I_1^2
4	U:	*Yes*	R_1^2 E_1

to the nature of their first member. Table 17.1 shows an excerpt of Levinson's classification.

Levinson's model, however, is not sufficient to take possible errors and confirmation into account. Moeschler and others at the University of Geneva (Moeschler 1989; Moeschler and Reboul 1994) proposed a more elaborate model, which corrects some flaws. The model divides a dialogue as a sequence of exchanges, but it complements pairs with a final assessment. An exchange is a sequence of three different interventions:

- *initiative interventions*, which open an exchange (I)
- *reaction interventions*, which are answers to initiatives (R)
- *evaluation interventions*, which assess exchanges and possibly close them (E)

In addition, exchanges may be nested and hence have a recursive structure.

Table 17.2 shows a first exchange pictured by the leftmost edge of the automaton in Fig. 17.2. It is annotated with intervention tags: I, R, and E. The two first turns are an initiative and a reaction. The last turn is an implicit acknowledgment showing that the system has understood the user command.

Along with the deposit question (middle edge), Fig. 17.2 shows a nested interaction in the evaluation. There are first an initiative and a reaction. The third turn is an evaluation, which is a recursive exchange, consisting of an initiative and a reaction (Table 17.3).

17.5 Speech Acts: A Theory of Language Interaction

Turns triggered by isolated words and phrases are a rudimentary dialogue model. Ideally, systems should handle more complicated phrases or sentences. In addition, they should take more elaborate interaction structures into account.

Language is basically a means for people to act one upon the other. However, linguistic theories as we have considered them until now do not cover this interaction aspect. Some authors have tried to remedy this and to investigate other approaches. In contrast to formal or lexical semantics, which are based on logic, they have attempted to give language a more "performative" foundation. Such a framework has interested modern linguists such as Bühler (1934, 1982) and later language philosophers such as Austin (1962) and Searle (1969).

Bühler postulated that language had three semantic functions according to his organon model:

- A representation (*Darstellung*) of objects and the state of affairs that is being described,
- An expression (*Ausdruck*) materializing the psychological state of mind of the speaker – the sender of the message, and
- An appeal (*Appell*) corresponding to an effect on the hearer – the receiver of the message.

Although Bühler admitted the dominance of the representation function of language acknowledged before him, he stressed the psychological aspects of spoken communication describing participants as "psychophysical" systems. He was the first modern linguist to introduce that speech involved a sequence of acts that he named *Sprechakt*, enabling the hearer to recognize the speaker's state of mind or internal planning.

Austin came to a similar conclusion and also considered speech as a sequence of acts. For each of these acts, he distinguished what pertained to the classical side of linguistics and resorted to morphology, syntax, and semantics from pragmatics and the theory of action. He referred to the former as locutions and the latter as illocutions. From these considerations on, Austin modeled the act of saying something, with three components representing three different aspects of communication:

- *locutionary* – i.e., an act of saying something, corresponding to a phonetic utterance, a syntactic structure, and a formal semantics content
- *illocutionary* – i.e., a conversational act, which can be, for instance, to inform, to suggest, to answer, to ask
- *perlocutionary* – i.e., the effects of these acts, which can be to frighten, to worry, to convince, to persuade

Classical grammar recognizes certain links between locutionary and illocutionary content. Some types of syntactical forms are frequently associated with speech acts (Table 17.4).

Table 17.4 Syntactical forms and speech acts

Classical speech acts	Syntactic forms
Assertions, statements	Affirmatives or declaratives
Orders, commands	Imperatives
Questions	Interrogatives

Table 17.5 Conditions to *request*, *order*, *command* (After Searle (1969))

Conditions	Values
Propositional content	Future act *A* of Hearer
Preparatory	1. Hearer is able to do *A*. Speaker believes Hearer is able to do *A* 2. It is not obvious to both Speaker and Hearer that Hearer will do *A* in the normal course of events of his own accord 3. (For *order* and *command*) Speaker must be in a position of authority over Hearer
Sincerity	Speaker wants Hearer to do *A*
Essential	Counts as an attempt to get Hearer to do *A*

However, the association is not systematic. Speech acts are not always related to a logical – or propositional – content that could have been derived from the formal structure of sentences. Rhetorical questions such as *Can you open the door?* are in fact orders, and imperatives such as *Have a good day!* are greetings or wishes. In addition, a syntactical classification is too coarse to reflect the many needs of interaction analysis.

To cope with different aspects of communication, many authors have proposed a classification of illocutionary acts. We retain Searle's initial classes, which are best known because they probably capture essential interaction paradigms:

- **assertives**, such as stating, asserting, denying, informing.
- **directives**, such as requesting, asking, urging, commanding, ordering.
- **commissives**, such as promising, committing, threatening, consenting, refusing, offering.
- **declaratives**, such as declaring war, resigning, appointing, confirming, excommunicating. Declarative speech acts change states of affairs.
- **expressives**, which are related to emotions or feelings such as apologizing, thanking, protesting, boasting, complimenting.

Searle (1969) refines the speech act model by proposing conditions to complete an act successfully. Conditions are a set of conversational postulates that should be shared by speakers and hearers. These conditions are divided into a propositional content, a preparatory condition, a sincerity, and an essential condition. Tables 17.5 and 17.6 reproduce two success conditions to speech acts (Searle 1969, pp. 66–67).

The work of Austin and Searle has been very popular in the computational linguistics community and, far beyond it, in certain fields of philosophy and

Table 17.6 Conditions to *greeting* (After Searle (1969))

Conditions	Values
Propositional content	None
Preparatory	Speaker has just encountered (or has been introduced to, etc.) Hearer
Sincerity	None
Essential	Counts as courteous recognition of Hearer by Speaker

psychology. Although some people consider them as inventors, their findings are not completely new. Gorgias, a Greek rhetorician who lived 2,500 years before them, wrote:

> The effect of speech upon the condition of the soul is comparable to the power of drugs over the nature of bodies. For just as different drugs dispel different secretions from the body, and some bring an end to disease and others to life, so also in the case of speeches, some distress, others delight, some cause fear, others make the hearers bold, and some drug and bewitch the soul with a kind of evil persuasion.
>
> *Encomium of Helen* (Trans. RK Sprague)

17.6 Speech Acts and Human–Machine Dialogue

17.6.1 Speech Acts as a Tagging Model

Many language processing applications use the speech act theory as a kind of syntax to parse a discourse or a dialogue. Constituents are the discourse segments, and categories are illocution classes, termed broadly as speech acts or dialogue acts. As a result, a discourse is a sequence of segments annotated with conversation acts.

Authors may not follow the Searle's classification. Gazdar and Mellish (1989) provide a small set of "illocutionary acts," among which they quote: request, statement, suggestion, question. Using these acts, they can label the dialogue in Table 17.7.

Acts such as challenge or concession may be more suited to analyzing a human conversation rather than a spoken human–machine interaction. In addition, applications may need different sorts of acts. Therefore, most sets of speech acts are designed for a specific dialogue system and are closely tied to it. Acts then serve as tags to annotate discourse segments. Although disputable from a theoretical viewpoint, this interpretation of speech acts as tags is used as a model for scores of human–machine dialogue systems. We examine one of them in the next section.

17.6.2 Speech Acts Tags Used in the SUNDIAL Project

Bilange (1992) and Cozannet (1992) list a collection of speech acts that they used in the SUNDIAL project (Table 17.8). The acts are divided into initiatives, reactions,

Table 17.7 Illocutionary acts in a dialogue (After Gazdar and Mellish (1989, p. 385))

Turns	Utterances	Illocutionary acts
A	I really think the automobile needs servicing	Statement
B	But we had done it recently	Challenge
A	No, not for two years...	Challenge Interruption
A	Incidentally did you hear that gas prices are about to double?	Concession

Table 17.8 Speech acts used in SUNDIAL (slightly modified)

Acts	System/ User act	Descriptions
Initiatives		
`request(P)`	`S`	Open question or request for the value of P
`yn_question(P, Val)`	`S`	Is value of `P` `Val`? Answer should be *yes* or *no*
`altern_question(P)`	`S`	Alternative question: *Vanilla or strawberry*?
`repeat(P)`	`S/U`	Repetition request
`inform(P)`	`S/U`	Inform of `P`
`recap(P)`	`S`	Recapitulation of solved problems
Reactions		
`answer(P, Val)`	`U`	Gives a value `Val` on the request of `P`
`select(P, Val)`	`U`	Gives a value `Val` on an alternative question on `P`
`accept(P, Val)`	`U`	Accept or confirm the value `Val` of `P`
`reject(P, Val)`	`U`	Reject the value `Val` of `P`
Evaluations		
`impl_valid(P, Val)`	`S`	Implicit validation of confirmation of the value `Val` of `P`
`correct(P, Val)`	`U`	Gives a new value `Val` to `P`

and evaluations following Moeschler's (1989) dialogue modeling. They are intended to enable a user to make a train ticket reservation by telephone.

Other projects such as VERBMOBIL use speech acts that are even more tied to the application (Jekat et al. 1995). VERBMOBIL provides a language support to an appointment system and its acts include `INTRODUCE_NAME`, `ACCEPT_DATE`, `REJECT_DATE`, `SUGGEST_SUPPORT_DATE`.

17.6.3 Dialogue Parsing

Dialogue applications, for example, speech servers, are aimed at answering relatively simple inquiries such as providing information on train timetables, airfares, or credit card authorizations. Their possibilities are generally well understood by users who call them and who do not expect to have a philosophical conversation with the system.

For this reason, in many applications it is possible to restrict a human–machine transaction to a dialogue opening, a negotiation where a user formulates a problem

Table 17.9 An example of dialogue with SUNDIAL

Turns	Utterances
S1	*Bonjour puis-je vous aider?* 'Hello, can I help you?'
U1	*Je veux une réservation de Londres à Paris* 'I want a reservation from London to Paris'
S2	*Londres Paris à quelle date voulez-vous voyager?* 'London Paris which date?'
U2	*Je veux voyager le 20 juin* 'I want to travel June 20'
S3	*Le 20 juin à quelle heure?* 'June 20 which time?'
U3	*À 17 heures 15* '5.15 pm'
S4	*17 heures 15?* '5.15 pm?'
U4	*Oui* 'Yes'
S5	*Le vol BA-123 part à 17 heures 15 et arrive à Charles-de-Gaulle à 18 heures 39, est-ce que cela vous convient?* 'Flight BA-123 departure 5.15 pm and arrival 6.39 pm at Charles-de-Gaulle airport. Is that OK for you?'
U5	*Oui* 'Yes'
S6	*Quel est votre nom?* 'What is your name?'
U6	*Luc Martin*
S7	*Luc Martin une réservation pour Luc Martin sur le vol BA-123 arrivant à Charles-de-Gaulle à 18 heures 39* 'Luc Martin a reservation for Luc Martin on flight BA-123 arriving at Charles-de-Gaulle at 6.39 pm'
U7	*Oui* 'Yes'
S8	*Votre réservation a été confirmée au revoir* 'Your reservation has been confirmed goodbye'
U8	*Au revoir* 'Goodbye'

and solves it with the system, and a closing. Using Moeschler's model, we can describe each of these parts as a sequence of exchanges where utterances are divided into initiatives (I_i), reactions (I_r), and evaluations (I_e).

Table 17.9 shows a dialogue example from Andry (1992), and Table 17.10 shows the derived structure of the negotiation part. Utterances come either from the user (u) or the system (s) and consist of one or more speech acts. Utterance S2

London Paris which date?

is split into two acts. The first one (S1a)

London Paris

Table 17.10 Intervention structure

Exch.	Interventions	Recursive interventions	Turns
	`Ii(s, [request])`		S1
E1	`Ir(u, [answer])`		U1
	`Ie(s, [impl_valid])`		S2a
	`Ii(s, [request])`		S2b
E2	`Ir(u, [answer])`		U2
	`Ie(s, [impl_valid])`		S3a
	`Ii(s, [request])`		S3b
E3	`Ir(u, [answer])`		U3
E3e		`Ie(s, [impl_valid])`	S4
		`Ir(u, [accept])`	U4
E4	`Ii(s, [recap, yn_question])`		S5a S5b
	`Ir(u, [accept])`		U5
	`Ii(s, [request])`		S6
E5	`Ir(u, [answer])`		U6
	`Ie(s, [impl_valid])`		S7a
	`Ii(s, [recap])`		S7b
E6	`Ir(u, [accept])`		U7
	`Ie(s, [impl_valid])`		S8

corresponds to an implicit confirmation that the system has understood the departure and arrival cities `Ie(s, [impl_valid])`. The second one (S2b)

which date

is an explicit question to the user `Ii(s, [request])`.

We can parse the exchange in Table 17.10 and get its structure using DCG rules. We first write a grammar to model the nonrecursive exchanges. We use variables to unify the speaker – user or system – and the type of act.

```
exchange(ex(i(X, SA1), r(Y, SA2), e(E))) -->
  initiative([X, SA1]),
  reaction([Y, SA2]),
  evaluation(E),
  {X \= Y}.
exchange(ex(i(X, SA1), r(Y, SA2))) -->
  initiative([X, SA1]),
  reaction([Y, SA2]),
  {X \= Y}.
exchange(ex(e(X, SA1), r(Y, SA2))) -->
  evaluation([X, SA1]),
  reaction([Y, SA2]),
  {X \= Y}.
```

We model initiatives, reactions, and evaluations as a sequence of speech acts:

```
initiative([Speaker, SpeechActs]) -->
```

Table 17.11 Syntactic forms or templates linking utterances to speech acts

Syntactic features	Candidate speech acts
Interrogative sentence	`yn_question`, `altern_question`, `request`
yes, right, all right, OK	`accept`, `impl_valid`
no, not at all	`reject`
Declarative sentence	`inform`, `impl_valid`
sorry, pardon, can you repeat	`repeat`
not X but Y, that's not X it's Y in fact.	`correct`

```
    acts([Speaker, SpeechActs]).

reaction([Speaker, SpeechActs]) -->
    acts([Speaker, SpeechActs]).

evaluation([Speaker, SpeechActs]) -->
    acts([Speaker, SpeechActs]).
```

To take the recursive exchange into account, we have to add:

```
evaluation(S) --> exchange(S).
```

Finally, we define the dialogue as a sequence of exchanges:

```
dialogue([SE | RS]) --> exchange(SE), dialogue(RS).
dialogue([]) --> [].
```

Although these rules do not completely implement Moeschler's model, they give an insight to it.

17.6.4 Interpreting Speech Acts

To complete our dialogue survey, we outline ways to map utterances to speech acts, that is, in our example above, to annotate *What is your name?* as an open question. Some words, phrases, or syntactic features have a correspondence in terms of speech acts, as shown in Table 17.11. A first method is then to spot these specific patterns or cues. Cues enable us to delimit segments, to generate candidate speech acts, and to annotate the corresponding segment content. Once segments are identified, we can proceed to parse them and obtain their logical form.

However, identification is not straightforward because of ambiguity. Some words or syntactic features have more than one speech act candidate. The interrogative mode usually corresponds to questions, but not always, as in *Can you do that for me?*, which is likely to be a polite order in a human conversation. The system then produces several possible acts for each utterance or sequence of utterances: *Yes* in

the dialogue in Table 17.9 is either an acceptation (U5) or an implicit validation (S4).

The identification of speech acts for unrestricted dialogues has not received a definitive solution yet. However, it has attracted much attention and reasonably good solutions for applications like speech servers. In a spoken dialogue, what matters are the user's acts that an automatic system identifies using a tagging procedure. Tagsets can be relatively generic, as in the SUNDIAL project, or application-oriented, as in VERBMOBIL. DAMSL (Allen and Core 1997) is another oft-cited tagset.

Speech act tagging uses statistical approaches or reasoning rules, or possibly a combination of both. While many systems used by speech servers are based on rules, Alexandersson (1996) describes a statistical technique similar to those used in part-of-speech tagging. He uses a dialogue corpus where the turns are annotated with illocutionary acts instead of parts of speech. The tagger is a hidden Markov model, and the training procedure derives dialogue act n-grams. As for part-of-speech tagging, the stochastic dialogue act tagging consists in finding the most likely sequence of tags given a sequence of words and features.

As general principles, the features that rules or statistical modeling take into account are:

- Cue words or phrases, which may be linked to specific speech acts.
- The syntactic and semantic forms of the utterance.
- Expectations to apply constraints on possible speech acts. These are based on transitions from a previous state to the current state of the dialogue: when the system asks a question, it expects an answer, a rejection or a failure, and it can discard other acts.
- Task modeling and goal satisfaction. This point extends the previous one. It restrains possible user acts and parameter values according to the progress point where the user is in the dialogue.

17.6.5 EVAR: A Dialogue Application Using Speech Acts

EVAR – *Erkennen, Verstehen, Antworten, Rückfragen* – is a dialogue system intended to provide information on train schedules in Germany (Mast 1993; Mast et al. 1994). It gives an example of a thorough task modeling that enables the system to restrain the number of possible dialogue acts at a given point of the inquiry process. It avoids random initiatives from the user thanks to a constant guidance process. EVAR structures the dialogue process in a sequence of phases:

- A greeting by the system
- A request from the user
- A clarification consisting of a confirmation of data requested by the user and possible requests for details and specifications
- An answer from the system
- A closing

Table 17.12 A dialogue in the EVAR prototype (After Mast et al. (1994))

Turns	Dialogue	Phases
S:	*Guten Tag. Hier ist die automatische Intercity-Auskunft. Welche Information benötigen Sie?* 'Hello. This is the Automatic Travel Information System. What information do you need?'	Greeting
U:	*Ich möchte morgen nach Hamburg fahren.* 'Tomorrow I want to go to Hamburg.'	Request
S:	*Sie möchten morgen nach Hamburg fahren?* 'Tomorrow you want to go to Hamburg?'	Confirmation
U:	*Ja morgen.* 'Yes tomorrow.'	
S:	*Von wo möchten Sie abfahren?* 'Where do you want to start?'	Request for details
U:	*München* 'Munich'	
S:	*Morgen von München nach Hamburg* 'Tomorrow from Munich to Hamburg'	
U:	*Ja* 'Yes'	
S:	*Wann möchten Sie fahren?* 'When do you want to leave?'	Request for specification
U:	*Morgen früh* 'Early in the morning'	
S:	*Morgen früh von München nach Hamburg* 'Tomorrow morning from Munich to Hamburg'	
U:	*Ja* 'Yes'	
S:	*Ich habe mehere Verbindungen gefunden. Nach Hamburg HBF, fahren Sie ab München um 7 Uhr. Ankunft in Hamburg HBF um 15 Uhr, . . .* 'I found a number of connections. From Munich departure 7, Arrival Hamburg 15, . . . '	Answer
U:	*Vielen Dank* 'Thank you'	
S:	*Möchten Sie weitere Auskünfte?* 'Do you want further information?'	Closing
U:	*Nein danke* 'No thank you'	
S:	*Auf wiederhören.* 'Have a nice trip.'	

EVAR enables transactions such as those in Table 17.12.

The finite state automaton in Fig. 17.3 models the dialogue progress. The `S_` prefix denotes a system's turn, and `U_` denotes a user's turn.

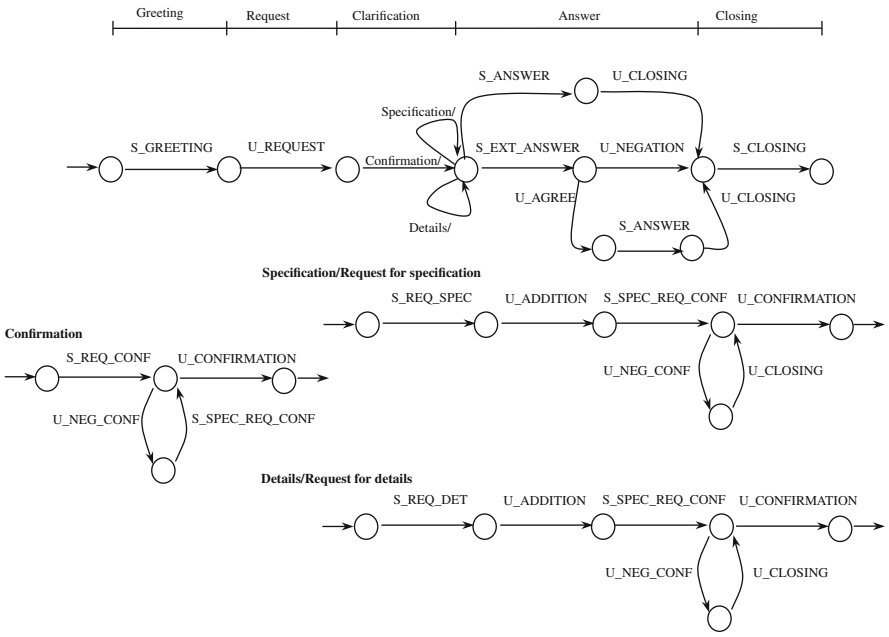


Fig. 17.3 The EVAR dialogue model (After Mast et al. (1994))

17.7 Taking Beliefs and Intentions into Account

The models of dialogue we have examined so far have an external viewpoint in the sense that they are based on observations of the user's behavior by the system. Parallel to them, some authors tried to take into account the user's beliefs, desires, and intentions. They hoped to build an internal model of the user and thus to gain a deeper insight into dialogue processes.

The ambition to model beliefs and intentions is appealing because it addresses concerns and questions that often puzzle people: *What does he want from me? What does she mean?* Knowing or extracting a user's intentions would certainly help a computer serve the user better. In addition, intention modeling recasts dialogue and interaction into a framework more general than other formalisms.

However, such a model may prove difficult to implement. It is first a philosophical challenge to figure out and describe what the beliefs of individuals are. Second, if beliefs or intentions could be completely described, it would be a second challenge to box them into a program and run them with a computer. Fortunately, most dialogue applications have a goal that is plainly prosaic, and simplifications can be made. We describe now a classical representation of user modeling introduced by Allen and Perrault (1980).

Table 17.13 Belief spaces

Patron's belief space	Waiter's belief space
$\exists x, cassoulet(x)$	
$\exists x, pytt\_i\_panna(x)$	$\exists x, pytt\_i\_panna(x)$
$\exists x, yorkshire\_pudding(x)$	$\exists x, yorkshire\_pudding(x)$

17.7.1 Representing Mental States

The idea of conversational systems based on belief, desire, and intention is to model the participants as processes – agents. The agents are human users as well as artificial ones, and each agent has a specific knowledge and the desire to complete an action. The agent's core is a representation of their mental states, which uses predicates aimed at describing their beliefs or knowledge spaces, and what they can do. Agents are modeled using operators such as:

- `want(A, X)`, which means that agent A wants to do X
- `can_do(A, X)`, which means that agent A can do X
- `believe(A, X)`, which means that agent A believes X
- `know(A, X)`, which means that agent A knows X

Since beliefs are personal, that is, individual, the definition of truth is no longer universal. For this reason, predicates have two arguments, the agent who is the believer and the proposition that is believed or known. This nonuniversal logic is called modal, and refers to the various modes of truth.

From these operators, some axioms can be derived such as:

```
(know(A, X), (X ⇒ Y)) ⇒ know(A, Y)
(believe(A, X), (X ⇒ Y)) ⇒ believe(A, Y)
(believe(A, X), X) ⇒ know(A, X)
```

Mental states can be different according to dialogue participants, whether they involve human beings together or humans and machines. Let us suppose that a patron goes to a restaurant, looks at the menu, and sees as main courses *cassoulet*, *pytt i panna*, and *Yorkshire pudding*. Let us also suppose that the restaurant is running out of *cassoulet*. When entering the restaurant, the belief spaces of the patron and the waiter are different (Table 17.13).

A short dialogue between the waiter and the patron when ordering the meal will enable them to synchronize their belief spaces (Table 17.14).

Patron: *I feel like a cassoulet*
Waiter: *Sorry sir, we have no more of it.*

Such an exchange is also called a grounding – that is, setting a common ground. Grounding is central to dialogue system design. The user must be sure that beliefs and knowledge are shared between her/him and the system. If not, misunderstanding would creep into many exchanges.

Table 17.14 Belief spaces after dialogue

Patron's belief space	Waiter's belief space
$\exists x, pytt\_i\_panna(x)$	$\exists x, pytt\_i\_panna(x)$
$\exists x, yorkshire\_pudding(x)$	$\exists x, yorkshire\_pudding(x)$

Table 17.15 Beliefs in Prolog

Patron's belief space	Waiter's belief space
`believe(patron('Pierre'), cassoulet(X))`	
`believe(patron('Pierre'), pytt_i_panna(X))`	`believe(waiter('Bill'), pytt_i_panna(X))`
`believe(patron('Pierre'), yorkshire_pudding(X))`	`believe(waiter('Bill'), yorkshire_pudding(X))`

Table 17.16 Intentions in Prolog

Patron's intentions	Waiter's intentions
`intend(patron('Pierre'), (cassoulet(X), order(X)))`	`intend(waiter('Bill'), take_order(X))`

Mutual beliefs can be expressed as `believe(A, P)` $\wedge$ `believe(B, P)` $\wedge$ `believe(A, believe(B, P))` $\wedge$ `believe(B, believe(A, P))` $\wedge$ `believe(A, believe(B, believe(A, P)))`, etc. Such a infinite conjunction is denoted `mutually_believe(A, B, P)`.

Mutual beliefs should not be explicitly listed all the time at the risk of being tedious. Most of the time, the user knows that there is an artificial system behind the box and expects something very specific from it. However, the system has to make sure the user is aware of its knowledge and beliefs, for instance, using implicit confirmation each time the user provides information.

Representing the corresponding beliefs and intentions using Prolog is straightforward (Tables 17.15 and 17.16)

Finally, modal operators can be used to transcribe speech acts. For instance, the act of informing can be associated to the operator `inform(A, B, P)` (`A` informs `B` of `P`), which will be applied with the following preconditions and effects:

- Preconditions: `know(A, P)`, `want(A, inform(A, B, P))`
- Effects: `know(B, P)`

The operator `request(A, B, P)` can be modeled as:

- Preconditions: `want(A, request(A, B, P))`, `believe(A, can_do(B, P))`
- Effects: `believe(A, want(B, P))`

17.7.2 The STRIPS Planning Algorithm

Mental state consistency is usually controlled using a planning algorithm. Planning has been extensively studied, and we introduce here the STRIPS algorithm (Fikes and Nilsson 1971; Nilsson 1998). STRIPS considers planning as a search problem given an initial and a final state. It uses rules describing an action – corresponding here to the operators – with preconditions and postconditions. Postconditions are divided into an add and a delete list, reflecting facts new to the world and facts to be removed.

```
%strips_rule(+operator, +preconditions, +add_list,
%  +delete_list).

strips_rule(inform(A, B, P), [know(A, P), want(A,
  inform(A, B, P))], [believe(B, P)], []).
```

Mental states or world state are described by lists of facts. The `apply/3` predicate applies an operator to a `State` that results in a `NewState`, subtracting facts to be deleted and adding facts to be added:

```
% apply(+Action, +State, -NewState)

apply(Action, State, NewState):-
  strips_rule(Action, _, _, DeleteList),
  subtract(State, DeleteList, TempState),
  strips_rule(Action, _, AddList, _),
  union(AddList, TempState, NewState).
```

where `subtract/3` and `union/3` are predicates built-in in most Prologs. They define set subtraction and union. `subtract(+Set, +Delete, -Result)` deletes all elements of list `Delete` from list `Set`, resulting in `Result`. `union(+Set1, +Set2, -Result)` makes the union of `Set1` and `Set2`, removing duplicates and resulting in `Result`.

STRIPS represents possible states as nodes of a graph (Fig. 17.4). Each node consists of a list of facts. The actions enable movement from one node to another, adding or deleting facts when the preconditions are met. Knowing an initial and a final list of facts representing the initial state and the goal, the problem is stated as finding the action plan that modifies the world, adding or deleting facts so that the initial state is transformed into the final one. This is a search problem, where STRIPS traverses a graph to find the plan actions as follows (Nilsson 1998, pp. 376–379).

- Repeat while `Goals` are not a subset of the current `State`,
  1. Select a `Goal` from the `Goals` that is not already in the current `State`.
  2. Find a STRIP rule whose `Action` adds `Goal` to the current `State` and make sure that `Action` has not already been done.

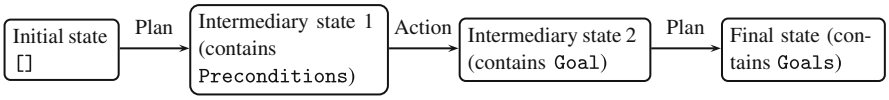


Fig. 17.4 The STRIPS schemata

3. `Action` may not be possible if the `Preconditions` are not met. Therefore, use STRIPS to solve recursively `Preconditions`. This results in the intermediary state 1 `(InterState1)`.
4. Apply `Action` to add and delete facts from the intermediary state 1. This results in the intermediary state 2 `(InterState2)`.
5. Recursively solve the rest of `Goals`.

STRIPS in Prolog uses the initial state and the goals as inputs and produces the final state and the plan, where the goals must be a subset of the final state. We need some auxiliary variables to carry out the computation. The Prolog recursive rule builds the `Plan` in a reverse order, adding the current `Action` to the head of the list, and unifies it to the `FinalPlan` variable when the goal is satisfied. Prolog also builds `State`, and unifies it to `FinalState` in a same way. To avoid possible infinite loops when finding preconditions to an action, the rule keeps a copy of the corresponding plan and prohibits the repetition of actions.

```
strips(Goals, InitState, FinState, Plan) :-
  strips(Goals, InitState, [],[], FinState, RevPlan),
  reverse(RevPlan, Plan).

% strips(+Goals, +State, +Plan, +PrecondPlan,
%  -FinalState, -FinalPlan)

strips(Goals, State, Plan, _, State, Plan) :-
  subset(Goals, State).
strips(Goals, State, Plan, PrecondPlan, FinalState,
    FinalPlan) :-
  member(Goal, Goals),    %Select a goal
  \+ member(Goal, State),
  strips_rule(Action, _, AddList, _), %Find an action
  member(Goal, AddList),
  \+ member(Action, PrecondPlan),
      % Find preconditions
  strips_rule(Action, Preconditions, _, _),
  % Get the FirstPlan and InterState1
  % to achieve preconditions
  strips(Preconditions, State, Plan,
    [Action | PrecondPlan], InterState1, FirstPlan),
  % Apply Action to the world
```

```
    apply(Action, InterState1, InterState2),
    % From FirstPlan move forward
    strips(Goals, InterState2, [Action | FirstPlan],
      PrecondPlan, FinalState, FinalPlan).
```

17.7.3 Causality

Planning provides a good operation model for dialogue and for other discourse phenomena such as **causality**. Causality occurs when some sentences are logically chained and are part of a same demonstration. Consider, for instance, these sentences:

Hedgehogs are back. Caterpillars shiver.

The second sentence is a consequence of the first. Causality can be related to a logical demonstration but also depends on time-ordered events. Causal rules represent specific events, which will result in certain facts or effects. Usually, they also require preconditions. They can be expressed in Prolog using predicates whose structure is similar to:

```
causes(preconditions, event, effects).
```

As we can see, this is also closely related to planning. The `causes` predicate means that if the `preconditions` are met, and if an `event` occurs, then we will have `effects`.

Many sentences involve sequences of actions – plans – that are temporally chained. For instance:

Phileas the hedgehog was thirsty. He went out to have a pint.

These two sentences correspond to an action, which is followed by another, the second one being a consequence of the first one. For both examples, discourse understanding can be restated as a plan recognition problem.

17.8 Further Reading

Many dialogue systems rely on spoken input and output. Speech engines for recognition as well as for synthesis for desktops and servers are available from multiple vendors. They include Microsoft, Apple, Nuance, and Google. The Web Speech API is an example of an application programming interface that uses JavaScript and is dedicated to web browsers, while Android speech uses Java and is intended for cellular telephones. Both APIs are supported by Google. Microsoft Tellme is another example mainly intended for speech servers. It uses VoiceXML, a language to specify simple dialogues. Open-source engines include

the Cambridge Hidden Markov Toolkit (Cambridge HTK, http://htk.eng.cam.ac.uk/) from the University of Cambridge for speech recognition and Festival from the University of Edinburgh for speech synthesis (http://www.cstr.ed.ac.uk/projects/festival/).

Speech acts theory in dialogue is mostly known from the works of Austin (1962) and Searle (1969, 1979), although Bühler (1982) pioneered it. Searle and Vanderveken (1985) describe a logical model of illocutionary acts as well as a list of English verbs classified according to Searle's ontology. Vanderveken (1988) expands this work to French verbs. Foundations of belief and intention modeling in dialogue are due to Hintikka (1962). Carberry (1990) provides accounts to plan recognition in dialogue.

EVAR and the SUNDIAL projects have been a source of valuable research and publications about spoken dialogue processing. Bilange (1992) gives an excellent overview of dialogue processing techniques and application examples. Other works include those of Andry (1992), Mast (1993), Eckert (1996), and Sagerer (1990). The TRAINS project (Allen et al. 1995) is another example of elaborate dialogue processing. Many applications, such as train reservation systems, are now available commercially.

Planning includes a large number of applications and has spurred many algorithms. In computational linguistics, it occurs within the frameworks of temporal reasoning, intention modeling, and other forms of constraint-based reasoning. Bratko (2012) gives a short introduction to planning and a collection of Prolog programs. Russell and Norvig (2010) provides another introduction to planning.

Exercises

17.1. Write a dialogue program using Prolog clauses – no DCG rules – asking a couple of questions and accepting yes or no answers only. Collect all the answers and print them out at the end of the session.

17.2. Write a dialogue program using Prolog clauses – no DCG rules – reproducing the dialogue of Fig. 17.2. Collect all the answers and print them out at the end of the session.

17.3. Write verbs in a language you know corresponding to Searle's ontology of illocutionary classes: assertives, directives, commissives, declaratives, and expressives.

17.4. Rewrite Exercise 17.1 using SUNDIAL's speech act predicates in Table 17.8.

17.5. Rewrite Exercise 17.2 using SUNDIAL's speech act predicates in Table 17.8.

17.6. The DCG dialogue rules described in Sect. 17.6.3 are not robust. Make a parallel with sentence parsing and give examples where they would fail and why.

17.7. Modify the rules of Sect. 17.6.3 so that they would never fail, but recover and start again.

17.8. Write an automaton in Prolog to model EVAR's main phases accepting a legal sequence of speech acts, such as `[S_GREETING, U_REQ_INFO, ...]`.

17.9. Modify the EVAR automaton of Exercise 17.8 to be interactive. Design questions and messages from the system and possible answers from the user. Replace the user and system turns with them.

17.10. Modify the EVAR automaton of Exercises 17.8 and 17.9 and use the SUNDIAL speech acts. Make the system work so that you have a more or less realistic dialogue.

Appendix A
An Introduction to Prolog

A.1 A Short Background

Prolog was designed in the 1970s by Alain Colmerauer and a team of researchers with the idea – new at that time – that it was possible to use logic to represent knowledge and to write programs. More precisely, Prolog uses a subset of predicate logic and draws its structure from theoretical works of earlier logicians such as Herbrand (1930) and Robinson (1965) on the automation of theorem proving.

Prolog was originally intended for the writing of natural language processing applications. Because of its conciseness and simplicity, it became popular well beyond this domain and now has adepts in areas such as:

- Formal logic and associated forms of programming
- Reasoning modeling
- Database programming
- Planning, and so on.

This chapter is a short review of Prolog. In-depth tutorials include: in English, Bratko (2012), Clocksin and Mellish (2003), Covington et al. (1997), and Sterling and Shapiro (1994); in French, Giannesini et al. (1985); and in German, Baumann (1991). Boizumault (1988, 1993) contain a didactical implementation of Prolog in Lisp. Prolog foundations rest on first-order logic. Apt (1997), Burke and Foxley (1996), Delahaye (1986), and Lloyd (1987) examine theoretical links between this part of logic and Prolog.

Colmerauer started his work at the University of Montréal, and a first version of the language was implemented at the University of Marseilles in 1972. Colmerauer and Roussel (1996) tell the story of the birth of Prolog, including their try-and-fail experimentation to select tractable algorithms from the mass of results provided by research in logic.

In 1995, the International Organization for Standardization (ISO) published a standard on the Prolog programming language. Standard Prolog (Deransart et al. 1996) is becoming prevalent in the Prolog community and most of the available

P.M. Nugues, *Language Processing with Perl and Prolog*, Cognitive Technologies,
DOI 10.1007/978-3-642-41464-0, © Springer-Verlag Berlin Heidelberg 2014

implementations now adopt it, either partly or fully. Unless specifically indicated, descriptions in this chapter conform to the ISO standard, and examples should run under any Standard Prolog implementation.

A.2 Basic Features of Prolog

A.2.1 Facts

Facts are statements that describe object properties or relations between objects. Let us imagine we want to encode that Ulysses, Penelope, Telemachus, Achilles, and others are characters of Homer's *Iliad* and *Odyssey*. This translates into Prolog facts ended with a period:

```
character(priam, iliad).
character(hecuba, iliad).
character(achilles, iliad).
character(agamemnon, iliad).
character(patroclus, iliad).
character(hector, iliad).
character(andromache, iliad).
character(rhesus, iliad).
character(ulysses, iliad).
character(menelaus, iliad).
character(helen, iliad).

character(ulysses, odyssey).
character(penelope, odyssey).
character(telemachus, odyssey).
character(laertes, odyssey).
character(nestor, odyssey).
character(menelaus, odyssey).
character(helen, odyssey).
character(hermione, odyssey).
```

Such a collection of facts, and later, of rules, makes up a **database**. It transcribes the knowledge of a particular situation into a logical format. Adding more facts to the database, we express other properties, such as the gender of characters:

```
% Male characters          % Female characters

male(priam).               female(hecuba).
male(achilles).            female(andromache).
male(agamemnon).           female(helen).
male(patroclus).           female(penelope).
```

```
male(hector).
male(rhesus).
male(ulysses).
male(menelaus).
male(telemachus).
male(laertes).
male(nestor).
```

or relationships between characters such as parentage:

```
% Fathers                          % Mothers
father(priam, hector).             mother(hecuba, hector).
father(laertes, ulysses).          mother(penelope, telemachus).
father(atreus, menelaus).          mother(helen, hermione).
father(menelaus, hermione).
father(ulysses, telemachus).
```

Finally, would we wish to describe kings of some cities and their parties, this would be done as:

```
king(ulysses, ithaca, achaean).
king(menelaus, sparta, achaean).
king(nestor, pylos, achaean).
king(agamemnon, argos, achaean).
king(priam, troy, trojan).
king(rhesus, thrace, trojan).
```

From these examples, we understand that the general form of a Prolog fact is: `relation(object1, object2, ..., objectn)`. Symbols or names representing objects, such as `ulysses` or `penelope`, are called **atoms**. Atoms are normally strings of letters, digits, or underscores "_", and begin with a lowercase letter. An atom can also be a string beginning with an uppercase letter or including white spaces, but it must be enclosed between quotes. Thus `'Ulysses'` or `'Pallas Athena'` are legal atoms.

In logic, the name of the symbolic `relation` is the **predicate**, the objects `object1`, `object2`, ..., `objectn` involved in the relation are the **arguments**, and the number `n` of the arguments is the **arity**. Traditionally, a Prolog predicate is indicated by its name and arity: `predicate/arity`, for example, `character/2`, `king/3`.

A.2.2 Terms

In Prolog, all forms of data are called **terms**. The constants, i.e., atoms or numbers, are terms. The fact `king(menelaus, sparta, achaean)` is a **compound term** or a **structure**, that is, a term composed of other terms – **subterms**. The

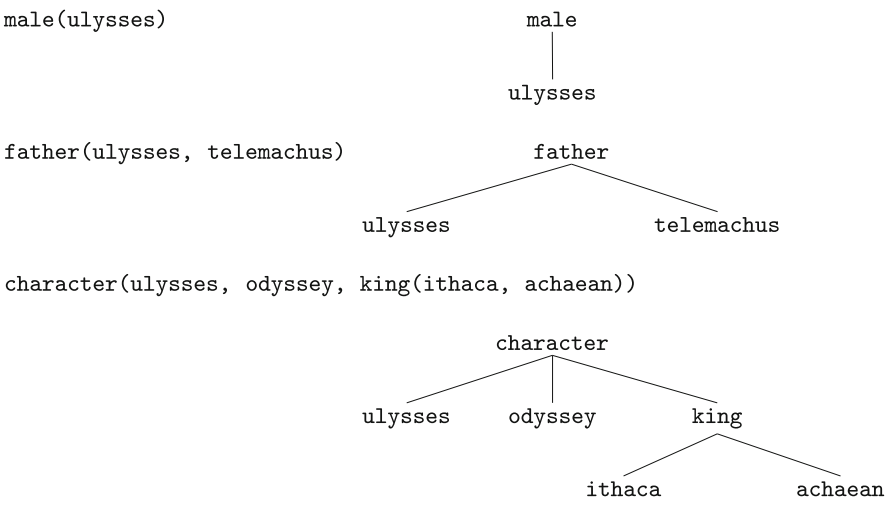


Fig. A.1 Graphical representations of terms

arguments of this compound term are constants. They can also be other compound terms, as in

```
character(priam, iliad, king(troy, trojan)).
character(ulysses, iliad, king(ithaca, achaean)).
character(menelaus, iliad, king(sparta, achaean)).
```

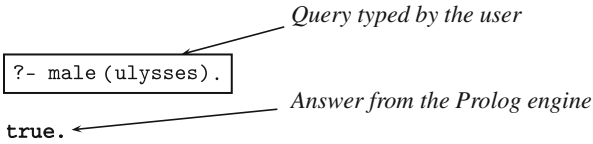
where the arguments of the predicate `character/3` are two atoms and a compound term.

It is common to use trees to represent compound terms. The nodes of a tree are then equivalent to the functors of a term. Figure A.1 shows examples of this.

Syntactically, a compound term consists of a **functor** – the name of the relation – and arguments. The leftmost functor of a term is the **principal functor**. A same principal functor with a different arity corresponds to different predicates: `character/3` is thus different from `character/2`. A constant is a special case of a compound term with no arguments and an arity of 0. The constant `abc` can thus be referred to as `abc/0`.

A.2.3 Queries

A query is a request to prove or retrieve information from the database, for example, if a fact is true. Prolog answers yes if it can prove it, that is, here if the fact is in the database, or no if it cannot: if the fact is absent. The question *Is Ulysses a male*? corresponds to the query:



which has a positive answer. A same question with Penelope would give:

```
?- male(penelope).
false.
```

because this fact is not in the database.

The expressions `male(ulysses)` or `male(penelope)` are **goals** to prove. The previous queries consisted of single goals. Some questions require more goals, such as *Is Menelaus a male and is he the king of Sparta and an Achaean?*, which translates into:

```
?- male(menelaus), king(menelaus, sparta, achaean).
true.
```

where "`,`" is the conjunction operator. It indicates that Prolog has to prove both goals. The simple queries have one goal to prove, while the **compound queries** are a conjunction of two or more goals:

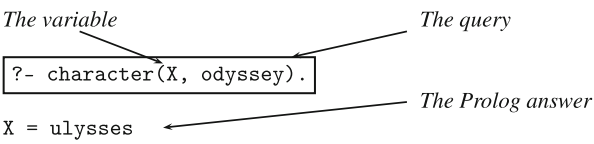
```
?- G1, G2, G3, ..., Gn.
```

Prolog proves the whole query by proving that all the goals `G1` ... `Gn` are true.

A.2.4 Logical Variables

The logical variables are the last kind of Prolog terms. Syntactically, variables begin with an uppercase letter, for example, `X`, `Xyz`, or an underscore "`_`". Logical variables stand for any term: constants, compound terms, and other variables. A term containing variables such as `character(X, Y)` can unify with a compatible fact, such as `character(penelope, odyssey)`, with the **substitutions** `X = penelope` and `Y = odyssey`.

When a query term contains variables, the Prolog resolution algorithm searches terms in the database that unify with it. It then substitutes the variables to the matching arguments. Variables enable users to ask questions such as *What are the characters of the Odyssey?*



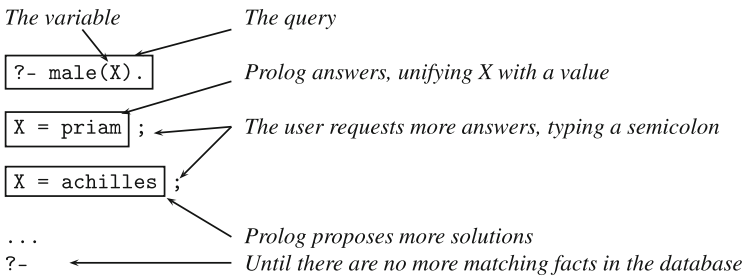
Or *What is the city and the party of king Menelaus?* etc.

```
?- king(menelaus, X, Y).
X = sparta, Y = achaean

?- character(menelaus, X, king(Y, Z)).
X = iliad, Y = sparta, Z = achaean

?- character(menelaus, X, Y).
X = iliad, Y = king(sparta, achaean)
```

When there are multiple solutions, Prolog considers the first fact to match the query in the database. The user can type "`;`" to get the next answers until there is no more solution. For example:



A.2.5 Shared Variables

Goals in a conjunctive query can share variables. This is useful to constrain arguments of different goals to have the same value. To express the question *Is the king of Ithaca also a father?* in Prolog, we use the conjunction of two goals `king(X, ithaca, Y)` and `father(X, Z)`, where the variable `X` is shared between the goals:

```
?- king(X, ithaca, Y), father(X, Z).
X = ulysses, Y = achaean, Z = telemachus
```

In this query, we are not interested in the name of the child although Prolog responds with `Z = telemachus`. We can indicate to Prolog that we do not need to know the values of `Y` and `Z` using **anonymous variables**. We then replace `Y` and `Z` with the symbol "`_`", which does not return any value:

```
?- king(X, ithaca, _), father(X, _).
X = ulysses
```

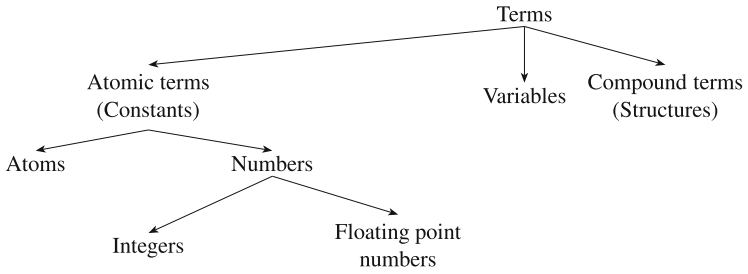


Fig. A.2 Kinds of terms in Prolog

A.2.6 Data Types in Prolog

To sum up, every data object in Prolog is a term. Terms divide into atomic terms, variables, and compound terms (Fig. A.2).

Syntax of terms may vary according to Prolog implementations. You should consult reference manuals for their specific details. Here is a list of simplified conventions from Standard Prolog (Deransart et al. 1996):

- Atoms are sequences of letters, numbers, and/or underscores beginning with a lowercase letter, such as `ulysses`, `iSLanD3`, `king_of_Ithaca`.
- Some single symbols, called solo characters, are atoms: `! ;`
- Sequences consisting entirely of some specific symbols or graphic characters are atoms: `+ - * / ^ < = > ~ : . ? @ # $ & \ '`
- Any sequence of characters enclosed between single quotes is also an atom, such as `'king of Ithaca'`. A quote within a quoted atom must be double quoted: `'I''m'`
- Numbers are either decimal integers, such as `-19`, `1960`, octal integers when preceded by `0o`, as `0o56`, hexadecimal integers when preceded by `0x`, as `0xF4`, or binary integers when preceded by `0b`, as `0b101`.
- Floating-point numbers are digits with a decimal point, as `3.14`, `-1.5`. They may contain an exponent, as `23E-5` ($23\ 10^{-5}$) or `-2.3e5` ($2.3\ 10^{-5}$).
- The ASCII numeric value of a character `x` is denoted `0'x`, as `0'a` (97), `0'b` (98), etc.
- Variables are sequences of letters, numbers, and/or underscores beginning with an uppercase letter or the underscore character.
- Compound terms consist of a functor, which must be an atom, followed immediately by an opening parenthesis, a sequence of terms separated by commas, and a closing parenthesis.

Finally, Prolog uses two types of comments:

- Line comments go from the "`%`" symbol to the end of the line:
  `% This is a comment`

- Multiline comments begin with a "`/*`" and end with a "`*/`":

```
/*
this
is
a comment */
```

A.2.7 Rules

Rules enable us to derive a new property or relation from a set of existing ones. For instance, the property of being the son of somebody corresponds to either the property of having a father and being a male, or having a mother and being a male. Accordingly, the Prolog predicate `son(X, Y)` corresponds either to conjunction `male(X), father(Y, X)`, or to `male(X), mother(Y, X)`. Being a son admits thus two definitions that are transcribed as two Prolog rules:

```
son(X, Y) :- father(Y, X), male(X).
son(X, Y) :- mother(Y, X), male(X).
```

More formally, rules consist of a term called the **head** or consequent, followed by symbol "`:-`", read if, and a conjunction of goals. They have the form:

```
HEAD :- G1, G2, G3, ... Gn.
```

where the conjunction of goals is the **body** or antecedent of the rule. The head is true if the body is true. Variables of a rule are shared between the body and the head. Rules can be queried just like facts:

```
?- son(telemachus, Y).
Y = ulysses;
Y = penelope;
?-
```

Rules are a flexible way to deduce new information from a set of facts. The `parent/2` predicate is another example of a family relationship that is easy to define using rules. Somebody is a parent if s/he is either a mother or a father:

```
parent(X, Y) :- mother(X, Y).
parent(X, Y) :- father(X, Y).
```

Rules can call other rules as with `grandparent/2`. A grandparent is the parent of a parent and is defined in Prolog as

```
grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
```

where `Z` is an intermediate variable shared between goals. It enables us to find the link between the grandparent and the grandchild: a mother or a father.

We can generalize the `grandparent/2` predicate and write `ancestor/2`. We use two rules, one of them being recursive:

```
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
```

This latter pattern is quite common for Prolog rules. One or more rules express a general case using recursion. Another set of rules or facts describes simpler conditions without recursion. They correspond to boundary cases and enable the recursion to terminate.

A query about the ancestors of Hermione yields:

```
?- ancestor(X, hermione).
X = helen;
X = menelaus;
X = atreus;
false.
?-
```

Facts and rules are also called **clauses**. A predicate is defined by a set of clauses with the same principal functor and arity. Facts are indeed special cases of rules: rules that are always true and `relation(X, Y)` is equivalent to `relation(X, Y) :- true`, where `true/0` is a built-in predicate that always succeeds. Most Prolog implementations require clauses of the same name and arity to be grouped together.

In the body of a rule, the comma "`,`" represents a conjunction of goals. It is also possible to use a disjunction with the operator "`;`". Thus:

```
A :-
  B
  ;
  C.
```

is equivalent to

```
A :- B.
A :- C.
```

However, "`;`" should be used scarcely because it impairs somewhat the legibility of clauses and programs. The latter form is generally better.

A.3 Running a Program

The set of facts and rules of a file makes up a **Prolog text** or program. To run it and use the information it contains, a Prolog system has to load the text and add it to the current database in memory. Once Prolog is launched, it displays a prompt symbol "`?-`" and accepts commands from the user.

Ways to load a program are specific to each Prolog implementation. A user should look them up in the reference manual because the current standard does not define them. There are, however, two commands drawn from the Edinburgh Prolog tradition (Pereira 1984) implemented in most systems: `consult/1` and `reconsult/1`.

The predicate `consult/1` loads a file given as an argument and adds all the clauses of the file to the current database in memory:

```
?- consult(file_name).
```

`file_name` must be an atom as, for example,

```
?- consult('odyssey.pl').
```

It is also possible to use the shortcut:

```
?- [file_name].
```

to load one file, for example,

```
?- ['odyssey.pl'].
```

or more files:

```
?- [file1, file2].
```

The predicate `reconsult/1` is a variation of `consult`. Usually, a programmer writes a program, loads it using `consult`, runs it, debugs it, modifies the program, and reloads the modified program until it is correct. While `consult` adds the modified clauses to the old ones in the database, `reconsult` updates the database instead. It loads the modified file and replaces clauses of existing predicates in the database by new clauses contained in the file. If a predicate is in the file and not in the database, `reconsult` simply adds its clauses. In some Prolog systems, `reconsult` does not exist, and `consult` discards existing clauses to replace them by the new definition from the loaded file. Once a file is loaded, the user can run queries.

The `listing/0` built-in predicate displays all the clauses in the database, and `listing/1`, the definition of a specific predicate. The `listing/1` argument format is either `Predicate` or `Predicate/Arity`:

```
?- listing(character/2).
character(priam, iliad).
character(hecuba, iliad).
character(achilles, iliad).
...
```

A program can also include directives, i.e., predicates to run at load time. A directive is a rule without a head: a term or a conjunction of terms with a "`:-`" symbol to its left-hand side:

```
:- predicates_to_execute.
```

Directives are run immediately as they are encountered. If a directive is to be executed once the program is completely loaded, it must occur at the end of the file.

Finally, `halt/0` quits Prolog.

A.4 Unification

A.4.1 Substitution and Instances

When Prolog answers a query made of a term T containing variables, it applies a **substitution**. This means that Prolog replaces variables in T by values so that it proves T to be true. The substitution {`X = ulysses, Y = odyssey`} is a solution to the query `character(X, Y)` because the fact `character(ulysses, odyssey)` is in the database. In the same vein, the substitution {`X = sparta, Y = achaean`} is a solution to the query `king(menelaus, X, Y)`.

More formally, a substitution is a set {`X1 = t1, X2 = t2, ..., Xn = tn`}, where `Xi` is a variable and `ti` is a term. Applying a substitution σ to a term T is denoted $T\sigma$ and corresponds to the replacement of all the occurrences of variable `Xi` with term `ti` in T for `i` ranging from `1` to `n`. Applying the (meaningless) substitution σ_1 = {`X = ulysses`} to the term `T1 = king(menelaus, X, Y)` yields `T1' = king(menelaus, ulysses, Y)`. Applying the substitution σ_2 = {`X = iliad, Y = king(sparta, achaean)`} to the term `T2 = character(menelaus, X, Y)` yields `T2' = character(menelaus, iliad, king(sparta, achaean))`.

A term T' resulting from a substitution $T\sigma$ is an **instance** of T. More generally, T' is an instance of T if there is a substitution so that $T' = T\sigma$. If T' is an instance of T, then T is **more general** than T'. Terms can be ordered according to possible compositions of instantiations. For example, `character(X, Y)` is more general than `character(ulysses, odyssey)`; `king(X, Y, Z)` is more general than `king(menelaus, Y, Z)`, which is more general than `king(menelaus, Y, achaean)`, which is itself more general than `king(menelaus, sparta, achaean)`.

A substitution mapping a set of variables onto another set of variables such as σ = {`X = A, Y = B`} onto term `character(X, Y)` is a **renaming substitution**. Initial and resulting terms `character(X, Y)` and `character(A, B)` are said to be **alphabetical variants**. Finally, a **ground** term is a term that contains no variable such as `king(menelaus, sparta, achaean)`.

A.4.2 Terms and Unification

To equate two terms, `T1` and `T2`, Prolog uses unification, which substitutes variables in the terms so that they are identical. Unification is a logical mechanism that carries out a two-way matching, from `T1` to `T2` and the reverse, and merges them into a

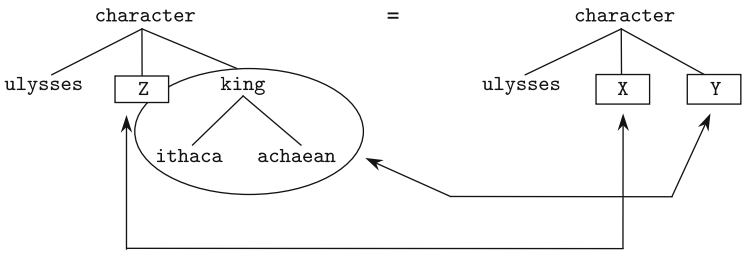


Fig. A.3 Unification of terms: a graphical interpretation

common term. Prolog unifies terms to solve equations such as `T1 = T2`. It also uses unification in queries to match a goal or a subgoal to the head of the rule. Figure A.3 shows the intuitive unification of terms

```
T1 = character(ulysses, Z, king(ithaca, achaean))
```

and

```
T2 = character(ulysses, X, Y)
```

through a graphical superposition.

The superposition of the two terms requires finding an instance common to both terms T_1 and T_2. This can be restated as there exist two substitutions σ_1 and σ_2 such that $T_1\sigma_1 = T_2\sigma_2$. A **unifier** is a substitution making T_1 and T_2 identical: $T_1\sigma = T_2\sigma$. In our example, there is an infinite number of possible unifiers. Candidates include the substitution σ = {`Z = c(a), X = c(a), Y = king(ithaca, achaean)`}, which yields the common instance: `character(ulysses,c(a), king(ithaca, achaean))`. They also include σ = {`Z = female, Z = female, Y = king(ithaca, achaean)`}, which yields another common instance: `character(ulysses, female, king(ithaca, achaean))`, etc.

Intuitively, these two previous unifiers are special cases of the unification of `T1` and `T2`. In fact, all the unifiers are instances of the substitution σ = {`X = Z, Y = king(ithaca, achaean)`}, which is the **most general unifier** or **MGU**.

Using SWI-Prolog to unify `T1` and `T2`, we obtain:

```
?- character(ulysses, Z, king(ithaca, achaean)) =
character(ulysses, X, Y).
Z = X, Y = king(ithaca, achaean).
```

A.4.3 The Herbrand Unification Algorithm

The reference algorithm to unify terms is due to Herbrand (1930) and Martelli and Montanari (1982). It takes the two terms to unify as input. The output is either a failure, if the terms do not unify or the MGU: σ.

The algorithm initializes the substitution to the empty set and pushes terms on a stack. The main loop consists in popping terms, comparing their functors, and pushing their arguments on the stack. When a variable is found, the corresponding substitution is added to σ (Deransart et al. 1996; Sterling and Shapiro 1994).

- **Initialization step**

  Initialize σ to {}
  Initialize `failure` to `false`
  Push the equation $T_1 = T_2$ on the stack

- **Loop**

  repeat {
    pop $x = y$ from the stack
    if x is a constant and $x == y$. Continue.
    else if x is a variable and x does not appear in y.
      Replace x with y in the stack and in σ. Add the substitution $\{x = y\}$ to σ.
    else if x is a variable and $x == y$. Continue.
    else if y is a variable and x is not a variable.
      Push $y = x$ on the stack.
    else if x and y are compounds with $x = f(x_1, ..., x_n)$ and $y = f(y_1, ..., y_n)$.
      Push on the stack $x_i = y_i$ for i ranging from 1 to n.
    else Set `failure` to `true`, and σ to {}. Break.
  } until (stack $\neq \emptyset$)

A.4.4 Example

Let us exemplify the Herbrand algorithm with terms: `f(g(X, h(X, b)), Z)` and `f(g(a, Z), Y)`. We will use a two-way stack: one for the left term and one for the right term, and let us scan and push term arguments from right to left.

For the first iteration of the loop, x and y are compounds. After this iteration, the stack looks like:

Left term of the stack (x)		**Right term of the stack** (y)
`g(X, h(X, b))`	=	`g(a, Z)`
`Z`	=	`Y`

with the substitution σ = { }.

The second iteration pops the top terms of the left and right parts of the stack. The loop condition corresponds to compound terms again. The algorithm pushes the arguments of left and right terms on the stack:

Left term of the stack (x)		**Right term of the stack** (y)
`X`	`=`	`a`
`h(X, b)`	`=`	`Z`
`Z`	`=`	`Y`

with the substitution $\sigma = \{\}$.

The third iteration pops the equation `X = a`. The algorithm adds this substitution to σ and carries out the substitution in the stack:

Left term of the stack (x)		**Right term of the stack** (y)
`h(X, b)` $\sim$ `h(a, b)`	`=`	`Z`
`Z`	`=`	`Y`

with the substitution $\sigma = \{$`X = a`$\}$.

The next iteration pops `h(a, b) = Z`, swaps the left and right terms, and yields:

Left term of the stack (x)		**Right term of the stack** (y)
`Z`	`=`	`h(a, b)`
`Z`	`=`	`Y`

The fifth iteration pops `Z = h(a, b)` and yields:

Left term of the stack (x)		**Right term of the stack** (y)
`Z` $\sim$ `h(a, b)`	`=`	`Y`

with the substitution $\sigma = \{$`X = a, Z = h(a, b)`$\}$.

Finally, we get the MGU $\sigma = \{$`X = a, Z = h(a, b), Y = h(a, b)`$\}$ that yields the unified term `f(g(a, h(a, b)), h(a, b))`.

A.4.5 The Occurs-Check

The Herbrand algorithm specifies that variables `X` or `Y` must not appear – occur – in the right or left member of the equation to be a successful substitution. The unification of `X` and `f(X)` should then fail because `f(X)` contains `X`.

However, most Prolog implementations do not check the occurrence of variables to keep the unification time linear on the size of the smallest of the terms being unified (Pereira 1984). Thus, the unification `X = f(X)` unfortunately succeeds resulting in a stack overflow. The term `f(X)` infinitely replaces `X` in σ, yielding `X = f(f(X))`, `f(f(f(X)))`, `f(f(f(f(X))))`, etc., until the memory is exhausted. It results in a system crash with many Prologs.

Although theoretically better, a unification algorithm that would implement an occurs-check is not necessary most of the time. An experienced programmer will not write unification equations with a potential occurs-check problem. That is why Prolog systems compromised the algorithm purity for speed. Should the occurs-check be necessary, Standard Prolog provides the `unify_with_occurs_check/2` built-in predicate:

```
?- unify_with_occurs_check(X, f(X)).
false.

?- unify_with_occurs_check(X, f(a)).
X = f(a)
```

A.5 Resolution

A.5.1 Modus Ponens

The Prolog resolution algorithm is based on the *modus ponens* form of inference that stems from traditional logic. The idea is to use a general rule – the major premise – and a specific fact – the minor premise – like the famous:

All men are mortal
Socrates is a man

to conclude, in this case, that

Socrates is mortal

Table A.1 shows the modus ponens in the classical notation of predicate logic and in Prolog.

Prolog runs a reversed modus ponens. Using symbols in Table A.1, Prolog tries to prove that a query (β) is a consequence of the database content (α, $\alpha \Rightarrow \beta$). Using the major premise, it goes from β to α, and using the minor premise, from α to true. Such a sequence of goals is called a **derivation**. A derivation can be finite or infinite.

A.5.2 A Resolution Algorithm

Prolog uses a resolution algorithm to chain clauses mechanically and prove a query. This algorithm is generally derived from Robinson's resolution principle (1965), known as the SLD resolution. SLD stands for "linear resolution" with a "selection function" for "definite clauses" (Kowalski and Kuehner 1971). Here "definite clauses" are just another name for Prolog clauses.

Table A.1 The modus ponens notation in formal logic and its Prolog equivalent

	Formal notation	**Prolog notation**
Facts	α	`man('Socrates').`
Rules	$\alpha \Rightarrow \beta$	`mortal(X) :- man(X).`
Conclusion	β	`mortal('Socrates').`

The resolution takes a program – a set of clauses, rules, and facts – and a query `Q` as an input (Deransart et al. 1996; Sterling and Shapiro 1994). It considers a conjunction of current goals to prove, called the **resolvent**, that it initializes with `Q`. The resolution algorithm selects a goal from the resolvent and searches a clause in the database so that the head of the clause unifies with the goal. It replaces the goal with the body of that clause. The resolution loop replaces successively goals of the resolvent until they all reduce to true and the resolvent becomes empty. The output is then a success with a possible instantiation of the query goal `Q'`, or a failure if no rule unifies with the goal. In case of success, the final substitution, σ, is the composition of all the MGUs involved in the resolution restricted to the variables of `Q`. This type of derivation, which terminates when the resolvent is empty, is called a **refutation**.

- **Initialization**

  Initialize `Resolvent` to `Q`, the initial goal of the resolution algorithm.
  Initialize σ to {}
  Initialize `failure` to `false`

- **Loop with `Resolvent = G1, G2, ..., Gi, ..., Gm`**

  while (`Resolvent` $\neq \emptyset$) {

  1. Select the goal `Gi` $\in$ `Resolvent`;
  2. If `Gi == true`, delete it and continue;
  3. Select the rule `H :- B1, ..., Bn` in the database such that `Gi` and `H` unify with the MGU θ. If there is no such rule, then set `failure` to `true`; break;
  4. Replace `Gi` with `B1, ..., Bn` in `Resolvent`
     % `Resolvent = G1,...,Gi-1, B1,...,Bn, Gi+1,..., Gm`
  5. Apply θ to `Resolvent` and to `Q`;
  6. Compose σ with θ to obtain the new current σ;

  }

Each goal in the resolvent – i.e., in the body of a rule – must be different from a variable. Otherwise, this goal must be instantiated to a nonvariable term before it is called. The `call/1` built-in predicate then executes it as in the rule:

```
daughter(X, Y) :-
  mother(Y, X), G = female(X), call(G).
```

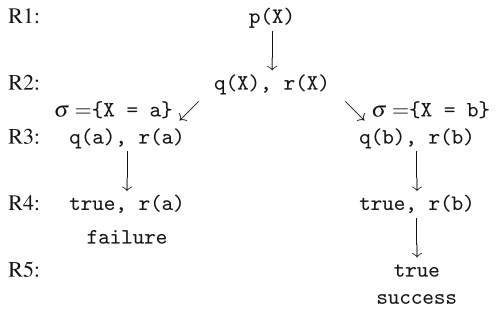


Fig. A.4 The search tree and successive values of the resolvent

where `call(G)` solves the goal `G` just as if it were `female(X)`. In fact, Prolog automatically inserts `call/1` predicates when it finds that a goal is a variable. `G` is thus exactly equivalent to `call(G)`, and the rule can be rewritten more concisely in:

```
daughter(X, Y) :-
  mother(Y, X), G = female(X), G.
```

A.5.3 Derivation Trees and Backtracking

The resolution algorithm does not tell us how to select a goal from the resolvent. It also does not tell how to select a clause in the program. In most cases, there is more than one choice. The selection order of goals is of no consequence because Prolog has to prove all of them anyway. In practice, Prolog considers the leftmost goal of the resolvent. The selection of the clause is more significant because some derivations lead to a failure although a query can be proved by other derivations. Let us show this with the program:

```
p(X) :- q(X), r(X).
q(a).
q(b).
r(b).
r(c).
```

and the query `?- p(X).`

Let us compute the possible states of the resolvent along with the resolution's iteration count. The first resolvent (R1) is the query itself. The second resolvent (R2) is the body of `p(X):q(X), r(X)`; there is no other choice. The third resolvent (R3) has two possible values because the leftmost subgoal `q(X)` can unify either with the facts `q(a)` or `q(b)`. Subsequently, according to the fact selected and the corresponding substitution, the derivation succeeds or fails (Fig. A.4).

The Prolog resolution can then be restated as a search, and the picture of successive states of the resolvent as a search tree. Now how does Prolog select a

clause? When more than one is possible, Prolog could expand the resolvent as many times as there are clauses. This strategy would correspond to a breadth-first search. Although it gives all the solutions, this is not the one Prolog employs because it would be unbearable in terms of memory.

Prolog uses a depth-first search strategy. It scans clauses from top to bottom and selects the first one to match the leftmost goal in the resolvent. This sometimes leads to a subsequent failure, as in our example, where the sequence of resolvents is first `p(X)`, then the conjunction `q(X), r(X)`, after that `q(a), r(a)`, and finally the goal `r(a)`, which is not in the database. Prolog uses a backtracking mechanism then. During a derivation, Prolog keeps a record of backtrack points when there is a possible choice, that is, where more than one clause unifies with the current goal. When a derivation fails, Prolog backs up to the last point where it could select another clause, undoes the corresponding unification, and proceeds with the next possible clause. In our example, it corresponds to resolvent R2 with the second possible unification: `q(b)`. The resolvent R3 is then `q(b), r(b)`, which leads to a success. Backtracking explores all possible alternatives until a solution is found or it reaches a complete failure.

However, although the depth-first strategy enables us to explore most search trees, it is only an approximation of a complete resolution algorithm. In some cases, the search path is infinite, even when a solution exists. Consider the program:

```
p(X) :- p(X), q(X).
p(a).
q(a).
```

where the query `p(a)` does not succeed because of Prolog's order of rule selection. Fortunately, most of the time there is a workaround. Here it suffices to invert the order of the subgoals in the body of the rule.

A.6 Tracing and Debugging

Bugs are programming errors, that is, when a program does not do what we expect from it. To isolate and remove them, the programmer uses a **debugger**. A debugger enables programmers to trace the goal execution and unification step by step. It would certainly be preferable to write bug-free programs, but to err is human. And debugging remains, unfortunately, a frequent part of program development.

The Prolog debugger uses an execution model describing the control flow of a goal (Fig. A.5). It is pictured as a box representing the goal predicate with four ports, where:

- The Call port corresponds to the invocation of the goal.
- If the goal is satisfied, the execution comes out through the Exit port with a possible unification.
- If the goal fails, the execution exits through the Fail port.

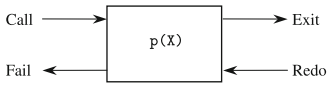


Fig. A.5 The execution model of Prolog

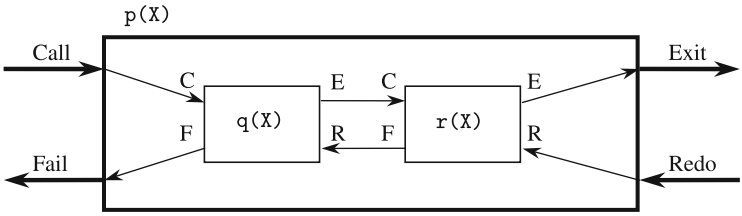


Fig. A.6 The execution box representing the rule `p(X) :- q(X), r(X)`

- Finally, if a subsequent goal fails and Prolog backtracks to try another clause of the predicate, the execution re-enters the box through the Redo port.

The built-in predicate `trace/0` launches the debugger and `notrace/0` stops it. The debugger may have different commands according to the Prolog system you are using. Major ones are:

- `creep` to proceed through the execution ports. Simply type return to creep.
- `skip` to skip a goal giving the result without examining its subgoals. (Type `s` to skip).
- `retry` starts the current goal again from an exit or redo port (type `r`).
- `fail` makes a current goal to fail (type `f`).
- `abort` to quit the debugger (type `a`).

Figure A.6 represents the rule `p(X) :- q(X), r(X)`, where the box corresponding to the head encloses a chain of subboxes picturing the conjunction of goals in the body. The debugger enters goal boxes using the `creep` command.

As an example, let us trace the program:

```
p(X) :- q(X), r(X).
q(a).
q(b).
r(b).
r(c).
```

with the query `p(X)`.

```
?- trace.
true.
?- p(X).
   Call:  (  7) p(_G106) ? creep
   Call:  (  8) q(_G106) ? creep
   Exit:  (  8) q(a) ? creep
```

```
    Call:  (  8) r(a) ? creep
    Fail:  (  8) r(a) ? creep
    Redo:  (  8) q(_G106) ? creep
    Exit:  (  8) q(b) ? creep
    Call:  (  8) r(b) ? creep
    Exit:  (  8) r(b) ? creep
    Exit:  (  7) p(b) ? creep
X = b
```

A.7 Cuts, Negation, and Related Predicates

A.7.1 Cuts

The cut predicate, written "`!`", is a device to prune some backtracking alternatives. It modifies the way Prolog explores goals and enables a programmer to control the execution of programs. When executed in the body of a clause, the cut always succeeds and removes backtracking points set before it in the current clause. Figure A.7 shows the execution model of the rule `p(X) :- q(X), !, r(X)` that contains a cut.

Let us suppose that a predicate `P` consists of three clauses:

```
P :- A1, ..., Ai, !, Ai+1, ..., An.
P :- B1, ..., Bm.
P :- C1, ..., Cp.
```

Executing the cut in the first clause has the following consequences:

1. All other clauses of the predicate below the clause containing the cut are pruned. That is, here the two remaining clauses of `P` will not be tried.
2. All the goals to the left of the cut are also pruned. That is, A_1, ..., A_i will no longer be tried.
3. However, it will be possible to backtrack on goals to the right of the cut.

```
P :- A1, ..., Ai, !, Ai+1, ..., An.
P :- B1, ..., Bm.
P :- C1, ..., Cp.
```

Cuts are intended to improve the speed and memory consumption of a program. However, wrongly placed cuts may discard some useful backtracking paths and solutions. Then, they may introduce vicious bugs that are often difficult to track. Therefore, cuts should be used carefully.

An acceptable use of cuts is to express determinism. Deterministic predicates always produce a definite solution; it is not necessary then to maintain backtracking possibilities. A simple example of it is given by the minimum of two numbers:

```
minimum(X, Y, X) :- X < Y.
minimum(X, Y, Y) :- X >= Y.
```

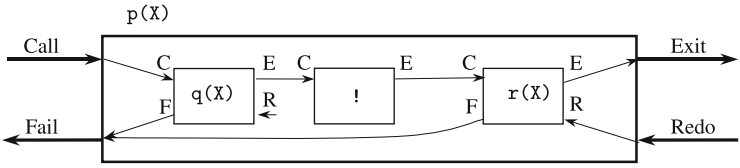


Fig. A.7 The execution box representing the rule `p(X) :- q(X), !, r(X)`

Once the comparison is done, there is no means to backtrack because both clauses are mutually exclusive. This can be expressed by adding two cuts:

```
minimum(X, Y, X) :- X < Y, !.
minimum(X, Y, Y) :- X >= Y, !.
```

Some programmers would rewrite `minimum/3` using a single cut:

```
minimum(X, Y, X) :- X < Y, !.
minimum(X, Y, Y).
```

The idea behind this is that once Prolog has compared `X` and `Y` in the first clause, it is not necessary to compare them again in the second one. Although the latter program may be more efficient in terms of speed, it is obscure. In the first version of `minimum/3`, cuts respect the logical meaning of the program and do not impair its legibility. Such cuts are called **green cuts**. The cut in the second `minimum/3` predicate is to avoid writing a condition explicitly. Such cuts are error-prone and are called **red cuts**. Sometimes red cuts are crucial to a program but when overused, they are a bad programming practice.

A.7.2 Negation

A logic program contains no negative information, only queries that can be proven or not. The Prolog built-in negation corresponds to a query failure: the program cannot prove the query. The negation symbol is written "`\+`" or `not` in older Prolog systems:

- If `G` succeeds then `\+ G` fails.
- If `G` fails then `\+ G` succeeds.

The Prolog negation is defined using a cut:

```
\+(P) :- P, !, fail.
\+(P) :- true.
```

where `fail/0` is a built-in predicate that always fails.

Most of the time, it is preferable to ensure that a negated goal is ground: all its variables are instantiated. Let us illustrate it with the somewhat odd rule:

```
mother(X, Y) :- \+ male(X), child(Y, X).
```

and facts:

```
child(telemachus, penelope).
male(ulysses).
male(telemachus).
```

The query

```
?- mother(X, Y).
```

fails because the subgoal `male(X)` is not ground and unifies with the fact `male(ulysses)`. If the subgoals are inverted:

```
mother(X, Y) :- child(Y, X), \+ male(X).
```

the term `child(Y, X)` unifies with the substitution `X = penelope` and `Y = telemachus`, and since `male(penelope)` is not in the database, the goal `mother(X, Y)` succeeds.

Predicates similar to "\+" include if-then and if-then-else constructs. If-then is expressed by the built-in `'->'/2` operator. Its syntax is

```
Condition -> Action
```

as in

```
print_if_parent(X, Y) :-
  (parent(X, Y) -> write(X), nl, write(Y), nl).

?- print_if_parent(X, Y).
penelope
telemachus

X = penelope, Y = telemachus
```

Just like negation, `'->'/2` is defined using a cut:

```
'->'(P, Q):- P, !, Q.
```

The if-then-else predicate is an extension of `'->'/2` with a second member to the right. Its syntax is

```
Condition -> Then ; Else
```

If `Condition` succeeds, `Then` is executed, otherwise `Else` is executed.

A.7.3 The `once/1` Predicate

The built-in predicate `once/1` also controls Prolog execution. `once(P)` executes `P` once and removes backtrack points from it. If `P` is a conjunction of goals as in the rule:

```
A :- B1, B2, once((B3, ..., Bi)), Bi+1, ..., Bn.
```

the backtracking path goes directly from B_{i+1} to B_2, skipping B_3, ..., B_i. It is necessary to bracket the conjunction inside `once` twice because its arity is equal to one. A single level of brackets, as in `once(`B_3`, ..., `B_i`)`, would tell Prolog that `once/1` has an arity of `i-3`.

`once(Goal)` is defined as:

```
once(Goal) :- Goal, !.
```

A.8 Lists

Lists are data structures essential to many programs. A Prolog list is a sequence of an arbitrary number of terms separated by commas and enclosed within square brackets. For example:

- `[a]` is a list made of an atom.
- `[a, b]` is a list made of two atoms.
- `[a, X, father(X, telemachus)]` is a list made of an atom, a variable, and a compound term.
- `[[a, b], [[[father(X, telemachus)]]]]` is a list made of two sublists.
- `[]` is the atom representing the empty list.

Although it is not obvious from these examples, Prolog lists are compound terms and the square bracketed notation is only a shortcut. The list functor is a dot: "`./2`", and `[a, b]` is equivalent to the term `.(a, .(b, []))`.

Computationally, lists are recursive structures. They consist of two parts: a head, the first element of a list, and a tail, the remaining list without its first element. The head and the tail correspond to the first and second argument of the Prolog list functor. Figure A.8 shows the term structure of the list `[a, b, c]`. The tail of a list is possibly empty as in `.(c, [])`.

The notation "`|`" splits a list into its head and tail, and `[H | T]` is equivalent to `.(H, T)`. Splitting a list enables us to access any element of it and therefore it is a very frequent operation. Here are some examples of its use:

```
?- [a, b] = [H | T].
H = a, T = [b]
```

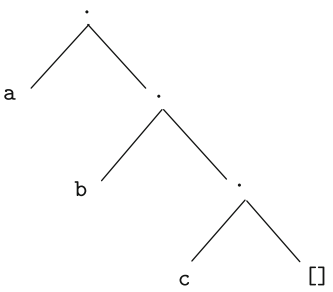


Fig. A.8 The term structure of the list `[a, b, c]`

```
?- [a] = [H | T].
H = a, T = []

?- [a, [b]] = [H | T].
H = a, T = [[b]]

?- [a, b, c, d] = [X, Y | T].
X = a, Y = b, T = [c, d]

?- [[a, b, c], d, e] = [H | T].
H = [a, b, c], T = [d, e]
```

The empty list cannot be split:

```
?- [] = [H | T].
false.
```

A.9 Some List-Handling Predicates

Many applications require extensive list processing. This section describes some useful predicates. Generally, Prolog systems provide a set of built-in list predicates. Consult your manual to see which ones; there is no use in reinventing the wheel.

A.9.1 The `member/2` Predicate

The `member/2` predicate checks whether an element is a member of a list:

```
?- member(a, [b, c, a]).
true.

?- member(a, [c, d]).
false.
```

`member/2` is defined as

```
member(X, [X | Y]).      % Termination case
member(X, [Y | YS]) :-   % Recursive case
  member(X, YS).
```

We could also use anonymous variables to improve legibility and rewrite `member/2` as

```
member(X, [X | _]).
member(X, [_ | YS]) :- member(X, YS).
```

`member/2` can be queried with variables to generate elements member of a list, as in:

```
?- member(X, [a, b, c]).
X = a ;
X = b ;
X = c ;
?-
```

Or lists containing an element:

```
?- member(a, Z).
Z = [a | Y] ;
Z = [Y, a | X] ;
etc.
```

Finally, the query:

```
?- \+ member(X, L).
```

where `X` and `L` are ground variables, returns `true` if `member(X, L)` fails and `false` if it succeeds.

A.9.2 The `append/3` Predicate

The `append/3` predicate appends two lists and unifies the result to a third argument:

```
?- append([a, b, c], [d, e, f], [a, b, c, d, e, f]).
true.

?- append([a, b], [c, d], [e, f]).
false.
```

```
?- append([a, b], [c, d], L).
L = [a, b, c, d]

?- append(L, [c, d], [a, b, c, d]).
L = [a, b]

?- append(L1, L2, [a, b, c]).
L1 = [], L2 = [a, b, c] ;
L1 = [a], L2 = [b, c] ;
```

etc., with all the combinations.

`append/3` is defined as

```
append([], L, L).
append([X | XS], YS, [X | ZS]) :-
  append(XS, YS, ZS).
```

A.9.3 The `delete/3` Predicate

The `delete/3` predicate deletes a given element from a list. Its synopsis is: `delete(List, Element, ListWithoutElement)`. It is defined as:

```
delete([], _, []).
delete([E | List], E, ListWithoutE):-
  !,
  delete(List, E, ListWithoutE).
delete([H | List], E, [H | ListWithoutE]):-
  H \= E,
  !,
  delete(List, E, ListWithoutE).
```

The three clauses are mutually exclusive, and the cuts make it possible to omit the condition `H \= E` in the second rule. This improves the program efficiency but makes it less legible.

A.9.4 The `intersection/3` Predicate

The `intersection/3` predicate computes the intersection of two sets represented as lists: `intersection(InputSet1, InputSet2, Intersection)`.

```
?- intersection([a, b, c], [d, b, e, a], L).
L = [a, b]
```

`InputSet1` and `InputSet2` should be without duplicates; otherwise `intersection/3` approximates the intersection set relatively to the first argument:

```
?- intersection([a, b, c, a], [d, b, e, a], L).
L = [a, b, a]
```

The predicate is defined as:

```
% Termination case
intersection([], _, []).
% Head of L1 is in L2
intersection([X | L1], L2, [X | L3]) :-
  member(X, L2),
  !,
  intersection(L1, L2, L3).
% Head of L1 is not in L2
intersection([X | L1], L2, L3) :-
  \+ member(X, L2),
  !,
  intersection(L1, L2, L3).
```

As for `delete/3`, clauses of `intersection/3` are mutually exclusive, and the programmer can omit the condition `\+ member(X, L2)` in the third clause.

A.9.5 The `reverse/2` Predicate

The `reverse/2` predicate reverses the elements of a list. There are two classic ways to define it. The first definition is straightforward but consumes much memory. It is often called the naïve reverse:

```
reverse([],[]).
reverse([X | XS], YS) :-
  reverse(XS, RXS),
  append(RXS, [X], YS).
```

A second solution improves the memory consumption. It uses a third argument as an accumulator.

```
reverse(X, Y) :-
  reverse(X, [], Y).

reverse([], YS, YS).
reverse([X | XS], Accu, YS):-
  reverse(XS, [X | Accu], YS).
```

A.9.6 The Mode of an Argument

The **mode** of an argument defines if it is typically an input (`+`) or an output (`-`). Inputs must be instantiated, while outputs are normally uninstantiated. Some predicates have multiple modes of use. We saw three modes for `append/3`:

- `append(+List1, +List2, +List3)`,
- `append(+List1, +List2, -List3)`, and
- `append(-List1, -List2, +List3)`.

A question mark "`?`" denotes that an argument can either be instantiated or not. Thus, the two first modes of `append/3` can be compacted into

```
append(+List1, +List2, ?List3).
```

The actual mode of `append/3`, which describes all possibilities is, in fact,

```
append(?List1, ?List2, ?List3).
```

Finally, "`@`" indicates that the argument is normally a compound term that shall remain unaltered.

It is good programming practice to annotate predicates with their common modes of use.

A.10 Operators and Arithmetic

A.10.1 Operators

Prolog defines a set of prefix, infix, and postfix operators that includes the classical arithmetic symbols: "`+`", "`-`", "`*`", and "`/`". The Prolog interpreter considers operators as functors and transforms expressions into terms. Thus, `2 * 3 + 4 * 2` is equivalent to `+(*(2, 3), *(4, 2))`.

The mapping of operators onto terms is governed by rules of priority and classes of associativity:

- The priority of an operator is an integer ranging from 1 to 1,200. It enables us to determine recursively the principal functor of a term. Higher-priority operators will be higher in the tree representing a term.
- The associativity determines the bracketing of term `A op B op C`:
  1. If `op` is left-associative, the term is read `(A op B) op C`;
  2. If `op` is right-associative, the term is read `A op (B op C)`.

Prolog defines an operator by its name, its **specifier**, and its priority. The specifier is a mnemonic to denote the operator class of associativity and whether it is infixed, prefixed, or postfixed (Table A.2).

Table A.2 Operator specifiers

Operator	Nonassociative	Right-associative	Left-associative
Infix	*xfx*	*xfy*	*yfx*
Prefix	*fx*	*fy*	–
Postfix	*xf*	–	*yf*

Table A.3 Priority and specifier of operators in Standard Prolog

Priority	Specifier	Operators
1,200	`xfx`	`:- -->`
1,200	`fx`	`:- ?-`
1,100	`xfy`	`;`
1,050	`xfy`	`->`
1,000	`xfy`	`','`
900	`fy`	`\+`
700	`xfx`	`= \=`
700	`xfx`	`== \== @< @=< @> @>=`
700	`xfx`	`=..`
700	`xfx`	`is =:= =\= < =< > >=`
550	`xfy`	`:`
500	`yfx`	`+ - # /\ \/`
400	`yfx`	`* / // rem mod << >>`
200	`xfx`	`**`
200	`xfy`	`^`
200	`fy`	`+ - \`

Table A.3 shows the priority and specifier of predefined operators in Standard Prolog.
It is possible to declare new operators using the directive:

```
:- op(+Priority, +Specifier, +Name).
```

A.10.2 Arithmetic Operations

The evaluation of an arithmetic expression uses the `is/2` built-in operator. `is/2` computes the value of the `Expression` to the right of it and unifies it with `Value`:

```
?- Value is Expression.
```

where `Expression` must be computable. Let us exemplify it. Recall first that "=" does not evaluate the arithmetic expression:

```
?- X = 1 + 1 + 1.
X = 1 + 1 + 1 (or X = +(+(1, 1), 1)).
```

To get a value, it is necessary to use `is`

```
?- X = 1 + 1 + 1, Y is X.
X = 1 + 1 + 1, Y = 3.
```

If the arithmetic expression is not valid, `is/2` returns an error, as in

```
?- X is 1 + 1 + a.
Error
```

because `a` is not a number, or as in

```
?- X is 1 + 1 + Z.
Error
```

because `Z` is not instantiated to a number. But

```
?- Z = 2, X is 1 + 1 + Z.
Z = 2, X = 4
```

is correct because `Z` has a numerical value when `X` is evaluated.

A.10.3 Comparison Operators

Comparison operators process arithmetic and literal expressions. They evaluate arithmetic expressions to the left and to the right of the operator before comparing them, for example:

```
?- 1 + 2 < 3 + 4.
true.
```

Comparison operators for literal expressions rank terms according to their lexical order, for example:

```
?- a @< b.
true.
```

Standard Prolog defines a lexical ordering of terms that is based on the ASCII value of characters and other considerations. Table A.4 shows a list of comparison operators for arithmetic and literal expressions.

It is a common mistake of beginners to confuse the arithmetic comparison (`=:=`), literal comparison (`==`), and even sometimes unification (`=`). Unification is a logical operation that finds two substitutions to render two terms identical; an arithmetic comparison computes the numerical values of the left and right expressions and compares their resulting value; a term comparison compares literal values of terms but does not perform any operation on them. Here are some examples:

Table A.4 Comparison operators

	Arithmetic comparison	Literal term comparison
Equality operator	`=:=`	`==`
Inequality operator	`=\=`	`\==`
Less than	`<`	`@<`
Less than or equal	`=<`	`@=<`
Greater than	`>`	`@>`
Greater than or equal	`>=`	`@>=`

```
?- 1 + 2 =:= 2 + 1.
true.
?- 1 + 2 = 2 + 1.
false.
?- 1 + 2 = 1 + 2.
true.
?- 1 + X = 1 + 2.
X = 2
?- 1 + X =:= 1 + 2.
Error
```

```
?- 1 + 2 == 1 + 2.
true.
?- 1 + 2 == 2 + 1.
false.
?- 1 + X == 1 + 2.
false.
?- 1 + a == 1 + a.
true.
```

A.10.4 Lists and Arithmetic: The `length/2` Predicate

The `length/2` predicate determines the length of a list

```
?- length([a, b, c], 3).
true.

?- length([a, [a, b], c], N).
N = 3
```

`length(+List, ?N)` traverses the list `List` and increments a counter `N`. Its definition in Prolog is:

```
length([],0).
length([X | XS], N) :-
  length(XS, N1),
  N is N1 + 1.
```

The order of subgoals in the rule is significant because `N1` has no value until Prolog has traversed the whole list. This value is computed as Prolog pops the recursive calls from the stack. Should subgoals be inverted, the computation of the length would generate an error telling us that `N1` is not a number.

A.10.5 Lists and Comparison: The `quicksort/2` Predicate

The `quicksort/2` predicate sorts the elements of a list `[H | T]`. It first selects an arbitrary element from the list to sort, here the head, `H`. It splits the list into two sublists containing the elements smaller than this arbitrary element and the elements greater. `Quicksort` then sorts both sublists recursively and appends them once they are sorted. In this program, the `before/2` predicate compares the list elements using the `@</2` literal operator.

```
% quicksort(+InputList, -SortedList)

quicksort([], []) :- !.
quicksort([H | T], LSorted) :-
  split(H, T, LSmall, LBig),
  quicksort(LSmall, LSmallSorted),
  quicksort(LBig, LBigSorted),
  append(LSmallSorted, [H | LBigSorted], LSorted).

split(X, [Y | L], [Y | LSmall], LBig) :-
  before(Y, X),
  !,
  split(X, L, LSmall, LBig).
split(X, [Y | L], LSmall, [Y | LBig]) :-
  !,
  split(X, L, LSmall, LBig).
split(_, [], [], []) :- !.

before(X, Y) :- X @< Y.
```

A.11 Some Other Built-in Predicates

The set of built-in predicates may vary according to Prolog implementations. Here is a list common to many Prologs. Consult your reference manual to have the complete list.

A.11.1 Type Predicates

The type predicates check the type of a term. Their mode of use is
`type_predicate(?Term)`.

- `integer/1`: Is the argument an integer?

```
?- integer(3).
true.

?- integer(X).
false.
```

- `number/1`: Is the argument a number?

```
?- number(3.14).
true.
```

- `float/1`: Is the argument a floating-point number?
- `atom/1`: Is the argument an atom?

```
?- atom(abc).
true.

?- atom(3).
false.
```

- `atomic/1`: Is the argument an atomic value, i.e., a number or an atom?
- `var/1`: Is the argument a variable?

```
?- var(X).
true.

?- X = f(Z), var(X).
false.
```

- `nonvar/1`: The opposite of `var/1`.

```
?- nonvar(X).
false.
```

- `compound/1`: Is the argument a compound term?

```
?- compound(X).
false.

?- compound(f(X, Y)).
true.
```

- `ground/1`: Is the argument a ground term?

```
?- ground(f(a, b)).
true.

?- ground(f(a, Y)).
false.
```

A.11.2 Term Manipulation Predicates

The term manipulation predicates enable us to access and modify elements of compound terms.

- The built-in predicate `functor(+Term, ?Functor, ?Arity)` gets the principal functor of a term and its arity.

```
?- functor(father(ulysses, telemachus), F, A).
F = father, A = 2
```

`functor` also returns the most general term given a functor name and an arity. `Functor` and `Arity` must then be instantiated: `functor(-Term, +Functor, +Arity)`

```
?- functor(T, father, 2).
T = father(X, Y)
```

- The predicate `arg(+N, +Term, ?X)` unifies `X` to the argument of rank `N` in `Term`.

```
?- arg(1, father(ulysses, telemachus), X).
X = ulysses
```

- The operator `Term =.. List`, also known as the *univ* predicate, transforms a term into a list.

```
?- father(ulysses, telemachus) =.. L.
L = [father, ulysses, telemachus]

?- T =.. [a, b, c].
T = a(b, c)
```

Univ has two modes of use: `+Term =.. ?List`, or `-Term =.. +List`.

- The predicate `name(?Atom, ?List)` transforms an atom into a list of ASCII codes.

```
?- name(abc, L).
L = [97, 98, 99]

?- name(A, [97, 98, 99]).
A = abc
```

Standard Prolog provides means to encode strings more naturally using double quotes. Thus

```
?- "abc" = L.
L = [97, 98, 99]
```

A.12 Handling Run-Time Errors and Exceptions

Standard Prolog features a mechanism to handle run-time errors. An error or exception occurs when the execution cannot be completed normally either successfully or by a failure. Examples of exceptions include division by zero, the attempt to evaluate arithmetically nonnumerical values with `is/2`, and calling a noninstantiated variable in the body of a rule:

```
?- X is 1/0.
ERROR: //2: Arithmetic evaluation error: zero_divisor

?- X is 1 + Y.
ERROR: Arguments are not sufficiently instantiated

?- X.
ERROR: Arguments are not sufficiently instantiated
```

In the normal course of a program, such faulty clauses generate run-time errors and stop the execution. The programmer can also trap these errors and recover from them using the `catch/3` built-in predicate.

`catch(+Goal, ?Catcher, ?Recover)` executes `Goal` and behaves like `call/1` if no error occurs. If an error is raised and unifies with `Catcher`, `catch/3` proceeds with `Recover` and continues the execution.

Standard Prolog defines catchers of built-in predicates under the form of the term `error(ErrorTerm, Information)`, where `ErrorTerm` is a standard description of the error and `Information` depends on the implementation. The query:

```
?- catch((X is 1 + Y), Error, (write(Error),nl,fail)).
error(instantiation_error, context(system: (is)/2, _GXyz))
false.
```

attempts to execute `X is Y + 1`, raises an error, and executes the recover goal, which prints the error and fails. The constant `instantiation_error` is part of the set of error cases defined by Standard Prolog.

Built-in predicates execute a `throw/1` to raise exceptions when they detect an error. The `throw` predicate immediately goes back to a calling `catch/3`. If there is no such `catch`, by default, the execution is stopped and the control is transferred to the user.

User-defined predicates can also make use of `throw(+Exception)` to throw an error, as in:

```
throw_error :- throw(error(error_condition,context)).
```

The corresponding error can be caught as in the query:

```
?- catch(throw_error, Error, (write(Error),nl,fail)).
error(error_condition, context)
false.
```

A.13 Dynamically Accessing and Updating the Database

A.13.1 Accessing a Clause: The `clause/2` Predicate

The built-in predicate `clause(+Head, ?Body)` returns the body of a clause whose head unifies with `Head`. Let us illustrate this with the program:

```
hero(ulysses).
heroine(penelope).

daughter(X, Y) :-
  mother(Y, X),
  female(X).
daughter(X, Y) :-
  father(Y, X),
  female(X).
```

and the query:

```
?- clause(daughter(X, Y), B).
B = (mother(Y, X), female(X));
B = (father(Y, X), female(X));

?- clause(heroine(X), B).
X = penelope, B = true.
```

A.13.2 Dynamic and Static Predicates

The built-in predicates `asserta/1`, `assertz/1`, `retract/1`, and `abolish/1` add or remove clauses – rules and facts – during the execution of a program. They allow us to update the database – and hence to modify the program – dynamically.

A major difference between Prolog implementations is whether the system interprets the program or compiles it. Roughly, an interpreter does not change the format of rules and facts to run them. A compiler translates clauses into a machine-dependent code or into more efficient instructions (Maier and Warren 1988). A compiled program runs much faster then.

Compiling occurs once at load time, and the resulting code is no longer modifiable during execution. To run properly, the Prolog engine must be told which predicates are alterable at run-time – the **dynamic** predicates – and which ones will remain unchanged – the **static** predicates. Prolog compiles static predicates and runs dynamic predicates using an interpreter.

A predicate is static by default. Dynamic predicates must either be declared using the `dynamic/1` directive or be entirely created by assertions at run time. In the

latter case, the first assertion of a clause declares automatically the new predicate to be dynamic. The directive specifying that a predicate is dynamic precedes all its clauses, if any. For example, the program:

```
:- dynamic parent/2, male/1.
...
parent(X, Y) :-
...
male(xy).
...
```

declares that `parent/2` and `male/1` clauses may be added or removed at run time.

The predicates `asserta/1`, `assertz/1`, `retract/1`, and `abolish/1` can modify clauses of dynamic predicates only. Adding or removing a clause for a static predicate raises an error condition.

A.13.3 Adding a Clause: The `asserta/1` and `assertz/1` Predicates

The predicate `asserta(+P)` adds the clause `P` to the database. `P` is inserted just before the other clauses of the same predicate. As we have seen before, the predicate corresponding to the clause `P` must be dynamic: declared using the `dynamic/1` directive or entirely asserted at run time.

```
                              % State of the database
                              % Before assertion
                              % hero(ulysses).
                              % hero(hector).
?- asserta(hero(achilles)).
                              % State of the database
                              % After assertion
                              % hero(achilles).
                              % hero(ulysses).
                              % hero(hector).
```

The predicate `assertz/1` also adds a new clause, but as the last one of the procedure this time.

Adding rules is similar. It requires double parentheses, as in

```
asserta((P :- B, C, D)).
```

However, it is never advised to assert rules. Modifying rules while running a program is rarely useful and may introduce nasty bugs.

Novice Prolog programmers may try to communicate the results of a procedure by asserting facts to the database. This is not a good practice because it hides what is the real output of a predicate. Results, especially intermediate results, should be passed along from one procedure to another using arguments. Assertions should only reflect a permanent change in the program state.

A.13.4 Removing Clauses: The `retract/1` and `abolish/2` Predicates

The built-in predicates `retract/1` and `abolish/1` remove clauses of a dynamic predicate. `retract(+P)` retracts clause `P` from the database.

```
                              % State of the database
                              % Before removal
                              % hero(ulysses).
                              % hero(achilles).
                              % hero(hector).
?- retract(hero(hector)).
                              % State of the database
                              % After
                              % hero(ulysses).
                              % hero(achilles).
?- retract(hero(X)).
X = ulysses ;
X = achilles ;

?- hero(X).
false.
```

The predicate `abolish(+Predicate/Arity)` removes all clauses of `Predicate` with arity `Arity` from the database.

A.13.5 Handling Unknown Predicates

When a static predicate is called and is not in the database, it is often a bug. A frequent cause is due to wrong typing as, for example, `parnet(X, Y)` instead of `parent(X, Y)`, where `n` and `e` are twiddled. For this reason, by default, Prolog raises an error in the case of such a call.

An effect of `dynamic/1` is to declare a predicate to the Prolog engine. Such a predicate 'exists' then, even if it has no clauses. A call to a dynamic predicate

that has no clauses in the database is not considered as an error. It fails, simply and silently.

The Prolog engine behavior to calls to unknown predicates can be modified using the `unknown/2` directive:

```
:- unknown(-OldValue, +NewValue).
```

where `OldValue` and `NewValue` can be:

- `warning` – A call to an unknown predicate issues a warning and fails.
- `error` – A call to an unknown predicate raises an error. As we saw, this is the default value.
- `fail` – A call to an unknown predicate fails silently.

A Prolog flag also defines this behavior. It can be set by `set_prolog_flag/2`:

```
?- set_prolog_flag(+FlagName, +NewValue).
```

where `FlagName` is set to `unknown` and possible values are `error`, `warning`, or `fail`. The current flag status is obtained by `current_prolog_flag/2`:

```
?- current_prolog_flag(+FlagName, ?Value).
```

A.14 All-Solutions Predicates

The second-order predicates `findall/3`, `bagof/3`, and `setof/3` return all the solutions to a given query. The predicate `findall` is the basic form of all-solutions predicates, while `bagof` and `setof` are more elaborate. We exemplify them with the database:

```
character(ulysses, iliad).
character(hector, iliad).
character(achilles, iliad).
character(ulysses, odyssey).
character(penelope, odyssey).
character(telemachus, odyssey).
```

and the `male` and `female` predicates from Sect. A.2.1.

`findall(+Variable, +Goal, ?Solution)` unifies `Solution` with the list of all the possible values of `Variable` when querying `Goal`.

```
?- findall(X, character(X, iliad), B).
B = [ulysses, hector, achilles]

?- findall(X, character(X, Y), B).
B = [ulysses, hector, achilles, ulysses, penelope, telemachus]
```

The predicate `bagof(+Variable, +Goal, ?Solution)` is similar to `findall/3`, except that it backtracks on the free variables of `Goal`:

```
?- bagof(X, character(X, iliad), Bag).
Bag = [ulysses, hector, achilles]

?- bagof(X, character(X, Y), Bag).
Bag =[ ulysses, hector, achilles], Y = iliad ;
Bag = [ulysses, penelope, telemachus], Y = odyssey ;
?-
```

Variables in `Goal` are not considered free if they are existentially quantified. The existential quantifier uses the infix operator "`^`". Let `X` be a variable in `Goal`. `X^Goal` means that there exists `X` such that `Goal` is true. `bagof/3` does not backtrack on it. For example:

```
?- bagof(X, Y^character(X, Y), Bag).
Bag = [ulysses, hector, achilles, ulysses, penelope, telemachus]

?- bagof(X, Y^(character(X, Y), female(X)), Bag).
Bag = [penelope]
```

The predicate `setof(+Variable, +Goal, ?Solution)` does the same thing as `bagof/3`, except that the `Solution` list is sorted and duplicates are removed from it:

```
?- setof(X, Y^character(X, Y), Bag).
Bag = [achilles, hector, penelope, telemachus, ulysses]
```

A.15 Fundamental Search Algorithms

Many problems in logic can be represented using a graph or a tree, where finding a solution corresponds to searching a path going from an initial state to a goal state. The search procedure starts from an initial node, checks whether the current node meets a goal condition, and, if not, goes to a next node. The transition from one node to a next one is carried out using a successor predicate, and the solution is the sequence of nodes traversed to reach the goal. In the context of search, the graph is also called the **state space**.

In this section, we will review some fundamental search strategies, and as an application example we will try to find our way through the labyrinth shown in Fig. A.9. As we saw, Prolog has an embedded search mechanism that can be used with little adaptation to implement other algorithms. It will provide us with the Ariadne's thread to remember our way in the maze with minimal coding efforts.

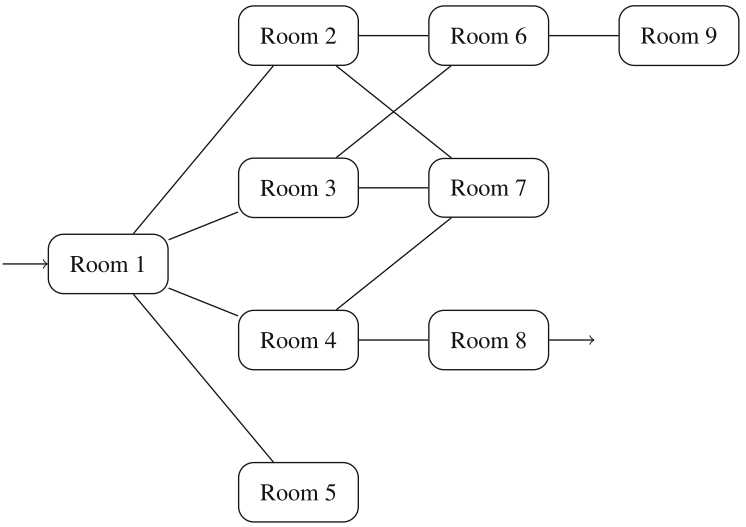


Fig. A.9 The graph representing the labyrinth

A.15.1 Representing the Graph

We use a successor predicate `s(X, Y)` to represent the graph, where `Y` is the successor of `X`. For the labyrinth, the `s/2` predicate describes the immediate links from one room to another. The links between the rooms are:

```
link(r1, r2). link(r1, r3). link(r1, r4). link(r1, r5).
link(r2, r6). link(r2, r7). link(r3, r6). link(r3, r7).
link(r4, r7). link(r4, r8). link(r6, r9).
```

Since links can be traversed both ways, the `s/2` predicate is:

```
s(X, Y) :- link(X, Y).
s(X, Y) :- link(Y, X).
```

The goal is expressed as:

```
goal(X) :- minotaur(X).
```

where

```
minotaur(r8).
```

Finally, we could associate a cost to the link, for instance, to take into account its length. The predicate would then be:

```
s(X, Y, Cost).
```

A.15.2 Depth-First Search

A depth-first search is just the application of the Prolog resolution strategy. It explores the state space by traversing a sequence of successors to the initial node until it finds a goal. The search goes down the graph until it reaches a node without successor. It then backtracks from the bottom to the last node that has successors.

Searching a path in a labyrinth is then very similar to other programs we have written before. It consists of a first rule to describe the goal condition and second recursive one to find a successor node when the condition is not met. The `depth_first_search(+Node, -Path)` predicate uses the initial node as input and returns the path to reach the goal:

```
%% depth_first_search(+Node, -Path)
depth_first_search(Node, [Node]) :-
  goal(Node).
depth_first_search(Node, [Node | Path]) :-
  s(Node, Node1),
  depth_first_search(Node1, Path).
```

This short program does not work, however, because the path could include infinite cycles: Room 2 to Room 6 to Room 2 to Room 6... To prevent them, we need to remember the current path in an accumulator variable and to avoid the successors of the current node that are already members of the path. We use a `depth_first_search/3` auxiliary predicate, and the new program is:

```
%% depth_first_search(+Node, -Path)
depth_first_search(Node, Path) :-
  depth_first_search(Node, [], Path).

%% depth_first_search(+Node, +CurrentPath,-FinalPath)
depth_first_search(Node, Path, [Node | Path]) :-
  goal(Node).
depth_first_search(Node, Path, FinalPath) :-
  s(Node, Node1),
  \+ member(Node1, Path),
  depth_first_search(Node1, [Node | Path],FinalPath).
```

The result of the search is:

```
?- depth_first_search(r1, L).
L = [r8, r4, r7, r3, r6, r2, r1] ;
L = [r8, r4, r7, r2, r1] ;
L = [r8, r4, r7, r2, r6, r3, r1] ;
L = [r8, r4, r7, r3, r1] ;
L = [r8, r4, r1] ;
false.
?-
```

A.15.3 Breadth-First Search

The breadth-first search explores the paths in parallel. It starts with the first node, all the successors of the first node, all the successors of the successors, and so on, until it finds a solution.

If the list `[Node | Path]` describes a path to a node, the search needs to expand all the successors of `Node`. It generates the corresponding paths as lists. There are as many lists as there are successors to `Node`. The search then sets the successors as the heads of these lists. This is done compactly using the `bagof/3` predicate:

```
expand([Node | Path], ExpandedPaths) :-
  bagof(
    [Node1, Node | Path],
    (s(Node, Node1), \+ member(Node1, Path)),
    ExpandedPaths).
```

As with the depth-first search, the breadth-first search consists of two rules. The first rule describes the goal condition. It extracts the first path from the list and checks whether the head node is a goal. The second rule implements the recursion. It expands the first path – the head of the list – into a list of paths that go one level deeper in the graph and appends them to the end of the other paths. The `breadth_first_search(+Node, -Path)` predicate uses the initial node as input and returns the path to reach the goal. The program needs to start with a list of lists, and it uses the auxiliary predicate `bf_search_aux/2`.

```
%% breadth_first_search(+Node, -Path)
breadth_first_search(Node, Path) :-
  bf_search_aux([[Node]], Path).

bf_search_aux([[Node | Path] | _], [Node | Path]) :-
  goal(Node).
bf_search_aux([CurrentPath | NextPaths], FinalPath) :-
  expand(CurrentPath, ExpandedPaths),
  append(NextPaths, ExpandedPaths, NewPaths),
  bf_search_aux(NewPaths, FinalPath).
```

The program is not completely correct, however, because `expand/2` can fail and make the whole search fail. A failure of `expand/2` means that the search cannot go further in this path and it has found no goal node in it. We can remove the path from the list then. To reflect this, we must add a second rule to `expand/2` that sets the path to the empty list and prevents the first rule from backtracking:

```
expand([Node | Path], ExpandedPaths) :-
  bagof(
    [Node1, Node | Path],
```

```
      (s(Node, Node1), \+ member(Node1, Path)),
      ExpandedPaths),
    !.
expand(Path, []).
```

The result of the search is:

```
?- breadth_first_search(r1, L).
L = [r8, r4, r1] ;
L = [r8, r4, r7, r2, r1] ;
L = [r8, r4, r7, r3, r1] ;
L = [r8, r4, r7, r3, r6, r2, r1] ;
L = [r8, r4, r7, r2, r6, r3, r1] ;
false.
?-
```

The breadth-first search strategy guarantees that it will find the shortest path to the solution. A disadvantage is that it must store and maintain all exploration paths in parallel. This requires a huge memory, even for a limited search depth.

A.15.4 A Search*

The A* search is a variation and an optimization of the breadth-first search. Instead of expanding the first path of the list, it uses heuristics to select a better candidate. While searching the graph, A* associates a value to paths it traverses. This value is a function f of the node being traversed. $f(n)$ at node n is the sum of two terms $f(n) = g(n) + h(n)$, where $g(n)$ is the length of the path used to reach node n and $h(n)$ is the estimate of the remaining length to reach the goal node. From a given node, A* ranks the possible subsequent nodes minimizing $f(n)$. It then explores "best nodes" first and thus avoids a blind searching.

The main difficulty of the A* search is to find a suitable h function. Its presentation is outside the scope of this appendix. Russell and Norvig (2010) examine search strategies in detail. Bratko (2012) describes an implementation of A* in Prolog.

A.16 Input/Output

The first Prolog systems had only primitive input/output facilities. Standard Prolog defines a complete new set of predicates. They represent a major change in the Prolog language, and although they are more flexible they are not universally accepted yet. This section introduces both sets of predicates. It outlines Standard

Prolog input/output predicates and predicates conforming to the older tradition of Edinburgh Prolog. Most input/output predicates are deterministic, that is, they give no alternative solutions upon backtracking.

A.16.1 Input/Output with Edinburgh Prolog

Reading and Writing Characters

In Edinburgh Prolog, reading characters from the keyboard and writing to the screen is carried out using `get0/1` and `put/1`. Both predicates process characters using their ASCII codes. `get0/1` unifies with -1 when it reaches the end of a file. Here are some examples of use:

```
?- get0(X).
a <return>

X = 97

?- put(65).
a

?- get0(X).
^D

X = -1
```

Reading and Writing Terms

The built-in predicates `read/1` and `write/1` read and write terms from the current input and output streams. `read(?Term)` reads one term:

```
?- read(X).
character(ulysses, odyssey).

X = character(ulysses, odyssey)
```

where the input term must be terminated by a period. When reaching the end of a file, `X` unifies with the build-in atom `end_of_file`:

```
?- read(X).
^D
X = end_of_file
```

Writing terms is similar. `write(+Term)` writes one term to the current output stream and `nl/0` prints a new line:

```
?- T = character(ulysses, odyssey), write(T), nl.
character(ulysses, odyssey)

T = character(ulysses, odyssey)
?-
```

Opening and Closing Files

Prolog input and output predicates normally write on the screen – the standard output – and read from the keyboard – the standard input. The predicates `see/1` and `tell/1` redirect the input and output so that a program can read or write any file.

`see/1` and `tell/1` open a file for reading and for writing. Then input/output predicates such as `get0/1`, `read/1` or `put/1`, `write/1` are redirected to the current open file. Several files may be open at the same time. The program switches between open files using `see/1` or `tell/1` until they are closed. `seen/0` and `told/0` close the open input and the open output, respectively, and return to the standard input/output, that is, to the keyboard and the screen. Let us show this with an example.

`see(in_file),`	Opens `in_file` as the current input stream.
`see(user),`	The current stream becomes the user – the keyboard.
`see(in_file),`	`in_file` becomes the current input stream again with the reading the position it had before.
`seen,`	Closes the current input stream. The current stream becomes the keyboard.
`seeing(IN_STREAM),`	`IN_STREAM` unifies with the current input stream.
`tell(out_file),`	Opens `out_file` as the current output stream (creates a new file or empties a previously existing file).
`telling(OUT_STREAM),`	`OUT_STREAM` unifies with the current output stream.
`tell(user),`	The current output stream becomes the user – the screen.
`told.`	Closes the current output stream. The current output stream becomes the user.

Here is a short program to read a file:

```
read_file(FileName, CodeList) :-
  see(FileName),
  read_list(CodeList),
  seen.
```

```
read_list([C | L]) :-
  get0(C),
  C =\= -1, % end of file
  !,
  read_list(L).
read_list([]).
```

A.16.2 Input/Output with Standard Prolog

Reading and Writing Characters

Standard Prolog uses streams to read and write characters. A stream roughly corresponds to an open file. Streams are divided into output streams or sinks, and input streams or sources. By default, there are two current open streams: the standard input stream, which is usually the keyboard, and the standard output stream, the screen. Other streams are opened and closed using `open/4`, `open/3`, `close/1`, and `close/2`.

The predicates to read and write a character are `get_char/1`, `get_char/2`, `put_char/1`, and `put_char/2`:

- `get_char(?Char)` unifies `Char` with the next character of the current input stream.
- `get_char(+Stream, ?Char)` unifies `Char` with the next character of the open input stream `Stream`. `get_char/1` and `get_char/2` predicates unify with `end_of_file` when they reach the end of a file.
- `put_char(+Char)` writes `Char` to the current output stream.
- `put_char(+Stream, ?Char)` writes `Char` to the open output `Stream`.
- `nl/0` and `nl(+Stream)` write a new line to the current output stream or to `Stream`.

Here is a short example:

```
?- get_char(X).
a <return>

X = a

?- put_char(a).
a

?- get_char(X).
^D

X = end_of_file
```

Instead of reading and writing characters, we may want to read or write their numeric code, ASCII or Unicode, as with Edinburgh's `get0/1`. The corresponding Standard Prolog predicates are `get_code/1`, `get_code/2`, `put_code/1`, and `put_code/2`.

The predicates `get_char` and `get_code` read a character or a code, remove it from the input stream, and move to the next character. Sometimes it is useful to read a character without removing it. The predicates `peek_char` and `peek_code` do just that. They unify with the current character but stay at the same position and leave the character in the stream.

Reading and Writing Terms

The Standard Prolog predicates `read/1` and `write/1` are identical to those of Edinburgh Prolog:

- `read(?Term)` reads one term from the current input stream.
- `write(+Term)` writes a term to the current output stream.

`read/2` and `write/2` read and write terms from and to a file:

- `read(+Stream, ?Term)` reads a term from `Stream`.
- `write(+Stream, ?Term)` writes a term to `Stream`.

The predicates `read_term` and `write_term` read and write terms with a list of options, either to the current input/output, `read_term/2` and `write_term/2`, or to a file, `read_term/3` and `write_term/3`. The options make it possible to adjust the printing format, for instance. They may depend on the implementation and the operating system. Consult your manual to have the complete list. The predicates `read` and `write` are equivalent to `read_term` and `write_term` with an empty list of options.

Opening and Closing Files

The predicates to open and close a stream are `open/4`, `open/3`, `close/1`, and `close/2`:

- `open(+SourceSink, +Mode, -Stream)` opens the file `SourceSink` in an input or output `Mode`. The `Mode` value is one of `read`, `write`, `append`, or `update`. `Stream` unifies with the opened stream and is used for the subsequent input or output operations.
- `open(+SourceSink, +Mode, -Stream, +Options)` opens the file with a list of options. `open/3` is equivalent to `open/4` with an empty list of options. Consult your manual to have the complete list.
- `close(+Stream)` closes the stream `Stream`.
- `close(+Stream, +Options)` closes the stream `Stream` with a list of options. `close/1` is equivalent to `close/2` with an empty list of options.

Here is a short program to read a file with Standard Prolog predicates:

```
read_file(FileName, CharList) :-
  open(FileName, read, Stream),
  read_list(Stream, CharList),
  close(Stream).

read_list(Stream, [C | L]) :-
  get_char(Stream, C),
  C \== end_of_file, % end of file
  !,
  read_list(Stream, L).
read_list(_, []).
```

Other useful predicates include `current_input/1`, `current_output/1`, `set_input/1`, and `set_output/1`:

- `current_input(?Stream)` unifies `Stream` with the current input stream.
- `current_output(?Stream)` unifies `Stream` with the current output.
- `set_input(+Stream)` sets `Stream` to be the current input stream.
- `set_output(+Stream)` sets `Stream` to be the current output stream.

A.16.3 Writing Loops

Programmers sometimes wonder how to write iterative loops in Prolog, especially with input/output to read or to write a sequence of terms. This is normally done with a recursive rule, as to read a file. Counting numbers down to 0 takes the form:

```
countdown(X) :-
  number(X),
  X < 0.
countdown(X):-
  number(X),
  X >= 0,
  write(X), nl,
  NX is X - 1,
  countdown(NX).
```

For example,

```
?- countdown(4).
4
3
2
```

```
1
0
true.
?-
```

In some other cases, backtracking using the `repeat/0` built-in predicate can substitute a loop. The `repeat/0` definition is:

```
repeat.
repeat :- repeat.
```

`repeat` never fails and, when inserted as a subgoal, any subsequent backtracking goes back to it and the sequence of subgoals to its right gets executed again. So, a sequence of subgoals can be executed any number of times until a condition is satisfied. The `read_write/1` predicate below reads and writes a sequence of atoms until the atom `end` is encountered. It takes the form of a repetition (`repeat`) of reading a term `X` using `read/1`, writing it (`write/1`), and a final condition (`X == end`). It corresponds to the rule:

```
read_write :-
  repeat,
  read(X),
  write(X), nl,
  X == end,
  !.
```

A.17 Developing Prolog Programs

A.17.1 Presentation Style

Programs are normally written once and then are possibly read and modified several times. A major concern of the programmer should be to write clear and legible code. It helps enormously with the maintenance and debugging of programs.

Before programming, it is essential first to have a good formulation and decomposition of the problem. The program construction should then reflect the logical structure of the solution. Although this statement may seem obvious, its implementation is difficult in practice. Clarity in a program structure is rarely attained the first time. First attempts are rarely optimal but Prolog enables an incremental development where parts of the solution can be improved gradually.

A key to the good construction of a program is to name things properly. Cryptic predicates or variable names, such as `syntproc`, `def_code`, `X`, `Ynn`, and so on, should be banned. It is not rare that one starts with a predicate name and changes it in the course of the development to reflect a better description of the solution.

Since Prolog code is compact, the code of a clause should be short to remain easy to understand, especially with recursive programs. If necessary, the programmer should decompose a clause into smaller subclauses. Cuts and asserts should be kept to a minimum because they impair the declarativeness of a program. However, these are general rules that sometimes are difficult to respect when speed matters most.

Before its code definition, a predicate should be described in comments together with argument types and modes:

```
% predicate(+Arg1, +Arg2, -Arg3).
% Does this and that
% Arg1: list, Arg2: atom, Arg3: integer.
```

Clauses of a same predicate must be grouped together, even if some Prologs permit clauses to be disjoined. The layout of clauses should also be clear and adopt common rules of typography. Insert a space after commas or dots, for instance. The rule

```
pred1 :- pred2(c,d),e,f.
```

must be rejected because of sticking commas and obfuscated predicate names. Goals must be indented with tabulations, and there should be one single goal per line. Then

```
A :-
  B,
  C,
  D.
```

should be preferred to

```
A :- B, C, D.
```

except when the body consists of a single goal. The rule

```
A :- B.
```

is also acceptable.

A.17.2 Improving Programs

Once a program is written, it is generally possible to enhance it. This section introduces three techniques to improve program speed: goal ordering, memo functions, and tail recursion.

Order of Goals

Ordering goals is meaningful for the efficiency of a program because Prolog tries them from left to right. The idea is to reduce the search space as much as possible

from the first goals. If predicate p_1 has 1,000 solutions in 1 s and p_2 has 1 solution taking 1,000 h to compute, avoid conjunction:

```
p1(X), p2(X).
```

A better ordering is:

```
p2(X), p1(X).
```

Lemmas or Memo Functions

Lemmas are used to improve the program speed. They are often exemplified with Fibonacci series. Fibonacci imagined around year 1200 how to estimate a population of rabbits, knowing that:

- A rabbit couple gives birth to another rabbit couple, one male and one female, each month (one month of gestation).
- A rabbit couple reproduces from the second month.
- Rabbits are immortal.

We can predict the number of rabbit couples at month n as a function of the number of rabbit couples at month $n-1$ and $n-2$:

$$rabbit(n) = rabbit(n-1) + rabbit(n-2)$$

A first implementation is straightforward from the formula:

```
fibonacci(1, 1).
fibonacci(2, 1).
fibonacci(M, N) :-
  M > 2,
  M1 is M - 1, fibonacci(M1, N1),
  M2 is M - 2, fibonacci(M2, N2),
  N is N1 + N2.
```

However, this program has an expensive double recursion and the same value can be recomputed several times. A better solution is to store Fibonacci values in the database using `asserta/1`. So an improved version is

```
fibonacci(1, 1).
fibonacci(2, 1).
fibonacci(M, N) :-
  M > 2,
  M1 is M - 1, fibonacci(M1, N1),
  M2 is M - 2, fibonacci(M2, N2),
  N is N1 + N2,
  asserta(fibonacci(M, N)).
```

The rule is then tried only if the value is not in the database.

The generic form of the lemma is:

```
lemma(P):-
  P,
  asserta((P :- !)).
```

with "`!`" to avoid backtracking.

Tail Recursion

A tail recursion is a recursion where the recursive call is the last subgoal of the last rule, as in

```
f(X) :- fact(X).
f(X) :- g(X, Y), f(Y).
```

Recursion is generally very demanding in terms of memory, which grows with the number of recursive calls. A tail recursion is a special case that the interpreter can transform into an iteration. Most Prolog systems recognize and optimize it. They execute a tail-recursive predicate with a constant memory size.

It is therefore significant not to invert clauses of the previous program, as in

```
f(X) :- g(X, Y), f(Y).
f(X) :- fact(X).
```

which is not tail recursive.

It is sometimes possible to transform recursive predicates into a tail recursion equivalent, adding a variable as for `length/2`:

```
length(List, Length) :-
  length(List, 0, Length).

length([], N, N).
length([X | L], N1, N) :-
  N2 is N1 + 1,
  length(L, N2, N).
```

It is also sometimes possible to force a tail recursion using a cut, for example,

```
f(X) :- g(X, Y), !, f(Y).
f(X) :- fact(X).
```

Exercises

A.1. Describe a fragment of your family using Prolog facts.

A.2. Using the model of `parent/2` and `ancestor/2`, write rules describing family relationships.

A.3. Write a program to describe routes between cities. Use a `connect/2` predicate to describe direct links between cities as facts, for example, `connect(paris, london)`, `connect(london, edinburgh)`, etc., and write the `route/2` recursive predicate that finds a path between cities.

A.4. Unify the following pairs:

```
f(g(A, B), a) = f(C, A).
f(X, g(a, b)) = f(g(Z), g(Z, X)).
f(X, g(a, b)) = f(g(Z), g(Z, Y)).
```

A.5. Trace the `son/2` program.

A.6. What is the effect of the query

```
?- f(X, X).
```

given the database:

```
f(X, Y) :- !, g(X), h(Y).
g(a).
g(b).
h(b).
```

A.7. What is the effect of the query

```
?- f(X, X).
```

given the database:

```
f(X, Y) :- g(X), !, h(Y).
g(a).
g(b).
h(b).
```

A.8. What is the effect of the query

```
?- f(X, X).
```

given the database:

```
f(X, Y) :- g(X), h(Y), !.
g(a).
g(b).
h(b).
```

A.9. What is the effect of the query

```
?- \+  f(X,  X).
```

given the databases of the three previous exercises (Exercises A.6–A.8)? Provide three answers.

A.10. Write the `last(?List, ?Element)` predicate that succeeds if `Element` is the last element of the list.

A.11. Write the `nth(?Nth, ?List, ?Element)` predicate that succeeds if `Element` is the `Nth` element of the list.

A.12. Write the `maximum(+List, ?Element)` predicate that succeeds if `Element` is the greatest of the list.

A.13. Write the `flatten/2` predicate that flattens a list, i.e., removes nested lists:

```
?- flatten([a, [a, b, c], [[[d]]], L).
L = [a, a, b, c, d]
```

A.14. Write the `subset(+Set1, +Set2)` predicate that succeeds if `Set1` is a subset of `Set2`.

A.15. Write the `subtract(+Set1, +Set2, ?Set3)` predicate that unifies `Set3` with the subtraction of `Set2` from `Set1`.

A.16. Write the `union(+Set1, +Set2, ?Set3)` predicate that unifies `Set3` with the union of `Set2` and `Set1`. `Set1` and `Set2` are lists without duplicates.

A.17. Write a program that transforms the lowercase characters of a file into their uppercase equivalent. The program should process accented characters, for example, *é* will be mapped to *É*.

A.18. Implement A* in Prolog.

References

Abeillé, A. (1993). *Les nouvelles syntaxes: Grammaires d'unification et analyse du français*. Paris: Armand Colin.

Abeillé, A., & Clément, L. (2003). Annotation morpho-syntaxique. Les mots simples – Les mots composés. Corpus Le Monde. Technical report, LLF, Université Paris 7, Paris.

Abeillé, A., Clément, L., & Toussenel, F. (2003). Building a treebank for French. In A. Abeillé (Ed.), *Treebanks: Building and using parsed corpora* (Text, speech and language technology, Vol. 20, chapter 10, pp. 165–187). Dordrecht: Kluwer Academic.

Abney, S. (1994). Partial parsing. http://www.vinartus.net/spa/publications.html. Retrieved 7 Nov 2013. Tutorial given at ANLP-94, Stuttgart.

Abney, S. (1996). Partial parsing via finite-state cascades. In *Proceedings of the ESSLLI'96 robust parsing workshop*, Prague.

Agnäs, M.-S., Alshawi, H., Bretan, I., Carter, D., Ceder, K., Collins, M., Crouch, R., Digalakis, V., Ekholm, B., Gambäck, B., Kaja, J., Karlgren, J., Lyberg, B., Price, P., Pulman, S., Rayner, M., Samuelsson, C., & Svensson, T. (1994). Spoken language translator, first-year report. Research report R94:03, SICS, Kista.

Aho, A. V., Sethi, R., & Ullman, J. D. (1986). *Compilers: Principles, techniques, and tools*. Reading: Addison-Wesley.

Alexandersson, J. (1996). Some ideas for the automatic acquisition of dialogue structure. In S. LuperFoy, A. Nijholt, & G. V. van Zanten (Eds.), *Proceedings of the eleventh Twente workshop on language technology* (pp. 149–158). Enschede: Universiteit Twente.

Allauzen, C., Riley, M., Schalkwyk, J., Skut, W., & Mohri, M. (2007). OpenFst: A general and efficient weighted finite-state transducer library. In J. Holub & J. Žďárek (Eds.), *12th international conference on implementation and application of automata (CIAA 2007)*, Prague, July 2007. Revised selected papers (Lecture notes in computer science, Vol. 4783, pp. 11–23). Berlin/Heidelberg/New York: Springer.

Allemang, D., & Hendler, J. (2011). *Semantic web for the working ontologist: Effective modeling in RDFS and OWL* (2nd ed.). Waltham: Morgan Kaufmann.

Allen, J. F. (1983). Maintaining knowledge about temporal intervals. *Communications of the ACM, 26*(11), 832–843.

Allen, J. F. (1984). Towards a general theory of action and time. *Artificial Intelligence, 23*(2), 123–154.

Allen, J. F. (1994). *Natural language understanding* (2nd ed.). Redwood City: Benjamin/Cummings.

Allen, J. F., & Core, M. (1997). Draft of DAMSL: dialog annotation markup in several layers. Retrieved November 7, 2013, from http://www.cs.rochester.edu/research/cisd/resources/damsl/

Allen, J. F., & Perrault, C. R. (1980). Analyzing intentions in utterances. *Artificial Intelligence, 15*(3), 143–178.

DOI 10.1007/978-3-642-41464-0, © Springer-Verlag Berlin Heidelberg 2014

Allen, J. F., Schubert, L. K., Ferguson, G., Heeman, P., Hwang, C. H., Kato, T., Light, M., Martin, N. G., Miller, B. W., Poesio, M., & Traum, D. R. (1995). The TRAINS project: A case study in building a conversational planning agent. *Journal of Experimental and Theoretical AI, 7*, 7–48.

Alshawi, H. (Ed.). (1992). *The core language engine*. Cambridge, MA: MIT.

Alshawi, H. (1996). Head automata and bilingual tiling: Translation with minimal representations. In *Proceedings of the 34th annual meeting on association for computational linguistics*, Santa Cruz (pp. 167–176).

Andry, F. (1992). *Mise en œuvre de prédictions linguistiques dans un système de dialogue oral homme-machine coopératif*. PhD thesis, Université Paris Nord.

Antworth, E. L. (1994). Morphological parsing with a unification-based word grammar. In *North Texas natural language processing workshop*, University of Texas at Arlington.

Antworth, E. L. (1995). *User's guide to PC-KIMMO, version 2*. Dallas: Summer Institute of Linguistics.

Apache OpenNLP Development Community. (2012). *Apache OpenNLP developer documentation*. The Apache Software Foundation, 1.5.2 ed.

Appelt, D., Hobbs, J., Bear, J., Israel, D., Kameyama, M., & Tyson, M. (1993). SRI: Description of the JV-FASTUS system used for MUC-5. In *Fifth message understanding conference (MUC-5): Proceedings of a conference held in Baltimore* (pp. 221–235). San Francisco: Morgan Kaufmann.

Apt, K. (1997). *From logic programming to Prolog*. London: Prentice Hall.

Atkins, B. T. (Ed.). (1996). *Collins-Robert French-English, English-French dictionary*. New York/ Paris: HarperCollins and Dictionnaires Le Robert.

Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R., & Ives, Z. (2007). DBpedia: A nucleus for a web of open data. In *The semantic web, proceedings of 6th international semantic web conference, 2nd Asian semantic web conference (ISWC 2007 + ASWC 2007)*, Busan (pp. 722–735). Springer.

Austin, J. L. (1962). *How to do things with words*. Cambridge, MA: Harvard University Press.

Baeza-Yates, R., & Ribeiro-Neto, B. (2011). *Modern information retrieval: The concepts and technology behind search* (2nd ed.). New York: Addison-Wesley.

Bagga, A., & Baldwin, B. (1998). Algorithms for scoring coreference chains. In *Proceedings of the linguistic coreference workshop at the first international conference on language resources and evaluation*, Granada (pp. 563–566).

Ball, G., Ling, D., Kurlander, D., Miller, J., Pugh, D., Skelly, T., Stankosky, A., Thiel, D., Dantzich, M. V., & Wax, T. (1997). Lifelike computer characters: The Persona project at Microsoft research. In J. M. Bradshaw (Ed.), *Software agents* (pp. 191–222). Cambridge, MA: AAAI/MIT.

Baumann, R. (1991). *PROLOG. Einführungskurs*. Stuttgart: Klett Schulbuch.

Beesley, K. R., & Karttunen, L. (2003). *Finite state morphology*. Stanford: CSLI Publications.

Bentley, J., Knuth, D., & McIlroy, D. (1986). Programming pearls. *Communications of the ACM, 6*(29), 471–483.

Berkson, J. (1944). Application of the logistic function to bio-assay. *Journal of the American Statistical Association, 39*(227), 357–365.

Bescherelle, M. (1980). *L'art de conjuguer*. Paris: Hatier.

Bilange, E. (1992). *Dialogue personne-machine*. Paris: Hermès.

Bird, S., Klein, E., & Loper, E. (2009). *Natural language processing with Python*. Sebastopol: O'Reilly Media.

Bizer, C., Lehmann, J., Kobilarov, G., Auer, S., Becker, C., Cyganiak, R., & Hellmann, S. (2009). DBpedia—a crystallization point for the web of data. *Journal of Web Semantics, 7*, 154–165.

Björkelund, A., Bohnet, B., Hafdell, L., & Nugues, P. (2010). A high-performance syntactic and semantic dependency parser. In *Coling 2010: Demonstration volume*, Beijing, (pp. 33–36). Coling 2010 Organizing Committee.

Björkelund, A., Hafdell, L., & Nugues, P. (2009). Multilingual semantic role labeling. In *Proceedings of the thirteenth conference on computational natural language learning (CoNLL-2009)*, Boulder, (pp. 43–48).

Björkelund, A., & Nugues, P. (2011). Exploring lexicalized features for coreference resolution. In *Proceedings of the 15th conference on computational natural language learning (CoNLL-2011): Shared task*, Portland (pp. 45–50).

Black, E., Abney, S., Flickenger, D., Gdaniec, C., Grishman, R., Harrison, P., Hindle, D., Ingria, R., Jelinek, F., Klavans, J., Liberman, M., Marcus, M., Roukos, S., Santorini, B., & Strzalkowski, T. (1991). A procedure for quantitatively comparing the syntactic coverage of English grammars. In *Speech and natural language: Proceedings of a workshop*, Pacific Grove (pp. 306–311). San Mateo: DARPA/Morgan Kaufmann.

Blackburn, P., & Bos, J. (2005). *Representation and inference for natural language. A first course in computational semantics*. Stanford: CSLI Publications.

Boizumault, P. (1988). *Prolog, l'implantation*. Paris: Masson.

Boizumault, P. (1993). *The implementation of Prolog*. Princeton: Princeton University Press.

Boscovich, R. J. (1770). *Voyage astronomique et géographique dans l'État de l'Église*. Paris: N. M. Tilliard.

Boser, B., Guyon, I., & Vapnik, V. (1992). A training algorithm for optimal margin classifiers. In *Proceedings of the fifth annual workshop on computational learning theory*, Pittsburgh (pp. 144–152). ACM.

Boyer, M. (1988). Towards functional logic grammars. In V. Dahl & P. Saint-Dizier (Eds.), *Natural language understanding and logic programming, II* (pp. 45–61). Amsterdam: North-Holland.

Brants, S., Dipper, S., Hansen, S., Lezius, W., & Smith, G. (2002). The TIGER treebank. In *Proceedings of the first workshop on treebanks and linguistic theories*, Sozopol.

Brants, T., Popat, A. C., Xu, P., Och, F. J., & Dean, J. (2007). Large language models in machine translation. In *Proceedings of the 2007 joint conference on empirical methods in natural language processing and computational natural language learning*, Prague (pp. 858–867).

Bratko, I. (2012). *Prolog programming for artificial intelligence* (4th ed.). Harlow: Addison-Wesley.

Brill, E. (1995). Transformation-based error-driven learning and natural language processing: A case study in part-of-speech tagging. *Computational Linguistics, 21*(4), 543–565.

Brin, S., & Page, L. (1998). The anatomy of a large-scale hypertextual web search engine. *Computer Networks, 30*(1–7), 107–117. Proceedings of WWW7.

Bröker, N. (1998). How to define a context-free backbone for DGs: Implementing a DG in the LFG formalism. In S. Kahane & A. Polguère (Eds.), *Processing of dependency-based grammars. Proceedings of the workshop COLING-ACL*, Montréal (pp. 29–38).

Brown, P. F., Della Pietra, S. A., Della Pietra, V. J., & Mercer, R. L. (1993). The mathematics of statistical machine translation: Parameter estimation. *Computational Linguistics, 19*(2), 263–311.

Brown, P. F., Della Pietra, V. J., deSouza, P. V., Lai, J. C., & Mercer, R. L. (1992). Class-based n-gram models of natural language. *Computational Linguistics, 18*(4), 467–489.

Buchholz, S., & Marsi, E. (2006). CoNLL-X shared task on multilingual dependency parsing. In *Proceedings of the tenth conference on computational natural language learning (CoNLL-X)*, New York City (pp. 149–164). Association for Computational Linguistics.

Bühler, K. (1934). *Sprachtheorie. Die Darstellungsfunktion der Sprache*. Jena: Verlag von Gustav Fischer.

Bühler, K. (1982). *Sprachtheorie. Die Darstellungsfunktion der Sprache*. Stuttgart: UTB. First edition 1934.

Bunescu, R., & Paşca, M. (2006). Using encyclopedic knowledge for named entity disambiguation. In *Proceedings of the 11th conference of the European chapter of the association for computational linguistics*, Trento (pp. 9–16). Association for Computational Linguistics.

Burke, E., & Foxley, E. (1996). *Logic and its applications*. London: Prentice Hall.

Busa, R. (1974). *Index Thomisticus: Sancti Thomae Aquinatis operum omnium indices et concordantiae in quibus verborum omnium et singulorum formae et lemmata cum suis frequentiis et contextibus variis modis referuntur*. Stuttgart-Bad Cannstatt: Frommann-Holzboog.

Busa, R. (1996). *Thomae Aquinatis Opera omnia cum hypertextibus in CD-ROM*. Milan: Editoria Elettronica Editel.

Busa, R. (2009). From punched cards to treebanks: 60 years of computational linguistics. In *Website of the eighth international workshop on treebanks and linguistic theories*, Milan.

Candito, M., Crabbé, B., Denis, P., & Guérin, F. (2009). Analyse syntaxique du français: Des constituants aux dépendances. In *TALN 2009*, Senlis.

Carberry, S. (1990). *Plan recognition in natural language dialogue*. Cambridge, MA: MIT.

Carlberger, J., Domeij, R., Kann, V., & Knutsson, O. (2004). The development and performance of a grammar checker for Swedish: A language engineering perspective. Technical report, Kungliga Tekniska högskolan, Stockholm.

Carlberger, J., & Kann, V. (1999). Implementing an efficient part-of-speech tagger. *Software – Practice and Experience, 29*(2), 815–832.

Carlson, L., Marcu, D., & Okurowski, M. (2003). Building a discourse-tagged corpus in the framework of rhetorical structure theory. In *Current and new directions in discourse and dialogue* (Text, speech and language technology, Vol. 22, pp. 85–112). Dordrecht: Springer.

Carreras, X. (2007). Experiments with a higher-order projective dependency parser. In *Proceedings of the CoNLL shared task session of EMNLP-CoNLL*, Prague (pp. 957–961).

Cauchy, A. (1847). Méthode générale pour la résolution des systèmes d'équations simultanées. *Comptes Rendus de l'Académie des Sciences de Paris, 25*, 536–538.

Chang, C.-C., & Lin, C.-J. (2011). LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology, 2*, 27:1–27:27.

Chanod, J.-P. (1994). Finite-state composition of French verb morphology. Technical report MLTT-005, Rank Xerox Research Centre, Grenoble.

Charniak, E. (1993). *Statistical language learning*. Cambridge, MA: MIT.

Charniak, E. (1997a). Statistical parsing with a context-free grammar and word statistics. In *Proceedings of the fourteenth national conference on artificial intelligence*, Providence. Menlo Park: AAAI/MIT.

Charniak, E. (1997b). Statistical techniques for natural language parsing. *AI Magazine, 18*, 33–44.

Charniak, E. (2000). A maximum-entropy-inspired parser. In *Proceedings of the first meeting of the North American chapter of the ACL*, Seattle (pp. 132–139).

Charniak, E., Goldwater, S., & Johnson, M. (1998). Edge-based best-first chart parsing. In *Proceedings of the sixth workshop on very large corpora*, Montréal (pp. 127–133).

Chen, S. F., & Goodman, J. (1998). An empirical study of smoothing techniques for language modeling. Technical report TR-10-98, Harvard University, Cambridge, MA.

Chervel, A. (1979). Rhétorique et grammaire: Petite histoire du circonstanciel. *Langue française, 41*, 5–19.

Chinchor, N. (1997). MUC-7 named entity task definition. Technical report, Science Applications International Corporation. Cited November 2, 2005, from www.itl.nist.gov/iaui/894.02/related_projects/muc/proceedings/ne_task.html

Chomsky, N. (1957). *Syntactic structures*. The Hague: Mouton.

Chomsky, N. (1981). *Lectures on government and binding*. Dordrecht: Foris.

Christiansen, T., Foy, B. D., Wall, L., & Orwant, J. (2012). *Programming Perl* (4th ed.). Sebastopol: O'Reilly Media.

Chrupała, G. (2006). Simple data-driven context-sensitive lemmatization. In *Proceedings of SEPLN*, Zaragoza.

Church, A. (1941). *The calculi of lambda-conversion*. Princeton: Princeton University Press.

Church, K. W. (1988). A stochastic parts program and noun phrase parser for unrestricted text. In *Proceedings of the second conference on applied natural language processing*, Austin (pp. 136–143). ACL.

Church, K. W., & Hanks, P. (1990). Word association norms, mutual information, and lexicography. *Computational Linguistics, 16*(1), 22–29.

Church, K. W., & Mercer, R. L. (1993). Introduction to the special issue on computational linguistics using large corpora. *Computational Linguistics, 19*(1), 1–24.
Civit Torruella, M. (2002). Guía para la anotación morfológica del corpus CLiC–TALP (versión 3). Technical report, Universitat Politècnica de Catalunya, Barcelona.
Clarkson, P. R., & Rosenfeld, R. (1997). Statistical language modeling using the CMU-Cambridge toolkit. In *Proceedings ESCA Eurospeech*, Rhodes.
Clocksin, W. F., & Mellish, C. S. (2003). *Programming in Prolog: Using the ISO standard* (5th ed.). Berlin/Heidelberg/New York, Springer.
Collins, M. J. (1996). A new statistical parser based on bigram lexical dependencies. In *Proceedings of the 34th annual meeting of the association for computational linguistics*, Santa Cruz (pp. 184–191).
Collins, M. J. (1999). *Head-driven statistical models for natural language parsing*. PhD thesis, University of Pennsylvania.
Collins, M. J. (2002). Discriminative training methods for hidden Markov models: Theory and experiments with perceptron algorithms. In *Proceedings of the 2002 conference on empirical methods in natural language processing*, Prague (pp. 1–8).
Colmerauer, A. (1970). Les systèmes-Q ou un formalisme pour analyser et synthétiser des phrases sur ordinateur. Publication interne 43, Département d'informatique, Université de Montréal.
Colmerauer, A. (1978). Metamorphosis grammars. In L. Bolc (Ed.), *Natural language communication with computers* (Lecture notes in computer science, Vol. 63, pp. 133–189). Berlin/ Heidelberg/New York: Springer.
Colmerauer, A. (1982). An interesting subset of natural language. In K. L. Clark & S. Tärnlund (Eds.), *Logic programming* (pp. 45–66). London: Academic.
Colmerauer, A., Kanoui, H., Pasero, R., & Roussel, P. (1972). Un système de communication en français. Rapport préliminaire de fin de contrat IRIA, Groupe Intelligence Artificielle, Faculté des Sciences de Luminy, Université d'Aix-Marseille II.
Colmerauer, A., & Roussel, P. (1996). The birth of Prolog. In T. J. Bergin & R. G. Gibson (Eds.), *History of programming languages II*. New York: ACM/Reading: Addison-Wesley.
Constant, P. (1991). *Analyse syntaxique par couches*. Thèse de doctorat, École Nationale Supérieure des Télécommunications, Paris.
Cooper, D. (1999). Corpora: KWIC concordances with Perl. CORPORA mailing list archive, Concordancing thread.
Corbett, E. P. J., & Connors, R. J. (1999). *Classical rhetoric for the modern student* (4th ed.). New York, Oxford University Press.
Corston-Oliver, S. (1998). *Computing representations of the structure of written discourse*. PhD thesis, Linguistics Department, the University of California, Santa Barbara.
Cortes, C., & Vapnik, V. (1995). Support-vector networks. *Machine Learning, 20*(3), 273–297.
Coulthard, M. (1985). *An introduction to discourse analysis* (2nd ed.). Harlow: Longman.
Covington, M. A. (1989). GULP 2.0, an extension of Prolog for unification-based grammar. Research report AI-1989-01, Artificial Intelligence Programs, University of Georgia.
Covington, M. A. (1990). Parsing discontinuous constituents in dependency grammar. *Computational Linguistics, 16*(4), 234–236.
Covington, M. A. (1994a). Discontinuous dependency parsing of free and fixed word order: Work in progress. Research report AI-1994-02, Artificial Intelligence Programs, The University of Georgia.
Covington, M. A. (1994b). *Natural language processing for Prolog programmers*. Upper Saddle River: Prentice Hall.
Covington, M. A. (2001). A fundamental algorithm for dependency parsing. In *Proceedings of the 39th annual ACM Southeast conference*, Athens (pp. 95–102). Version with corrections: Retrieved October 25, 2013, from http://www.ai.uga.edu/mc/dgpacmcorr.pdf
Covington, M. A., Nute, D., & Vellino, A. (1997). *Prolog programming in depth*. Upper Saddle River: Prentice Hall.
Cozannet, A. (1992). A model for task driven oral dialogue. In *Proceedings of the second international conference on spoken language processing (ICSLP)*, Banff (pp. 1451–1454).

Crystal, D. (1997). *The Cambridge encyclopedia of language* (2nd ed.). Cambridge: Cambridge University Press.

Cucerzan, S. (2007). Large-scale named entity disambiguation based on wikipedia data. In *Proceedings of the 2007 joint conference on empirical methods in natural language processing and computational natural language learning*, Prague (pp. 708–716). Association for Computational Linguistics.

Cull, R. (1840). *Garrick's mode of reading the liturgy of the church of England*. London: John W. Parker.

d'Arc, S. J. (Ed.). (1970). *Concordance de la Bible, Nouveau Testament*. Paris: Éditions du Cerf – Desclées De Brouwer.

Davidson, D. (1966). The logical form of action sentences. In N. Rescher (Ed.), *The logic of decision and action*. Pittsburgh: University of Pittsburgh Press.

Davis, M., & Whistler, K. (2009). Unicode collation algorithm. Unicode technical standard 10, The Unicode Consortium. Version 5.2.

de la Briandais, R. (1959). File searching using variable length keys. In *Proceedings of the Western joint computer conference*, San Francisco (pp. 295–298). AFIPS.

Delahaye, J.-P. (1986). *Outils logiques pour l'intelligence artificielle*. Paris: Eyrolles.

Deransart, P., Ed-Dbali, A. A., & Cervoni, L. (1996). *Prolog: The standard, reference manual*. Berlin/Heidelberg/New York: Springer.

Dermatas, E., & Kokkinakis, G. K. (1995). Automatic stochastic tagging of natural language texts. *Computational Linguistics, 21*(2), 137–163.

Domergue, U. (1782). *Grammaire française simplifiée, Nouvelle édition*. Paris: Durand.

Ducrot, O., & Schaeffer, J.-M. (Eds.). (1995). *Nouveau dictionnaire encyclopédique des sciences du langage*. Paris: Éditions du Seuil.

Dunning, T. (1993). Accurate methods for the statistics of surprise and coincidence. *Computational Linguistics, 19*(1), 61–74.

Dutoit, D. (1992). A set-theoretic approach to lexical semantics. In *Proceedings of the 15th international conference on computational linguistics, COLING-92*, Nantes (Vol. III, pp. 982–987).

Earley, J. C. (1970). An efficient context-free parsing algorithm. *Communications of the ACM, 13*(2), 94–102.

Eckert, W. (1996). *Gesprochener Mensch-Machine-Dialog*. Aachen: Shaker Verlag.

Einarsson, J. (1976). Talbankens skriftspråkskonkordans. Technical report, Lund University, Institutionen för nordiska språk, Lund.

Eisner, J. (1996). Three new probabilistic models for dependency parsing: An exploration. In *Proceedings of the 16th international conference on computational linguistics (COLING-96)*, Copenhagen (pp. 340–345).

Eisner, J. (2000). Bilexical grammars and their cubic-time parsing algorithms. In H. C. Bunt & A. Nijholt (Eds.), *Advances in probabilistic and other parsing technologies* (pp. 29–62). Kluwer Academic Publishers.

Ejerhed, E. (1988). Finding clauses in unrestricted text by finitary and stochastic methods. In *Second conference on applied natural language processing*, Austin (pp. 219–227). ACL.

Ejerhed, E., Källgren, G., Wennstedt, O., & Åström, M. (1992). The linguistic annotation system of the Stockholm-Umeå corpus project. Technical report 33, Department of General Linguistics, University of Umeå.

El Guedj, P.-O. (1996). *Analyse syntaxique par charts combinant règles de dépendance et règles syntagmatiques*. PhD thesis, Université de Caen.

Estoup, J.-B. (1912). *Gammes sténographiques: Recueil de textes choisis pour l'acquisition méthodique de la vitesse* (3e ed.). Paris: Institut sténographique.

Fan, R.-E., Chang, K.-W., Hsieh, C.-J., Wang, X.-R., & Lin, C.-J. (2008). LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research, 9*, 1871–1874.

Fan, J., Kalyanpur, A., Gondek, D. C., & Ferrucci, D. A. (2012). Automatic knowledge extraction from documents. *IBM Journal of Research and Development, 56*(3.4), 5:1–5:10.

Fano, R. M. (1961). *Transmission of information: A statistical theory of communications*. New York: MIT.

Fellbaum, C. (Ed.). (1998). *WordNet: An electronic lexical database (language, speech and communication)*. Cambridge, MA: MIT.

Ferrucci, D. A. (2012). Introduction to "This is Watson". *IBM Journal of Research and Development, 56*(3.4), 1:1–1:15.

Fikes, R., & Nilsson, N. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence, 2*(3/4), 189–208.

Fillmore, C. J. (1968). The case for case. In E. Bach & R. T. Harms (Eds.), *Universals in linguistic theory*, (pp. 1–88). New York: Holt, Rinehart and Winston.

Fillmore, C. J. (1976). Frame semantics and the nature of language. *Annals of the New York Academy of Sciences: Conference on the Origin and Development of Language and Speech, 280*, 20–32.

Finkel, J. R., Grenager, T., & Manning, C. (2005). Incorporating non-local information into information extraction systems by Gibbs sampling. In *Proceedings of the 43nd annual meeting of the association for computational linguistics (ACL 2005)*, Ann Arbor (pp. 363–370).

Francis, W. N., & Kucera, H. (1982). *Frequency analysis of English usage*. Boston: Houghton Mifflin.

Franz, A. (1996). *Automatic ambiguity resolution in natural language processing: An empirical approach* (Lecture notes in artificial intelligence, Vol. 1171). Berlin/Heidelberg/New York: Springer.

Franz, A., & Brants, T. (2006). All our n-gram are belong to you. Retrieved November 7, 2013, from http://googleresearch.blogspot.se/2006/08/all-our-n-gram-are-belong-to-you.html

Frege, G. (1879). *Begriffsschrift: Eine der arithmetischen nachgebildete Formelsprache des reinen Denkens*. Halle: Verlag von Louis Nebert.

Friedl, J. E. F. (2006). *Mastering regular expressions* (3rd ed.). Sebastopol: O'Reilly Media.

Fromkin, V. (Ed.). (2000). *Linguistics: An introduction to linguistic theory*. Oxford: Blackwell.

Fromkin, V., Rodman, R., & Hyams, N. (2010). *An introduction to language* (9th ed.). Boston: Wadsworth, Cengage Learning.

Frühwirth, T. (1998). Theory and practice of constraint handling rules. *Journal of Logic Programming, 37*(1–3), 95–138.

Gagnon, M., & Lapalme, G. (1996). From conceptual time to linguistic time. *Computational Linguistics, 22*(1), 91–127.

Gal, A., Lapalme, G., & Saint-Dizier, P. (1989). *Prolog pour l'analyse automatique du langage naturel*. Paris: Eyrolles.

Gal, A., Lapalme, G., Saint-Dizier, P., & Somers, H. (1991). *Prolog for natural language processing*. Chichester: Wiley.

Gale, W. A., & Church, K. W. (1993). A program for aligning sentences in bilingual corpora. *Computational Linguistics, 19*(1), 75–102.

Galton, F. (1886). Regression towards mediocrity in hereditary stature. *Journal of the Anthropological Institute, 15*, 246–263.

Gazdar, G., & Mellish, C. (1989). *Natural language processing in Prolog: An introduction to computational linguistics*. Wokingham: Addison-Wesley.

Gendner, V., Illouz, G., Jardino, M., Monceaux, L., Paroubek, P., Robba, I., & Vilnat, A. (2003). PEAS, the first instantiation of a comparative framework for evaluating parsers of French. In *Proceedings of the research note sessions of the 10th conference of the European chapter of the association for computational linguistics (EACL'03)*, Budapest (Vol. 2, pp. 95–98).

Giannesini, F., Kanoui, H., Pasero, R., & van Caneghem, M. (1985). *Prolog*. Paris: Interéditions.

Giménez, J., & Màrquez, L. (2004). SVMTool: A general POS tagger generator based on support vector machines. In *Proceedings of the 4th international conference on language resources and evaluation (LREC'04)*, Lisbon (pp. 43–46).

Godart-Wendling, B., Ildefonse, F., Pariente, J.-C., & Rosier, I. (1998). Penser le principe de compositionnalité: éléments de réflexion historiques et épistémologiques. *Traitement automatique des langues, 39*(1), 9–34.

Godéreaux, C., Diebel, K., El Guedj, P.-O., Revolta, F., & Nugues, P. (1996). An interactive spoken dialog interface to virtual worlds. In J. H. Connolly & L. Pemberton (Eds.), *Linguistic concepts and methods in CSCW* (Computer supported cooperative work, chapter 13, pp. 177–200). Berlin/Heidelberg/New York: Springer.

Godéreaux, C., El Guedj, P.-O., Revolta, F., & Nugues, P. (1998). Ulysse: An interactive, spoken dialogue interface to navigate in virtual worlds. Lexical, syntactic, and semantic issues. In J. Vince & R. Earnshaw (Eds.), *Virtual worlds on the Internet* (chapter 4, pp. 53–70, 308–312). Los Alamitos: IEEE Computer Society.

Good, I. J. (1953). The population frequencies of species and the estimation of population parameters. *Biometrika, 40*(16), 237–264.

Gosselin, L. (1996). *Sémantique de la temporalité en français: Un modèle calculatoire et cognitif du temps et de l'aspect*. Louvain-la-Neuve: Duculot.

Graham, S. L., Harrison, M. A., & Ruzzo, W. L. (1980). An improved context-free recognizer. *ACM Transactions on Programming Languages and Systems, 2*(3), 415–462.

Grefenstette, G., & Tapanainen, P. (1994). What is a word, what is a sentence? Problems of tokenization. MLTT technical report 4, Xerox.

Grosz, B. J., Joshi, A. K., & Weinstein, S. (1995). Centering: A framework for modeling the local coherence of discourse. *Computational Linguistics, 21*(2), 203–225.

Grosz, B. J., & Sidner, C. L. (1986). Attention, intention, and the structure of discourse. *Computational Linguistics, 12*(3), 175–204.

Habermas, J. (1988). *Nachmetaphysisches Denken*. Frankfurt am Main: Suhrkamp.

Haegeman, L., & Gueron, J. (1999). *English grammar: A generative perspective* (Number 14 in Blackwell textbooks in linguistics). Malden: Blackwell.

Halácsy, P., Kornai, A., & Oravecz, C. (2007). HunPos – an open source trigram tagger. In *Proceedings of the 45th annual meeting of the association for computational linguistics companion volume proceedings of the demo and poster sessions*, Prague (pp. 209–212).

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., & Witten, I. H. (2009). The WEKA data mining software: An update. *SIGKDD Explorations, 11*(1), 10–18.

Harper, M. H., Hockema, S. A., & White, C. M. (1999). Enhanced constraint dependency grammar parsers. In *Proceedings of the IASTED international conference on artificial intelligence and soft computing*, Honolulu.

Harris, Z. (1962). *String analysis of sentence structure*. The Hague: Mouton.

Harris, R., & Taylor, T. J. (1997). *Landmarks in linguistic thought, the Western tradition from Socrates to Saussure* (2nd ed.). London: Routledge.

Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The elements of statistical learning: Data mining, inference, and prediction* (2nd ed.). New York: Springer.

Hausser, R. (2000). *Grundlagen der Computerlinguistik. Mensch-Maschine-Kommunikation in natürlicher Sprache*. Berlin/Heidelberg/New York: Springer.

Hausser, R. (2014). *Foundations of computational linguistics. Human-computer communication in natural language* (3rd ed.). Berlin/Heidelberg/New York: Springer.

Hays, D. G. (1964). Dependency theory: A formalism and some observations. *Language, 40*(4), 511–525.

Hellwig, P. (1980). PLAIN – a program system for dependency analysis and for simulating natural language inference. In L. Bolc (Ed.), *Representation and processing of natural language* (pp. 271–376). München: Hanser.

Hellwig, P. (1986). Dependency unification grammar (DUG). In *Proceedings of the 11th international conference on computational linguistics (COLING 86)*, Bonn (pp. 195–198).

Herbrand, J. (1930). Recherches sur la théorie de la démonstration. *Travaux de la Société des Sciences et des Lettres de Varsovie, Classe III Sciences mathématiques et physiques*, 33.

Heringer, H.-J. (1993). Dependency syntax – basic ideas and the classical model. In J. Jacobs, A. von Stechow, W. Sternefeld, & T. Venneman (Eds.), *Syntax – an international handbook of contemporary research* (Vol. 1, chapter 12, pp. 298–316). Berlin/New York: Walter de Gruyter.

Hindle, D., & Rooth, M. (1993). Structural ambiguity and lexical relations. *Computational Linguistics, 19*(1), 103–120.

Hintikka, J. (1962). *Knowledge and belief, an introduction to the logic of the two notions*. Ithaca: Cornell University Press.
Hirschman, L., & Chinchor, N. (1997). MUC-7 coreference task definition. Technical report, Science Applications International Corporation.
Hjelmslev, L. (1935–1937). *La catégorie des cas. Étude de grammaire générale: Volume VII(1), IX(2) of Acta Jutlandica*. Aarhus: Universitetsforlaget i Aarhus.
Hjelmslev, L. (1943). *Omkring sprogteoriens grundlæggelse*. Festskrift udgivet af Københavns Universiteit, Copenhagen. English translation Prolegomena to a Theory of Language.
Hobbs, J. R., Appelt, D. E., Bear, J., Israel, D., Kameyama, M., Stickel, M., & Tyson, M. (1997). FASTUS: A cascaded finite-state transducer for extracting information from natural-language text. In E. Roche & Y. Schabes (Eds.), *Finite-state language processing* (chapter 13, pp. 383–406). Cambridge: MIT.
Hoffart, J., Yosef, M. A., Bordino, I., Fürstenau, H., Pinkal, M., Spaniol, M., Taneva, B., Thater, S., & Weikum, G. (2011). Robust disambiguation of named entities in text. In *Proceedings of the 2011 conference on empirical methods in natural language processing*, Edinburgh (pp. 782–792).
Hopcroft, J. E., Motwani, R., & Ullman, J. D. (2007). *Introduction to automata theory, languages, and computation* (3rd ed.). Boston: Addison-Wesley.
Hornby, A. S. (Ed.). (1974). *Oxford advanced learner's dictionary of current English* (3rd ed.). Oxford: Oxford University Press.
Hornby, A. S. (Ed.). (1995). *Oxford advanced learner's dictionary of current English* (5th ed.). Oxford: Oxford University Press.
Huang, X., Acero, A., Alleva, F., Hwang, M.-Y., Jiang, L., & Mahaja, M. (1995). Microsoft highly intelligent speech recognizer: Whisper. In *Proceedings of the international conference on acoustics, speech, and signal processing*, Detroit.
Huang, J., Gao, J., Miao, J., Li, X., Wang, K., & Behr, F. (2010). Exploring web scale language models for search query processing. In *Proceedings of the 19th international World Wide Web conference*, Raleigh (pp. 451–460).
Huls, C., Claassen, W., & Bos, E. (1995). Automatic referent resolution of deictic and anaphoric expressions. *Computational Linguistics, 21*(1), 59–79.
Ide, N., & Véronis, J. (1995). *Text encoding initiative: Background and context*. Dordrecht: Kluwer Academic.
Ide, N., & Véronis, J. (1998). Introduction to the special issue on word sense disambiguation: The state of the art. *Computational Linguistics, 24*(1), 1–40.
Imbs, P., & Quemada, B. (Eds.). (1971–1994). *Trésor de la langue française. Dictionnaire de la langue française du XIXe et du XXe siècle (1789–1960)*. Éditions du CNRS puis Gallimard, Paris. 16 volumes.
Ingria, B., & Pustejovsky, J. (2004). TimeML: A formal specification language for events and temporal expressions. Cited April 13, 2010, from http://www.timeml.org/site/publications/timeMLdocs/timeml_1.2.html
Jackendoff, R. (1990). *Semantic structures*. Cambridge, MA: MIT.
Jacob, A. (Ed.). (1989). *Encyclopédie philosophique universelle*. Paris: Presses Universitaires de France.
James, G., Witten, D., Hastie, T., & Tibshirani, R. (2013). *An introduction to statistical learning: With applications in R*. New York: Springer.
Järvinen, T., & Tapanainen, P. (1997). A dependency parser for English. Technical report TR-1, Department of General Linguistics, University of Helsinki.
Jekat, S., Klein, A., Maier, E., Maleck, I., Mast, M., & Quantz, J. (1995). Dialogue acts in Verbmobil. Verbmobil-report 65, Universität Hamburg, DFKI, Universität Erlangen, TU Berlin.
Jelinek, F. (1990). Self-organized language modeling for speech recognition. In A. Waibel & K.-F. Lee (Eds.), *Readings in speech recognition*. San Mateo: Morgan Kaufmann. Reprinted from an IBM report, 1985.
Jelinek, F. (1997). *Statistical methods for speech recognition*. Cambridge, MA: MIT.

Jelinek, F., & Mercer, R. L. (1980). Interpolated estimation of Markov source parameters from sparse data. In E. S. Gelsema & L. N. Kanal (Eds.), *Pattern recognition in practice* (pp. 38–397). Amsterdam: North-Holland.

Jensen, K., Heidorn, G., & Richardson, S. (Eds.). (1993). *Natural language processing: The PLNLP approach*. Boston: Kluwer Academic.

Joachims, T. (2002). *Learning to classify text using support vector machines*. Boston: Kluwer Academic.

Johansson, R., Berglund, A., Danielsson, M., & Nugues, P. (2005). Automatic text-to-scene conversion in the traffic accident domain. In *IJCAI-05, proceedings of the nineteenth international joint conference on artificial intelligence*, Edinburgh (pp. 1073–1078).

Johansson, R., & Nugues, P. (2007a). Extended constituent-to-dependency conversion for English. In J. Nivre, H.-J. Kaalep, K. Muischnek, & M. Koit (Eds.), *NODALIDA 2007 conference proceedings*, Tartu (pp. 105–112).

Johansson, R., & Nugues, P. (2007b). Incremental dependency parsing using online learning. In *Proceedings of the CoNLL shared task session of EMNLP-CoNLL*, Prague (pp. 1134–1138).

Johansson, R., & Nugues, P. (2008a). Dependency-based semantic role labeling of PropBank. In *Proceedings of the 2008 conference on empirical methods in natural language processing (EMNLP 2008)*, Honolulu (pp. 69–78).

Johansson, R., & Nugues, P. (2008b). Dependency-based syntactic–semantic analysis with PropBank and NomBank. In *Proceedings of CoNLL-2008: The twelfth conference on computational natural language learning*, Manchester (pp. 183–187).

Johnson, C. D. (1972). *Formal aspects of phonological description*. The Hague: Mouton.

Johnson, M. (1998). PCFG models of linguistic tree representation. *Computational Linguistics, 24*(4), 613–632.

Joshi, A. K., & Hopely, P. (1999). A parser from antiquity: An early application of finite state transducers to natural language processing. In A. Kornai (Ed.), *Extended finite state models of language* (Studies in natural language processing, pp. 6–15). Cambridge: Cambridge University Press.

Jurafsky, D., & Martin, J. H. (2008). *Speech and language processing, an introduction to natural language processing, computational linguistics, and speech recognition* (2nd ed.). Upper Saddle River: Pearson Education.

Kaeding, F. W. (1897). *Häufigkeitswörterbuch der deutschen Sprache*. Steglitz bei Berlin: Selbstverlag des Herausgebers.

Kameyama, M. (1997). Recognizing referential links: An information extraction perspective. In R. Mitkov & B. Boguraev (Eds.), *Proceedings of ACL workshop on operational factors in practical, robust anaphora resolution for unrestricted texts*, Madrid (pp. 46–53).

Kamp, H., & Reyle, U. (1993). *From discourse to logic: Introduction to modeltheoretic semantics of natural language, formal logic and discourse representation theory*. Dordrecht: Kluwer Academic.

Kaplan, R. M., & Bresnan, J. (1982). Lexical-functional grammar: A formal system for grammatical representation. In J. Bresnan (Ed.), *The mental representation of grammatical relations* (pp. 173–281). Cambridge, MA: MIT.

Kaplan, R. M., & Kay, M. (1994). Regular models of phonological rule systems. *Computational Linguistics, 20*(3), 331–378.

Karttunen, L. (1983). KIMMO: A general morphological processor. *Texas Linguistic Forum, 22*, 163–186.

Karttunen, L. (1994). Constructing lexical transducers. In *Proceedings of the 15th conference on computational linguistics, COLING-94*, Kyoto (Vol. 1, pp. 406–411).

Karttunen, L., Kaplan, R. M., & Zaenen, A. (1992). Two-level morphology with composition. In *Proceedings of the 15th conference on computational linguistics, COLING-92*, Nantes (Vol. 1, pp. 141–148).

Kasami, T. (1965). An efficient recognition and syntax analysis algorithm for context-free languages. Technical report AFCRL-65-758, Air Force Cambridge Research Laboratory, Bedford, MA. Cited from Wikipedia. Retrieved December 26, 2013.

Katz, S. M. (1987). Estimation of probabilities from sparse data for the language model component of a speech recognizer. *IEEE Transactions on Acoustics, Speech, and Signal Processing, 35*(3), 400–401.

Kernighan, M. D., Church, K. W., & Gale, W. A. (1990). A spelling correction program based on a noisy channel model. In *Papers presented to the 13th international conference on computational linguistics (COLING-90)*, Helsinki (Vol. II, pp. 205–210).

Kingsbury, P., Palmer, M., & Marcus, M. (2002). Adding semantic annotation to the Penn Treebank. In *Proceedings of the human language technology conference*, San Diego.

Kiraz, G. A. (2001). *Computational nonlinear morphology: With emphasis on semitic languages* (Studies in natural language processing). Cambridge: Cambridge University Press.

Kiss, T., & Strunk, J. (2006). Unsupervised multilingual sentence boundary detection. *Computational Linguistics, 32*(4), 485–525.

Kleene, S. C. (1956). Representation of events in nerve nets and finite automata. In C. E. Shannon & J. McCarthy (Eds.), *Automata studies* (pp. 3–42). Princeton: Princeton University Press.

Klein, D., & Manning, C. D. (2003). Accurate unlexicalized parsing. In *Proceedings of the 41st meeting of the association for computational linguistics*, Sapporo (pp. 423–430).

Klein, S., & Simmons, R. (1963). A computational approach to grammatical coding of English words. *Journal of the ACM, 10*(3), 334–347.

Knuth, D. E. (1986). *The TeXbook*. Reading: Addison-Wesley.

Koch, U. (1993). The enhancement of a dependency parser for Latin. Technical report AI-1993-03, Artificial Intelligence Programs, University of Georgia.

Koehn, P. (2010). *Statistical machine translation*. Cambridge: Cambridge University Press.

Kornai, A. (Ed.). (1999). *Extended finite state models of language* (Studies in natural language processing). Cambridge: Cambridge University Press.

Koskenniemi, K. (1983). Two-level morphology: A general computation model for word-form recognition and production. Technical report 11, Department of General Linguistics, University of Helsinki.

Kowalski, R. A., & Kuehner, D. (1971). Linear resolution with selection function. *Artificial Intelligence, 2*, 227–260.

Kübler, S., McDonald, R., & Nivre, J. (2009). *Dependency parsing*. San Rafael: Morgan and Claypool Publishers.

Kudoh, T., & Matsumoto, Y. (2000). Use of support vector learning for chunk identification. In *Proceedings of CoNLL-2000 and LLL-2000*, Lisbon (pp. 142–144).

Kunze, J. (1967). Die Behandlung nicht-projektiver Strukturen bei der syntaktischen Analyse und Synthese des englischen und des deutschen. In *MASPEREVOD-67: Internationales Symposium der Mitgliedsländer des RGW*, Budapest (pp. 2–15).

Kunze, J. (1975). *Abhängigkeitsgrammatik*. Berlin: Akademieverlag.

Lafferty, J., McCallum, A., & Pereira, F. (2001). Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proceedings of the eighteenth international conference on machine learning (ICML-01)*, Williamstown (pp. 282–289). Morgan Kaufmann Publishers.

Lallot, J. (Ed.). (1998). *La grammaire de Denys le Thrace*. CNRS Éditions, Collection Science du langage, Paris, 2e edition. Text in Greek, translated in French by Jean Lallot.

Landes, S., Leacock, C., & Tengi, R. (1998). Building semantic concordances. In C. Fellbaum (Ed.), *WordNet: An electronic lexical database*. Cambridge: MIT.

Laplace, P. (1820). *Théorie analytique des probabilités* (3rd ed.). Paris: Coursier.

Lasnik, H., Depiante, M. A., & Stepanov, A. (2000). *Syntactic structures revisited: Contemporary lectures on classic transformational theory* (Current studies in linguistics, Vol. 33). Cambridge, MA: MIT.

Lecerf, Y. (1960a). Programme de conflits, modèles de conflits. *Traduction automatique, 1*(4), 11–18.

Lecerf, Y. (1960b). Programme de conflits, modèles de conflits. *Traduction automatique, 1*(5), 17–36.

Lecerf, Y., & Ihm, P. (1960). Éléments pour une grammaire générale des langues projectives. Technical report 1, Communauté européenne de l'énergie atomique, Groupe de recherches sur l'information scientifique automatique.

Legendre, A.-M. (1805). *Nouvelles méthodes pour la détermination des orbites des comètes*. Paris: Firmin Didot.

Lesk, M. (1986). Automatic sense disambiguation using machine readable dictionaries: How to tell a pine cone from an ice cream cone. In *Proceedings of the 5th annual international conference on systems documentation*, Toronto (pp. 24–26).

Levinson, S. (1983). *Pragmatics*. Cambridge: Cambridge University Press.

Lewis, D. D., Yang, Y., Rose, T. G., & Li, F. (2004). RCV1: A new benchmark collection for text categorization research. *Journal of Machine Learning Research, 5*, 361–397.

Lin, D. (1995). A dependency-based method for evaluating broad-coverage parsers. In *Proceedings of IJCAI-95*, Montreal (pp. 1420–1427).

Linke, A., Nussbaumer, M., & Portmann, P. R. (2004). *Studienbuch linguistik* (5th ed.). Tübingen: Niemeyer.

Lloyd, J. W. (1987). *Foundations of logic programming* (2nd ed.). Berlin/Heidelberg/New York: Springer.

Luo, X. (2005). On coreference resolution performance metrics. In *Proceedings of human language technology conference and conference on empirical methods in natural language processing*, Vancouver (pp. 25–32).

Magerman, D. M. (1994). *Natural language parsing as statistical pattern recognition*. PhD thesis, Stanford University.

Maier, D., & Warren, D. S. (1988). *Computing with logic, logic programming with Prolog*. Menlo Park: Benjamin/Cummings.

Malmberg, B. (1983). *Analyse du langage au XXe siècle. Théorie et méthodes*. Paris: Presses universitaires de France.

Malmberg, B. (1991). *Histoire de la linguistique. De Sumer à Saussure*. Paris: Presses universitaires de France.

Mann, W. C., & Thompson, S. A. (1987). Rhetorical structure theory: A theory of text organization. Technical report RS-87-190, Information Sciences Institute of the University of Southern California.

Mann, W. C., & Thompson, S. A. (1988). Rhetorical structure theory: Toward a functional theory of text organization. *Text, 8*, 243–281.

Manning, C. D., Raghavan, P., & Schütze, H. (2008). *Introduction to information retrieval*. Cambridge: Cambridge University Press.

Manning, C. D., & Schütze, H. (1999). *Foundations of statistical natural language processing*. Cambridge, MA: MIT.

Marcu, D. (1997). *The rhetorical parsing, summarization, and generation of natural language texts*. PhD thesis, Department of Computer Science, University of Toronto.

Marcus, M., Kim, G., Marcinkiewicz, M. A., MacIntyre, R., Bies, A., Ferguson, M., Katz, K., & Schasberger, B. (1994). The Penn Treebank: Annotating predicate argument structure. In *ARPA human language technology workshop*, Plainsboro.

Marcus, M., Marcinkiewicz, M. A., & Santorini, B. (1993). Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics, 19*(2), 313–330.

Martelli, A., & Montanari, U. (1982). An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems, 4*(2), 258–282.

Maruyama, H. (1990). Constraint dependency grammar and its weak generative capacity. *Computer Software, 7*(3), 50–59. (In Japanese).

Mast, M. (1993). *Ein Dialogmodul für ein Spracherkennungs- und Dialogsystem* (Dissertationen zur Künstlichen Intelligenz, Vol. 50). Sankt Augustin: Infix.

Mast, M., Kummert, F., Ehrlich, U., Fink, G. A., Kuhn, T., Niemann, H., & Sagerer, G. (1994). A speech understanding and dialog system with a homogeneous linguistic knowledge base. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 16*(2), 179–194.

Mauldin, M. L., & Leavitt, J. R. R. (1994). Web-agent related research at the center for machine translation. In *Proceedings of the ACM SIG on networked information discovery and retrieval*, McLean.

Maxwell, D. (1995). Unification dependency grammar. ftp://ftp.ling.ohio-state.edu/pub/HPSG/Papers/UDG/. Draft. Cited October 28, 2005.

McDonald, R. (2006). *Discriminative training and spanning tree algorithms for dependency parsing*. PhD thesis, University of Pennsylvania.

McMahon, J. G., & Smith, F. J. (1996). Improving statistical language models performance with automatically generated word hierarchies. *Computational Linguistics, 22*(2), 217–247.

Mel'čuk, I. A. (1988). *Dependency syntax: Theory and practice*. Albany: State University Press of New York.

Mel'čuk, I. A., Clas, A., & Polguère, A. (1995). *Introduction à la lexicologie explicative et combinatoire*. Louvain-la-Neuve: Éditions Duculot.

Merialdo, B. (1994). Tagging English text with a probabilistic model. *Computational Linguistics, 20*(2), 155–171.

Meyers, A., Reeves, R., Macleod, C., Szekely, R., Zielinska, V., Young, B., & Grishman, R. (2004). The NomBank project: An interim report. In A. Meyers (Ed.), *HLT-NAACL 2004 workshop: Frontiers in corpus annotation*, Boston (pp. 24–31).

Microsoft. (2004). *Microsoft office word 2003 rich text format (RTF) specification*. Microsoft. RTF Version 1.8.

Mikheev, A. (2002). Periods, capitalized words, etc. *Computational Linguistics, 28*(3), 289–318.

Miller, G. A. (1995). WordNet: A lexical database for English. *Communications of the ACM, 38*(11), 39–41.

Miller, G. A., Beckwith, R., Fellbaum, C., Gross, D., Miller, K. J., & Tangi, R. (1993). Five papers on WordNet. Technical report, Princeton University. ftp://ftp.cogsci.princeton.edu/pub/wordnet/5papers.ps. Cited October 28, 2005.

Moeschler, J. (1989). *Modélisation du dialogue: Représentation de l'inférence argumentative*. Paris: Hermès.

Moeschler, J., & Reboul, A. (1994). *Dictionnaire encyclopédique de pragmatique*. Paris: Éditions du Seuil.

Mohri, M., Pereira, F. C. N., & Riley, M. (1998). A rational design for a weighted finite-state transducer library. In D. Wood & S. Yu (Eds.), *Automata implementation. Second international workshop on implementing automata, WIA '97, London, September 1997. Revised papers* (Lecture notes in computer science, Vol. 1436, pp. 144–158). Berlin/Heidelberg/New York: Springer.

Mohri, M., Pereira, F. C. N., & Riley, M. (2000). The design principles of a weighted finite-state transducer library. *Theoretical Computer Science, 231*(1), 17–32.

Monachini, M., & Calzolari, N. (1996). Synopsis and comparison of morphosyntactic phenomena encoded in lexicons and corpora: A common proposal and applications to European languages. Technical report, Istituto di Linguistica Computazionale del CNR, Pisa. EAGLES Document EAG–CLWG–MORPHSYN/R.

Montague, R. M. (1974). *Formal philosophy: Selected papers*. New Haven: Yale University Press.

Montemagni, S., Barsotti, F., Battista, M., Calzolari, N., Corazzari, O., Lenci, A., Zampolli, A., Fanciulli, F., Massetani, M., Raffaelli, R., Basili, R., Pazienza, M. T., Saracino, D., Zanzotto, F., Mana, N., Pianesi, F., & Delmonte, R. (2003). Building the Italian syntactic-semantic treebank. In A. Abeillé (Ed.), *Treebanks: Building and using parsed corpora* (Language and speech series, Vol. 20, pp. 189–210). Dordrecht: Kluwer Academic.

Morton, A. (2003). *A guide through the theory of knowledge* (3rd ed.). Malden: Blackwell.

MUC-5. (Ed.). (1993). *Proceedings of the fifth message understanding conference (MUC-5)*, Baltimore. San Francisco: Morgan Kaufmann.

Müller, S. (1999). *Deutsche Syntax deklarativ. Head-Driven Phrase Structure Grammar für das Deutsche* (Linguistische Arbeiten, Vol. 394). Tübingen: Max Niemeyer Verlag.

Murphy, K. P. (2012). *Machine learning: A probabilistic perspective*. Cambridge, MA: MIT.

Nguyen, L., Abdou, S., Afify, M., Makhoul, J., Matsoukas, S., Schwartz, R., Xiang, B., Lamel, L., Gauvain, J.-L., Adda, G., Schwenk, H., & Lefevre, F. (2004). The 2004 BBN/LIMSI 10xRT English broadcast news transcription system. In *Proceedings DARPA RT04*, Palisades, New York.

Nilsson, J., Hall, J., & Nivre, J. (2005). MAMBA meets TIGER: Reconstructing a Swedish treebank from antiquity. In *Proceedings of the NODALIDA special session on treebanks*, Joensuu.

Nilsson, N. (1998). *Artificial intelligence: A new synthesis*. San Francisco: Morgan Kaufmann.

Nilsson, P., & Nugues, P. (2010). Automatic discovery of feature sets for dependency parsing. In *Proceedings of the 23rd international conference on computational linguistics (COLING 2010)*, Beijing (pp. 824–832). Coling 2010 Organizing Committee.

Nivre, J. (2003). An efficient algorithm for projective dependency parsing. In *Proceedings of the 8th international workshop on parsing technologies (IWPT 03)*, Nancy (pp. 149–160).

Nivre, J. (2006). *Inductive dependency parsing*. Dordrecht: Springer.

Nivre, J., Hall, J., Kübler, S., McDonald, R., Nilsson, J., Riedel, S., & Yuret, D. (2007). The CoNLL 2007 shared task on dependency parsing. In *Proceedings of the CoNLL shared task session of EMNLP-CoNLL 2007*, Prague (pp. 915–932).

Nivre, J., Hall, J., Nilsson, J., Eryigit, G., & Marinov, S. (2006). Labeled pseudo-projective dependency parsing with support vector machines. In *Proceedings of the tenth conference on computational natural language learning (CoNLL)*, New York (pp. 221–225).

Nivre, J., & Nilsson, J. (2005). Pseudo-projective dependency parsing. In *Proceedings of the 43rd annual meeting of the association for computational linguistics (ACL'05)*, Ann Arbor (pp. 99–106).

Och, F. J. (2012). Breaking down the language barrier – six years in. Cited June 4, 2012, from http://googletranslate.blogspot.com

Och, F. J., & Ney, H. (2003). A systematic comparison of various statistical alignment models. *Computational Linguistics, 29*(1), 19–51.

Orwell, G. (1949). *Nineteen eighty-four*. London: Secker and Warburg.

Palmer, H. E. (1933). *Second interim report on English collocations, submitted to the tenth annual conference of English teachers under the auspices of the institute for research in English teaching*. Tokyo: Institute for Research in English Teaching.

Palmer, M., Gildea, D., & Kingsbury, P. (2005). The Proposition Bank: An annotated corpus of semantic roles. *Computational Linguistics, 31*(1), 71–105.

Palomar, M., Civit, M., Díaz, A., Moreno, L., Bisbal, E., Aranzabe, M., Ageno, A., Martí, M. A., & Navarro, B. (2004). 3LB: Construcción de una base de datos de árboles sintáctico-semánticos para el catalán, euskera y español. In *XX Congreso de la Sociedad Española para el Procesamiento del Lenguaje Natural (SEPLN)*, Barcelona (pp. 81–88).

Papineni, K., Roukos, S., Ward, T., & Zhu, W.-J. (2002). Bleu: A method for automatic evaluation of machine translation. In *Proceedings of 40th annual meeting of the association for computational linguistics*, Philadelphia (pp. 311–318).

Paumier, S. (2006). *Unitex 1.2, Manuel d'utilisation*. Université de Marne-la-Vallée. English translation: Unitex 1.2, User manual.

Peirce, C. S. (1885). On the algebra of logic: A contribution to the philosophy of notation. *The American Journal of Mathematics, 7*(2), 180–202.

Peirce, C. S. (1897). The logic of relatives. *The Monist, 7*(2), 161–217.

Pereira, F. C. N. (1981). Extraposition grammars. *Computational Linguistics, 7*(4), 243–256.

Pereira, F. C. N. (1984). *C-Prolog user's manual, version 1.5*. University of Edinburgh.

Pereira, F. C. N., & Shieber, S. M. (1987). *Prolog and natural-language analysis* (CSLI lecture notes, Vol. 10). Stanford: Center for the Study of Language and Information.

Pereira, F. C. N., & Warren, D. H. D. (1980). Definite clause grammar for language analysis–a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence, 13*(3), 231–278.

Perelman, C., & Olbrechts-Tyteca, L. (1976). *Traité de l'argumentation: la nouvelle rhétorique*. Brussels: Éditions de l'Université de Bruxelles.

Pérennou, G., & de Calmès, M. (1987). BDLex lexical data and knowledge base of spoken and written French. In *European conference on speech technology*, Edinburgh (pp. 393–396).

Petrov, S., Das, D., & McDonald, R. (2012). A universal part-of-speech tagset. In *Proceedings of the eighth international conference on language resources and evaluation (LREC 2012)*, Istanbul (pp. 2089–2096).

Petruszewycz, M. (1973). L'histoire de la loi d'Estoup-Zipf: Documents. *Mathématiques et Sciences Humaines, 44*, 41–56.

Pollard, C., & Sag, I. A. (1994). *Head-driven phrase structure grammar*. Chicago: University of Chicago Press.

Pradhan, S., Moschitti, A., Xue, N., Uryupina, O., & Zhang, Y. (2012). CoNLL-2012 shared task: Modeling multilingual unrestricted coreference in OntoNotes. In *Proceedings of the joint conference on EMNLP and CoNLL: Shared task*, Jeju Island (pp. 1–40). Association for Computational Linguistics.

Pradhan, S., Ramshaw, L., Marcus, M., Palmer, M., Weischedel, R., & Xue, N. (2011). CoNLL-2011 shared task: Modeling unrestricted coreference in OntoNotes. In *Proceedings of the fifteenth conference on computational natural language learning: Shared task*, Portland (pp. 1–27). Association for Computational Linguistics.

Prasad, R., Dinesh, N., Lee, A., Miltsakaki, E., Robaldo, L., Joshi, A., & Webber, B. (2008). The Penn discourse treebank 2.0. In *Proceedings of the 6th international conference on language resources and evaluation*, Marrakech.

Procter, P. (Ed.). (1978). *Longman dictionary of contemporary English*. Harlow: Longman.

Procter, P. (Ed.). (1995). *Cambridge international dictionary of English*. Cambridge: Cambridge University Press.

Pustejovsky, J. (1995). *The generative lexicon*. Cambridge, MA: MIT.

Quillian, M. R. (1967). Word concepts: A theory and simulation of some basic semantic capabilities. *Behavioral Science, 12*(5), 410–430.

Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning, 1*(1), 81–106.

Quinlan, J. R. (1993). *C4.5: Programs for machine learning*. San Mateo: Morgan Kaufmann.

Rabiner, L. R. (1989). A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE, 77*(2), 257–286.

Radford, A. (1988). *Transformational grammar: A first course* (Cambridge textbooks in linguistics). Cambridge: Cambridge University Press.

Raghunathan, K., Lee, H., Rangarajan, S., Chambers, N., Surdeanu, M., Jurafsky, D., & Manning, C. (2010). A multi-pass sieve for coreference resolution. In *Proceedings of the 2010 conference on empirical methods in natural language processing*, Cambridge, MA (pp. 492–501). Association for Computational Linguistics.

Ramshaw, L. A., & Marcus, M. P. (1995). Text chunking using transformation-based learning. In D. Yarowsky & K. Church (Eds.), *Proceedings of the third workshop on very large corpora*, Cambridge, MA (pp. 82–94).

Ratnaparkhi, A. (1996). A maximum entropy model for part-of-speech tagging. In E. Brill & K. Church (Eds.), *Proceedings of the conference on empirical methods in natural language processing*, Philadelphia (pp. 133–142).

Ray, E. T. (2003). *Learning XML* (2nd ed.). Sebastopol: O'Reilly Media.

Rayner, M., Bretan, I., Wirén, M., Rydin, S., & Beshai, E. (1996). Composition of transfer rules in a multi-lingual MT system. In *Proceedings of the workshop on future issues for multilingual text processing*, Cairns.

Rayner, M., & Carter, D. (1995). The spoken language translator project. In *Proceedings of the language engineering convention*, London.

Rayner, M., Carter, D., Bouillon, P., Digalakis, V., & Wirén, M. (Eds.). (2000). *The spoken language translator*. Cambridge: Cambridge University Press.

Reboul, O. (1994). *Introduction à la rhétorique: théorie et pratique* (2nd ed.). Paris: Presses universitaires de France.

Reichenbach, H. (1947). *Elements of symbolic logic*. New York: Macmillan.

Rey, A. (Ed.). (1988). *Le Robert micro*. Paris: Dictionnaires Le Robert.

Reynar, J. C. (1998). *Topic segmentation: Algorithms and applications*. PhD thesis, University of Pennsylvania, Philadelphia.
Reynar, J. C., & Ratnaparkhi, A. (1997). A maximum entropy approach to identifying sentence boundaries. In *Proceedings of the fifth conference on applied natural language processing*, Washington, DC (pp. 16–19).
Rich, E., & Knight, K. (1991). *Artificial intelligence* (2nd ed.). New York: McGraw-Hill.
Ritchie, G. D., Russell, G. J., Black, A. W., & Pulman, S. G. (1992). *Computational morphology. Practical mechanisms for the English lexicon*. Cambridge, MA: MIT.
Robertson, D. W., Jr. (1946). A note on the classical origin of "circumstances" in the medieval confessional. *Studies in Philology, 43*(1), 6–14.
Robins, R. H. (1997). *A short history of linguistics* (4th ed.). London: Longman.
Robinson, J. A. (1965). A machine-oriented logic based on the resolution principle. *Journal of the ACM, 12*(1), 23–41.
Roche, E., & Schabes, Y. (1995). Deterministic part-of-speech tagging with finite-state transducers. *Computational Linguistics, 21*(2), 227–253.
Roche, E., & Schabes, Y. (Eds.). (1997). *Finite-state language processing*. Cambridge, MA: MIT.
Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review, 65*(6), 386–408.
Rouse, R. H., & Rouse, M. A. (1974). The verbal concordance to the scriptures. *Archivum Fratrum Praedicatorum, 44*, 5–30.
Ruppenhofer, J., Ellsworth, M., Petruck, M. R. L., Johnson, C. R., & Scheffczyk, J. (2010). FrameNet II: Extended theory and practice. Retrieved November 7, 2013, from https://fndrupal.icsi.berkeley.edu/fndrupal/the_book
Russell, S. J., & Norvig, P. (2010). *Artificial intelligence, a modern approach* (3rd ed.). Upper Saddle River: Pearson.
Ruwet, N. (1970). *Introduction à la grammaire générative* (2nd ed.). Paris: Plon.
Sabah, G. (1990). *L'intelligence artificielle et le langage* (2nd ed.). Paris: Hermès.
Sagerer, G. (1990). *Automatisches Verstehen gesprochener Sprache* (Reihe Informatik, Vol. 74). Mannheim: B.I. Wissenschaftsverlag.
Salton, G. (1988). *Automatic text processing: The transformation, analysis, and retrieval of information by computer*. Reading: Addison-Wesley.
Salton, G., & Buckley, C. (1987). Term weighting approaches in automatic text retrieval. Technical report TR87-881, Department of Computer Science, Cornell University, Ithaca.
Saporta, G. (2011). *Probabilités, analyse des données et statistiques* (3rd ed.). Paris: Éditions Technip.
Saussure, F. (1916). *Cours de linguistique générale*. Reprinted Payot, 1995, Paris.
Schiffrin, D. (1994). *Approaches to discourse* (Number 8 in Blackwell textbooks in linguistics). Oxford: Blackwell.
Schmid, H. (1994). Probabilistic part-of-speech tagging using decision trees. In *Proceedings of international conference on new methods in language processing*, Manchester.
Schmid, H. (1995). Improvements in part-of-speech tagging with an application to German. In *Proceedings of the ACL SIGDAT workshop*, Dublin.
Schölkopf, B., & Smola, A. J. (2002). *Learning with Kernels: Support vector machines, regularization, optimization, and beyond* (Adaptive computation and machine learning). Cambridge, MA: MIT.
Schwartz, R. L., Foy, B. D., & Phoenix, T. (2011). *Learning Perl* (6th ed.). Sebastopol: O'Reilly Media.
Searle, J. R. (1969). *Speech acts. An essay in the philosophy of language*. Cambridge: Cambridge University Press.
Searle, J. R. (1979). *Expression and meaning, studies in the theory of speech acts*. Cambridge: Cambridge University Press.
Searle, J. R., & Vanderveken, D. (1985). *Foundations of illocutionary logic*. Cambridge: Cambridge University Press.

Shannon, C. E. (1948). A mathematical theory of communication. *Bell System Technical Journal, 27*, 398–403; 623–656.

Simone, R. (2007). *Fondamenti di linguistica* (10th ed.). Bari: Laterza.

Sinclair, J. (Ed.). (1987). *Collins COBUILD English language dictionary*. London: Collins.

Singhal, A. (2012). Introducing the knowledge graph: Things, not strings. Official Google Blog. Retrieved November 7, 2013, from http://googleblog.blogspot.com/2012/05/introducing-knowledge-graph-things-not.html

Sleator, D., & Temperley, D. (1993). Parsing English with a link grammar. In *Proceedings of the third international workshop on parsing technologies*, Tilburg (pp. 277–291).

Soon, W. M., Ng, H. T., & Lim, D. C. Y. (2001). A machine learning approach to coreference resolution of noun phrases. *Computational Linguistics, 27*(4), 521–544.

Sorin, C., Jouvet, D., Gagnoulet, C., Dubois, D., Sadek, D., & Toularhoat, M. (1995). Operational and experimental French telecommunication services using CNET speech recognition and text-to-speech synthesis. *Speech Communication, 17*(3–4), 273–286.

Sproat, R. (1992). *Morphology and computation*. Cambridge, MA: MIT.

Sterling, L., & Shapiro, E. (1994). *The art of Prolog. Advanced programming techniques* (2nd ed.). Cambridge, MA: MIT.

Stolcke, A. (2002). SRILM – an extensible language modeling toolkit. In *Proceedings of international conference spoken language processing*, Denver.

Suchanek, F. M., Kasneci, G., & Weikum, G. (2007). Yago: A core of semantic knowledge. In *Proceedings of the 16th international conference on World Wide Web*, Banff (pp. 697–706). ACM.

Surdeanu, M., Johansson, R., Meyers, A., Màrquez, L., & Nivre, J. (2008). The CoNLL 2008 shared task on joint parsing of syntactic and semantic dependencies. In *CoNLL 2008: Proceedings of the 12th conference on computational natural language learning*, Manchester (pp. 159–177).

Suri, L. Z., & McCoy, K. F. (1994). RAFT/RAPR and centering: A comparison and discussion of problems related to processing complex sentences. *Computational Linguistics, 20*(2), 301–317.

Tapanainen, P., & Järvinen, T. (1997). A non-projective dependency parser. In *Proceedings of the fifth conference on applied natural language processing (ANLP'97)*, Washington, DC (pp. 64–71).

TAUM (1971). Taum 71. Rapport annuel du projet de traduction automatique de l'université de Montréal, Université de Montréal.

Taylor, K. (1998). *Truth and meaning. An introduction to the philosophy of language*. Oxford: Blackwell.

Ter Meulen, A. (1995). *Representing time in natural language. The dynamic interpretation of tense and aspect*. Cambridge, MA: MIT.

Tesnière, L. (1966). *Éléments de syntaxe structurale* (2nd ed.). Paris: Klincksieck.

The Unicode Consortium. (2012). *The unicode standard, version 6.1 – core specification*. Mountain View: Unicode Consortium.

Thompson, K. (1968). Regular expression search algorithm. *Communications of the ACM, 11*(6), 419–422.

Tjong Kim Sang, E. F. (2002). Introduction to the CoNLL-2002 shared task: Language-independent named entity recognition. In *Proceedings of CoNLL-2002*, Taipei (pp. 155–158).

Tjong Kim Sang, E. F., & Buchholz, S. (2000). Introduction to the CoNLL-2000 shared task: Chunking. In *Proceedings of CoNLL-2000 and LLL-2000*, Lisbon (pp. 127–132).

Tjong Kim Sang, E. F., & De Meulder, F. (2003). Introduction to the CoNLL-2003 shared task: Language-independent named entity recognition. In *Proceedings of CoNLL-2003*, Edmonton (pp. 142–147).

van Benthem, J., & Ter Meulen, A. (Eds.). (1997). *Handbook of logic and language*. Amsterdam: North-Holland.

Vanderveken, D. (1988). *Les actes de discours: Essai de philosophie du langage et de l'esprit sur la signification des énonciations*. Bruxelles: Mardaga.

van Noord, G., & Gerdemann, D. (2001). An extendible regular expression compiler for finite-state approaches in natural language processing. In O. Boldt & H. Jürgensen (Eds.), *Automata implementation. 4th international workshop on implementing automata, WIA'99, Potsdam, July 1999, revised papers* (Lecture notes in computer science, Vol. 2214, pp. 122–139). Berlin/Heidelberg/New York: Springer.

Vendler, Z. (1967). *Linguistics in philosophy*. Ithaca: Cornell University Press.

Vergne, J. (1998). Entre arbre de dépendance et ordre linéaire, les deux processus de transformation: Linéarisation, puis reconstruction de l'arbre. *Cahiers de grammaire*, 23.

Vergne, J. (1999). *Étude et modélisation de la syntaxe des langues à l'aide de l'ordinateur. Analyse syntaxique automatique non combinatoire. Synthèse et résultats*. Habilitation à diriger des recherches, Université de Caen.

Verhulst, P.-F. (1838). Notice sur la loi que la population poursuit dans son accroissement. *Correspondance mathématique et physique, 10*, 113–121.

Verhulst, P.-F. (1845). Recherches mathématiques sur la loi d'accroissement de la population. *Nouveaux Mémoires de l'Académie Royale des Sciences et Belles-Lettres de Bruxelles, 18*, 1–42.

Vilain, M., Burger, J., Aberdeen, J., Connolly, D., & Hirschman, L. (1995). A model-theoretic coreference scoring scheme. In *Proceedings of the conference on sixth message understanding conference (MUC-6)*, Columbia (pp. 45–52).

Viterbi, A. J. (1967). Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory, 13*(2), 260–267.

Voutilainen, A., Heikkilä, J., & Anttila, A. (1992). Constraint grammar of English: A performance-oriented introduction. Technical report 21, Department of General Linguistics, University of Helsinki.

Voutilainen, A., & Järvinen, T. (1995). Specifying a shallow grammatical representation for parsing purposes. In *Proceedings of the seventh conference of the European chapter of the ACL*, Dublin (pp. 210–214).

Wahrig, G. (Ed.). (1978). *dtv-Wörterbuch der deutschen Sprache*. Munich: Deutscher Taschenbuch Verlag.

Wall, L., Christiansen, T., & Orwant, J. (2000). *Programming Perl* (3rd ed.). Sebastopol: O'Reilly Media.

Wang, K., Thrasher, C., Viegas, E., Li, X., & Hsu, B. (2010). An overview of Microsoft web n-gram corpus and applications. In *Proceedings of the NAACL HLT 2010: Demonstration session*, Los Angeles (pp. 45–48).

Warren, D. H. D. (1983). An abstract Prolog instruction set. Technical note 309, SRI International, Menlo Park.

Weizenbaum, J. (1966). ELIZA – A computer program for the study of natural language communication between man and machine. *Communications of the ACM, 9*(1), 36–45.

Wilks, Y. A., Slator, B. M., & Guthrie, L. M. (1996). *Electric words. Dictionaries, computers, and meanings*. Cambridge, MA: MIT.

Wilks, Y., & Stevenson, M. (1997). Sense tagging: Semantic tagging with a lexicon. In *Tagging text with lexical semantics: Why, what, and how? Proceedings of the workshop*, Washington, DC (pp. 74–78). ACL SIGLEX.

Witten, I. H., & Frank, E. (2005). *Data mining: Practical machine learning tools and techniques* (2nd ed.). San Francisco: Morgan Kaufmann.

Yarowsky, D. (1995). Unsupervised word sense disambiguation rivaling supervised methods. In *Proceedings of the 33rd annual meeting of the association for computational linguistics*, Cambridge, MA (pp. 189–196).

Yarowsky, D. (1996). Homograph disambiguation in speech synthesis. In J. van Santen, R. Sproat, J. P. Olive, & J. Hirschberg (Eds.), *Progress in speech synthesis* (pp. 159–175). Berlin/Heidelberg/New York: Springer.

Yu, H.-F., Ho, C.-H., Juan, Y.-C., & Lin, C.-J. (2013). LibShortText: A library for short-text classification and analysis. Retrieved November 1, 2013, from http://www.csie.ntu.edu.tw/~cjlin/libshorttext

Zampolli, A. (2003). Past and on-going trends in computational linguistics. *The ELRA Newsletter, 8*(3–4), 6–16.

Zaragoza, H., Craswell, N., Taylor, M., Saria, S., & Robertson, S. (2004). Microsoft Cambridge at TREC-13: Web and HARD tracks. In *Proceedings of TREC-2004*, Gaithersburg.

Zelle, J. M., & Mooney, R. J. (1997). An inductive logic programming method for corpus-based parser construction. Technical note, University of Texas at Austin.

Zhang, T., & Johnson, D. (2003). A robust risk minimization based named entity recognition system. In *Proceedings of CoNLL-2003*, Edmonton (pp. 204–207).

Zhang, Y., & Nivre, J. (2011). Transition-based dependency parsing with rich non-local features. In *Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies*, Portland (pp. 188–193).

Index

P.M. Nugues, *Language Processing with Perl and Prolog*, Cognitive Technologies,
DOI 10.1007/978-3-642-41464-0, © Springer-Verlag Berlin Heidelberg 2014