Introduction to Natural Language Processing

Authors: Steven Bird, Ewan Klein, Edward Loper
Version: 0.7.5 (draft only, please send feedback to authors)

Revision:
Date:

Contents

Preface

Most human knowledge — and most human communication — is represented and expressed using language. Language technologies permit computers to process human language automatically; handheld computers support predictive text and handwriting recognition; web search engines give access to information locked up in unstructured text. By providing more natural human-machine interfaces, and more sophisticated access to stored information, language processing has come to play a central role in the multilingual information society.

This textbook provides a comprehensive introduction to the field of natural language processing (NLP), covering the major techniques and theories. The book provides numerous worked examples and exercises, and can serve as the main text for undergraduate and introductory graduate courses on natural language processing or computational linguistics.

Audience

This book is intended for people in the language sciences and the information sciences who want to learn how to write programs that analyze written language. You won't need any prior knowledge of linguistics or computer science; those with a background in either area can simply skip over some of the discussions. Depending on which background you come from, and your motivation for being interested in NLP, you will gain different kinds of skills and knowledge from this book, as set out below:

Goals	Background	
	Linguistics	Computer Science
Linguistic Analysis	Programming to manage linguistic data, explore formal theories, and test empirical claims	Linguistics as a source of interesting problems in data modelling, data mining, and formal language theory
Language Technology	Learning to program with applications to familiar problems, to work in language technology or other technical field	Knowledge of linguistics as fundamental to developing high quality, maintainable language processing software

Table 1:

The structure of the book is biased towards linguists, in that the introduction to programming appears in the main chapter sequence, and early chapters contain many elementary examples. We hope that computer science readers can quickly skip over such materials till they reach content that is more linguistically challenging.

What You Will Learn

By the time you have dug into the material presented here, you will have acquired substantial skills and knowledge in the following areas:

- how simple programs can help linguists manipulate and analyze language data, and how to write these programs;
- key concepts from linguistic description and analysis;
- how linguistic knowledge is used in important language technology components;
- knowledge of the principal data structures and algorithms used in NLP, and skills in algorithmic problem solving, data modelling, and data management;
- understanding of the standard corpora and their use in formal evaluation;
- the organization of the field of NLP;
- skills in Python programming for NLP.

Download the Toolkit...

This textbook is a companion to the *Natural Language Toolkit*. All software, corpora, and documentation are freely downloadable from http://nltk.sourceforge.net/. Distributions are provided for Windows, Macintosh and Unix platforms. All NLTK distributions plus Python and WordNet distributions are also available in the form of an ISO image which can be downloaded and burnt to CD-ROM for easy local redistribution. We strongly encourage you to download the toolkit before you go beyond the first chapter of the book.

Emphasis

This book is a **practical** introduction to NLP. You will learn by example, write real programs, and grasp the value of being able to test an idea through implementation. If you haven't learnt already, this book will teach you **programming**. Unlike other programming books, we provide extensive illustrations and exercises from NLP. The approach we have taken is also **principled**, in that we cover the theoretical underpinnings and don't shy away from careful linguistic and computational analysis. We have tried to be **pragmatic** in striking a balance between theory and application, and alternate between the two several times each chapter, identifying the connections but also the tensions. Finally, we recognize that you won't get through this unless it is also **pleasurable**, so we have tried to include many applications and examples that are interesting and entertaining, sometimes whimsical.

Structure

The book is structured into three parts, as follows:

Part 1: Basics In this part, we focus on recognising simple structure in text. We start with individual words, then explore parts of speech and simple syntactic constituents.

Part 2: Parsing Here, we deal with syntactic structure, trees, grammars, and parsing.

Part 3: Advanced Topics This final part of the book contains chapters which address selected topics in NLP in more depth and to a more advanced level. By design, the chapters in this part can be read independently of each other.

The three parts have a common structure: they start off with a chapter on programming, followed by three chapters on various topics in NLP. The programming chapters are *foundational*, and you must master this material before progressing further.

Each chapter consists of an introduction, a sequence of sections that progress from elementary to advanced material, and finally a summary and suggestions for further reading. Most sections include exercises which are graded according to the following scheme: ☼ is for easy exercises that involve minor modifications to supplied code samples or other simple activities; ◑ is for intermediate exercises that explore an aspect of the material in more depth, requiring careful analysis and design; ★ is for difficult, open-ended tasks that will challenge your understanding of the material and force you to think independently (readers new to programming are encouraged to skip these). The exercises are important for consolidating the material in each section, and we strongly encourage you to try a few before continuing with the rest of the chapter.

For Instructors

Natural Language Processing (NLP) is often taught within the confines of a single-semester course at advanced undergraduate level or postgraduate level. Many instructors have found that it is difficult to cover both the theoretical and practical sides of the subject in such a short span of time. Some courses focus on theory to the exclusion of practical exercises, and deprive students of the challenge and excitement of writing programs to automatically process language. Other courses are simply designed to teach programming for linguists, and do not manage to cover any significant NLP content. The *Natural Language Toolkit* (NLTK) was developed to address this problem, making it feasible to cover a substantial amount of theory and practice within a single-semester course, even if students have no prior programming experience.

A significant fraction of any NLP syllabus covers fundamental data structures and algorithms. These are usually taught with the help of formal notations and complex diagrams. Large trees and charts are copied onto the board and edited in tedious slow motion, or laboriously prepared for presentation slides. It is more effective to use live demonstrations in which those diagrams are generated and updated automatically. NLTK provides interactive graphical user interfaces, making it possible to view program state and to study program execution step-by-step. Most NLTK components have a demonstration mode, and will perform an interesting task without requiring any special input from the user. It is even possible to make minor modifications to programs in response to "what if" questions. In this way, students learn the mechanics of NLP quickly, gain deeper insights into the data structures and algorithms, and acquire new problem-solving skills.

NLTK supports assignments of varying difficulty and scope. In the simplest assignments, students experiment with existing components to perform a wide variety of NLP tasks. This may involve no programming at all, in the case of the existing demonstrations, or simply changing a line or two of program code. As students become more familiar with the toolkit they can be asked to modify existing components or to create complete systems out of existing components. NLTK also provides students with a flexible framework for advanced projects, such as developing a multi-component system, by integrating and extending NLTK components, and adding on entirely new components. Here NLTK

helps by providing standard implementations of all the basic data structures and algorithms, interfaces to standard corpora, substantial corpus samples, and a flexible and extensible architecture. Thus, as we have seen, NLTK offers a fresh approach to NLP pedagogy, in which theoretical content is tightly integrated with application.

We believe our book is unique in providing a comprehensive pedagogical framework for students to learn about NLP in the context of learning to program. What sets our materials apart is the tight coupling of the chapters and exercises with NLTK, giving students — even those with no prior programming experience — a practical introduction to NLP. Once completing these materials, students will be ready to attempt one of the more advanced textbooks, such as *Foundations of Statistical Natural Language Processing*, by Manning and Schütze (MIT Press, 2000).

Course Plans; Lectures/Lab Sessions per Chapter		
Chapter	**Linguists**	**Computer Scientists**
1 Introduction	1	1
2 Programming	4	1
3 Words	2	2
4 Tagging	2-3	2
5 Chunking	0-2	2
6 Structured Programming	2-4	1
7 Grammars and Parsing	2-4	2-4
8 Advanced Parsing	1-4	3
9 Feature Based Grammar	2-4	2-4
10-14 Advanced Topics	2-8	2-16
Total	18-36	18-36

Table 2:

Further Reading:

The Association for Computational Linguistics (ACL) The ACL is the foremost professional body in NLP. Its journal and conference proceedings, approximately 10,000 articles, are available online with a full-text search interface, via http://www.aclweb.org/anthology/.

Linguistic Terminology A comprehensive glossary of linguistic terminology is available at:

http://www.sil.org/linguistics/GlossaryOfLinguisticTerms/.

Language Files *Materials for an Introduction to Language and Linguistics (Ninth Edition)*, The Ohio State University Department of Linguistics. For more information, see http://www.ling.ohio-state.edu/publications/files/.

Acknowledgements

NLTK was originally created as part of a computational linguistics course in the Department of Computer and Information Science at the University of Pennsylvania in 2001. Since then it has been developed and expanded with the help of dozens of contributors. It has now been adopted in courses in dozens of universities, and serves as the basis of many research projects.

In particular, we're grateful to the following people for their feedback, comments on earlier drafts, advice, contributions: Greg Aumann, Ondrej Bojar, Trevor Cohn, James Curran, Jean Mark Gawron, Baden Hughes, Christopher Maloof, Stuart Robinson, Rob Speer. Many others have contributed to the toolkit, and they are listed at http://nltk.sourceforge.net/contrib.html.

We also acknowledge the following sources: Carpenter and Chu-Carroll's ACL-99 Tutorial on Spoken Dialogue Systems (example dialogue in 1).

About the Authors

		
Steven Bird	Ewan Klein	Edward Loper

Table 3:

Steven Bird is an Associate Professor in the Department of Computer Science and Software Engineering at the University of Melbourne, and a Senior Research Associate in the Linguistic Data Consortium at the University of Pennsylvania. After completing a PhD at the University of Edinburgh on computational phonology (1990), Steven moved to Cameroon to conduct fieldwork on tone and orthography. Later he spent four years as Associate Director of the Linguistic Data Consortium where he developed models and tools for linguistic annotation. His current research interests are in linguistic databases and query languages.

Ewan Klein is Professor of Language Technology in the School of Informatics at the University of Edinburgh. He completed a PhD on formal semantics at the University of Cambridge in 1978. After some years working at the Universities of Sussex and Newcastle upon Tyne, he took up a teaching position at Edinburgh. His current research interests are in computational semantics.

Edward Loper is a doctoral student in the Department of Computer and Information Sciences at the University of Pennsylvania, conducting research on machine learning in NLP. In addition to NLTK, he has helped develop other major packages for documenting and testing Python software, epydoc and doctest.

Python and the Natural Language Toolkit

Why Python?

Python is a simple yet powerful programming language with excellent functionality for processing linguistic data. Python can be downloaded for free from `http://www.python.org/`.

Here is a five-line Python program which takes text input and prints all the words ending in `ing`:

```
>>> import sys                          # load the system library
>>> for line in sys.stdin:              # for each line of input
...     for word in line.split():       # for each word in the line
...         if word.endswith('ing'):    # does the word end in 'ing'?
...             print word              # if so, print the word
```

This program illustrates some of the main features of Python. First, whitespace is used to *nest* lines of code, thus the line starting with `if` falls inside the scope of the previous line starting with `for`, so the `ing` test is performed for each word. Second, Python is *object-oriented*; each variable is an entity which has certain defined attributes and methods. For example, `line` is more than a sequence of characters. It is a string object that has a **method** (or operation) called `split` that we can use to break a line into its words. To apply a method to an object, we give the object name, followed by a period, followed by the method name. Third, methods have *arguments* expressed inside parentheses. For instance, `split` had no argument because we were splitting the string wherever there was white space. To split a string into sentences delimited by a period, we could write `split('.')`. Finally, and most importantly, Python is highly readable, so much so that it is fairly easy to guess what the above program does even if you have never written a program before.

We chose Python as the implementation language for NLTK because it has a shallow learning curve, its syntax and semantics are transparent, and it has good string-handling functionality. As a scripting language, Python facilitates interactive exploration. As an object-oriented language, Python permits data and methods to be encapsulated and re-used easily. As a dynamic language, Python permits attributes to be added to objects on the fly, and permits variables to be typed dynamically, facilitating rapid development. Python comes with an extensive standard library, including components for graphical programming, numerical processing, and web data processing.

Python is heavily used in industry, scientific research, and education around the world. Python is often praised for the way it facilitates productivity, quality, and maintainability of software. A collection of Python success stories is posted at `http://www.python.org/about/success/`.

NLTK defines a basic infrastructure that can be used to build NLP programs in Python. It provides: basic classes for representing data relevant to natural language processing; standard interfaces for performing tasks such as tokenization, tagging, and parsing; standard implementations for each task which can be combined to solve complex problems; and extensive documentation including tutorials and reference documentation.

Teaching and Learning Python and NLTK

This book contains self-paced learning materials including many examples and exercises. An effective way for students to learn is simply to work through the materials, with the help of other students and instructors. The program fragments can be cut and pasted directly from the online tutorials. The HTML version has a blue bar beside each program fragment; click on the bar to automatically copy the program fragment to the clipboard (assumes appropriate browser security settings.) Note that the examples in each chapter often use functions and modules that were imported in earlier examples. When lifting examples from these materials it is often necessary to add one or more `import` statements.

This material can also be used as the basis for lecture-style presentations (and some slides are available for download from the NLTK website). We believe that an effective way to present the materials is through interactive presentation of the examples, entering them at the Python prompt, observing what they do, and modifying them to explore some empirical or theoretical question.

Integrated Development Environments: The easiest way to develop Python code, and to perform interactive Python demonstrations, is to use the simple editor and interpreter GUI that comes with Python called *IDLE*, the *Integrated DeveLopment Environment for Python*. A more sophisticated approach is to use a full-blown IDE such as *Eclipse* together with a Python plugin such as *PyDEV* (see the NLTK wiki for instructions).

NLTK Community: NLTK has a large and growing user base. There are mailing lists for annoucements about NLTK, for developers and for teachers. A wiki hosted on the NLTK website is available where registered users can share 'recipes', convenient short scripts for performing useful tasks. The wiki also lists courses around the world where NLTK has been adopted, serving as a useful source of associated resources.

The Design of NLTK

NLTK was designed with six requirements in mind:

Simplicity: We have tried to provide an intuitive and appealing framework along with substantial building blocks, for students to gain a practical knowledge of NLP without getting bogged down in the tedious house-keeping usually associated with processing annotated language data. We have provided software distributions for several platforms, along with platform-specific instructions, to make the toolkit easy to install.

Consistency: We have made a significant effort to ensure that all the data structures and interfaces are consistent, making it easy to carry out a variety of tasks using a uniform framework.

Extensibility: The toolkit easily accommodates new components, whether those components replicate or extend existing functionality. Moreover, the toolkit is organized so that it is usually obvious where extensions would fit into the toolkit's infrastructure.

Modularity: The interaction between different components of the toolkit uses simple, well-defined interfaces. It is possible to complete individual projects using small parts of the toolkit, without needing to understand how they interact with the rest of the toolkit. This allows students to learn how to use the toolkit incrementally throughout a course. Modularity also makes it easier to change and extend the toolkit.

Well-Documented: The toolkit comes with substantial documentation, including nomenclature, data structures, and implementations.

Contrasting with these requirements are three non-requirements — potentially useful features that we have deliberately avoided. First, while the toolkit provides a wide range of functions, it is not intended to be encyclopedic. There should be a wide variety of ways in which students can extend the toolkit. Second, while the toolkit should be efficient enough that students can use their NLP systems to perform meaningful tasks, it does not need to be highly optimized for runtime performance. Such optimizations often involve more complex algorithms, and sometimes require the use of C or C++. This would make the toolkit less accessible and more difficult to install. Third, we have avoided clever programming tricks, since clear implementations are far preferable to ingenious yet indecipherable ones.

NLTK Organization: NLTK is organized into a collection of task-specific packages. Each package is a combination of data structures for representing a particular kind of information such as trees, and implementations of standard algorithms involving those structures such as parsers. This approach is a standard feature of *object-oriented design*, in which components encapsulate both the resources and methods needed to accomplish a particular task.

The most fundamental NLTK components are for identifying and manipulating individual words of text. These include: `tokenize`, for breaking up strings of characters into word tokens; `tag`, for adding part-of-speech tags, including regular-expression taggers, n-gram taggers and Brill taggers; and the Porter stemmer.

The second kind of module is for creating and manipulating structured linguistic information. These components include: `tree`, for representing and processing parse trees; `featurestructure`, for building and unifying nested feature structures (or attribute-value matrices); `cfg`, for specifying context-free grammars; and `parse`, for creating parse trees over input text, including chart parsers, chunk parsers and probabilistic parsers.

Several utility components are provided to facilitate processing and visualization. These include: `draw`, to visualize NLP structures and processes; `probability`, to count and collate events, and perform statistical estimation; and `corpora`, to access tagged linguistic corpora.

A further group of components is not part of NLTK proper. These are a wide selection of third-party contributions, often developed as student projects at various institutions where NLTK is used, and distributed in a separate package called *NLTK Contrib*. Several of these student contributions, such as the Brill tagger and the HMM module, have now been incorporated into NLTK. Although these contributed components are not maintained, they may serve as a useful starting point for future student projects.

In addition to software and documentation, NLTK provides substantial corpus samples. Many of these can be accessed using the `corpora` module, avoiding the need to write specialized file parsing code before you can do NLP tasks. These corpora include: Brown Corpus — 1.15 million words of tagged text in 15 genres; a 10% sample of the Penn Treebank corpus, consisting of 40,000 words of syntactically parsed text; a selection of books from Project Gutenberg totally 1.7 million words; and other corpora for chunking, prepositional phrase attachment, word-sense disambiguation, information extraction

Note on NLTK-Lite: Since mid-2005, the NLTK developers have been creating a lightweight version NLTK, called NLTK-Lite. NLTK-Lite is simpler and faster than NLTK. Once it is complete, NLTK-Lite will provide the same functionality as NLTK. However, unlike NLTK, NLTK-Lite does not impose such a heavy burden on the programmer. Wherever possible, standard Python objects are used instead of custom NLP versions, so that students learning to program for the first time will be learning to program in Python with some useful libraries, rather than learning to program in NLTK.

NLTK Papers: NLTK has been presented at several international conferences with published proceedings, as listed below:

Edward Loper and Steven Bird (2002). NLTK: The Natural Language Toolkit, *Proceedings of the ACL Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics*, Somerset, NJ: Association for Computational Linguistics, pp. 62-69, http://arXiv.org/abs/cs/0205028

Steven Bird and Edward Loper (2004). NLTK: The Natural Language Toolkit, *Proceedings of the ACL demonstration session*, pp 214-217. http://eprints.unimelb.edu.au/archive/00001448/

Steven Bird (2005). NLTK-Lite: Efficient Scripting for Natural Language Processing, *4th International Conference on Natural Language Processing*, pp 1-8. http://eprints.unimelb.edu.au/archive/00001453/

Steven Bird (2006). NLTK: The Natural Language Toolkit, *Proceedings of the ACL demonstration session* http://www.ldc.upenn.edu/sb/home/papers/nltk-demo-06.pdf

Edward Loper (2004). NLTK: Building a Pedagogical Toolkit in Python, *PyCon DC 2004* Python Software Foundation, http://www.python.org/pycon/dc2004/papers/

Ewan Klein (2006). Computational Semantics in the Natural Language Toolkit, *Australian Language Technology Workshop*. http://www.alta.asn.au/events/altw2006/proceedings/Klein.pdf

Chapter 1

Introduction to Natural Language Processing

1.1 Why Language Processing is Useful

How do we write programs to manipulate natural language? What questions about language could we answer? How would the programs work, and what data would they need? These are just some of the topics we will cover in this book. Before we tackle the subject systematically, we will take a quick look at some simple tasks in which computational tools manipulate language data in a variety of interesting and non-trivial ways.

Our first example involves word *stress*. The CMU Pronunciation Dictionary is a machine-readable dictionary that gives the pronunciation of over 125,000 words in North American English. Each entry consists of a word in standard orthography followed by a phonological transcription. For example, the entry for *language* is the following:

(1) language / L AE1 NG G W AH0 JH .

Each character or group of characters following the slash represents an English *phoneme*, and the final numbers indicate word stress. That is, `AE1` is the nucleus of a stressed *syllable*, while `AH0` is the nucleus of an unstressed one. Let's suppose that we want to find every word in the dictionary which exhibits a particular stress pattern; say, words whose primary stress is on their fifth-last syllable (this is called **pre-preantepenultimate** stress). Searching through the dictionary by hand would be tedious, and we would probably miss some of the cases. We can write a simple program that will extract the numerals 0, 1 and 2 from transcription and create a new field `stress_pattern` for each word which is just a sequence of these stress numbers. After this has been done, it is easy to scan the extracted stress patterns for any sequence which ends with `10000`. Here are some of the words that we can find using this method:

(2)
```
ACCUMULATIVELY / AH0 K Y UW1 M Y AH0 L AH0 T IH0 V L IY0
AGONIZINGLY / AE1 G AH0 N AY0 Z IH0 NG L IY0
CARICATURIST / K EH1 R AH0 K AH0 CH ER0 AH0 S T
CUMULATIVELY / K Y UW1 M Y AH0 L AH0 T IH0 V L IY0
IMAGINATIVELY / IH2 M AE1 JH AH0 N AH0 T IH0 V L IY0
INSTITUTIONALIZES / IH2 N S T AH0 T UW1 SH AH0 N AH0 L AY0 Z AH0 Z
SPIRITUALIST / S P IH1 R IH0 CH AH0 W AH0 L AH0 S T
UNALIENABLE / AH0 N EY1 L IY0 EH0 N AH0 B AH0 L
```

Our second example also involves *phonology*. When we construct an inventory of sounds for a language, we are usually interested in just those sounds which can make a difference in word meaning. To do this, we look for **minimal pairs**; that is, distinct words which differ in only one sound. For example, we might argue that the sounds [p] and [b] in English are distinctive because if we replace one with the other, we often end up with a different word:

(3)
```
pat vs. bat
nip vs. nib
```

Suppose we want to do this more systematically for a language where we have a list of words, but are still trying to determine the sound inventory. As a case in point, NLTK includes a lexicon for *Rotokas*, an East Papuan language spoken on Bougainville Island, near Papua New Guinea. Let's suppose we are interested in how many vowels there are in Rotokas. We can write a program to find all four-letter words which differ only by their first vowel, and tabulate the results to illustrate vowel contrasts:

(4)
```
kasi -    kesi kusi kosi
kava -    -    kuva kova
karu kiru keru kuru koru
kapu kipu -    -    kopu
karo kiro -    -    koro
kari kiri keri kuri kori
kapa -    kepa -    kopa
kara kira kera -    kora
kaku -    -    kuku koku
kaki kiki -    -    koki
```

The two preceding examples have used lexical resources. We can also write programs to analyze texts in various ways. In this example, we try to build a model of the patterns of adjacent words in the book of *Genesis*. Each pair of adjacent words is known as a **bigram**, and we can build a very simple model of biblical language just by counting bigrams. There are many useful things we can do with such information, such as identifying genres of literature or even identifying the author of a piece of text. Here we use it for a more whimsical purpose: to generate random text in the style of Genesis. As you will see, we have managed to capture something about the flow of text from one word to the next, but beyond this it is simply nonsense:

(5)
```
lo, it came to the land of his father and he said, i will not be a
wife unto him, saying, if thou shalt take our money in their kind,
cattle, in thy seed after these are my son from off any more than all
that is this day with him into egypt, he, hath taken away unawares to
pass, when she bare jacob said one night, because they were born two
hundred years old, as for an altar there, he had made me out at her
pitcher upon every living creature after thee shall come near her:
yea,
```

For our last example, let's suppose we are engaged in research to study semantic contrast in English verbs. We hypothesize that one useful source of data for exploring such contrasts might be a list of verb

phrases which are conjoined with the word *but*. So we need to carry out some grammatical analysis to find conjoined verb phrases, and also need to be able to specify *but* as the conjunction. Rather than trying to do the grammatical analysis ourselves, we can make use of a resource in which the syntactic trees have already been added to lots of sentences. The best known of such resources is the University of Pennsylvania Treebank corpus (or **Penn Treebank** for short), and we can write a program to read trees from this corpus, find instances of verb phrase conjunctions involving the word *but*, and display parsed text corresponding to the two verb phrases.

(6)
```
(VBZ has) (VP opened its market to foreign cigarettes)
  BUT (VBZ restricts) (NP advertising) (PP-CLR to designated places)
(VBZ admits) (SBAR 0 she made a big mistake)
  BUT (VBD did) (RB n't) (VP elaborate)
(VBD confirmed) (SBAR 0 he had consented to the sanctions)
  BUT (VBD declined) (S *-1 to comment further)
(VBP are) (NP-PRD a guide to general levels)
  BUT (VBP do) (RB n't) (ADVP-TMP always) (VP represent actual transactions)
(VBN flirted) (PP with a conversion to tabloid format) (PP-TMP for years)
  BUT (ADVP-TMP never) (VBN executed) (NP the plan)
(VBD ended) (ADVP-CLR slightly higher)
  BUT (VBD trailed) (NP gains in the Treasury market)
(VBD confirmed) (NP the filing)
  BUT (MD would) (RB n't) (VP elaborate)
```

In presenting these examples, we have tried to give you a flavour of the range of things that can be done with natural language using computational tools. All the above examples were generated using simple programming techniques and a few lines of Python code. After working through the first few chapters of this book, you will be able write such programs yourself. In the process, you will come to understand the basics of **natural language processing**, henceforth abbreviated as NLP. In the remainder of this chapter, we will give you more reasons to think that NLP is both important and fun.

1.2 The Language Challenge

1.2.1 Language is rich and complex

Language is the chief manifestation of human intelligence. Through language we express basic needs and lofty aspirations, technical know-how and flights of fantasy. Ideas are shared over great separations of distance and time. The following samples from English illustrate the richness of language:

(7a) Overhead the day drives level and grey, hiding the sun by a flight of grey spears. (William Faulkner, *As I Lay Dying*, 1935)

(7b) When using the toaster please ensure that the exhaust fan is turned on. (sign in dormitory kitchen)

(7c) Amiodarone weakly inhibited CYP2C9, CYP2D6, and CYP3A4-mediated activities with Ki values of 45.1-271.6 μM (Medline, PMID: 10718780)

(7d) Iraqi Head Seeks Arms (spoof news headline)

(7e) The earnest prayer of a righteous man has great power and wonderful results. (James 5:16b)

(7f) Twas brillig, and the slithy toves did gyre and gimble in the wabe (Lewis Carroll, *Jabberwocky*, 1872)

(7g) There are two ways to do this, AFAIK :smile: (internet discussion archive)

Thanks to this richness, the study of language is part of many disciplines outside of linguistics, including translation, literary criticism, philosophy, anthropology and psychology. Many less obvious disciplines investigate language use, such as law, hermeneutics, forensics, telephony, pedagogy, archaeology, cryptanalysis and speech pathology. Each applies distinct methodologies to gather observations, develop theories and test hypotheses. Yet all serve to deepen our understanding of language and of the intellect which is manifested in language.

The importance of language to science and the arts is matched in significance by the cultural treasure embodied in language. Each of the world's ~7,000 human languages is rich in unique respects, in its oral histories and creation legends, down to its grammatical constructions and its very words and their nuances of meaning. Threatened remnant cultures have words to distinguish plant subspecies according to therapeutic uses which are unknown to science. Languages evolve over time as they come into contact with each other and they provide a unique window onto human pre-history. Technological change gives rise to new words like *blog* and new morphemes like *e-* and *cyber-*. In many parts of the world, small linguistic variations from one town to the next add up to a completely different language in the space of a half-hour drive. For its breathtaking complexity and diversity, human language is as a colourful tapestry stretching through time and space.

1.2.2 Language and the Internet

Today, people from all walks of life — including professionals, students, and the general population — are confronted by unprecedented volumes of information, the vast bulk of which is stored as unstructured text. In 2003, it was estimated that the annual production of books amounted to 8 Terabytes. (A Terabyte is 1,000 Gigabytes, i.e., equivalent to 1,000 pickup trucks filled with books.) It would take a human being about five years to read the new scientific material that is produced every 24 hours. Although these estimates are based on printed materials, increasingly the information is also available electronically. Indeed, there has been an explosion of text and multimedia content on the World Wide Web. For many people, a large and growing fraction of work and leisure time is spent navigating and accessing this universe of information.

The presence of so much text in electronic form is a huge challenge to NLP. Arguably, the only way for humans to cope with the information explosion is to exploit computational techniques which can sift through huge bodies of text.

Although existing search engines have been crucial to the growth and popularity of the Web, humans require skill, knowledge, and some luck, to extract answers to such questions as *What tourist sites can I visit between Philadelphia and Pittsburgh on a limited budget? What do expert critics say about digital SLR cameras? What predictions about the steel market were made by credible commentators in the past week?* Getting a computer to answer them automatically is a realistic long-term goal, but would involve a range of language processing tasks, including information extraction, inference, and summarization, and would need to be carried out on a scale and with a level of robustness that is still beyond our current capabilities.

1.2.3 The Promise of NLP

As we have seen, NLP is important for scientific, economic, social, and cultural reasons. NLP is experiencing rapid growth as its theories and methods are deployed in a variety of new language technologies. For this reason it is important for a wide range of people to have a working knowledge of NLP. Within academia, this includes people in areas from *humanities computing* and *corpus linguistics* through to *computer science* and *artificial intelligence*. Within industry, it includes people in *human-computer interaction*, *business information analysis*, and *Web software development*. We hope that you, a member of this diverse audience reading these materials, will come to appreciate the workings of this rapidly growing field of NLP and will apply its techniques in the solution of real-world problems.

The following chapters present a carefully-balanced selection of theoretical foundations and practical applications, and equips readers to work with large datasets, to create robust models of linguistic phenomena, and to deploy them in working language technologies. By integrating all of this into the Natural Language Toolkit (NLTK), we hope this book opens up the exciting endeavour of practical natural language processing to a broader audience than ever before.

1.3 Language and Computation

1.3.1 NLP and Intelligence

A long-standing challenge within computer science has been to build intelligent machines. The chief measure of *machine intelligence* has been a linguistic one, namely the *Turing Test*: can a dialogue system, responding to a user's typed input with its own textual output, perform so naturally that users cannot distinguish it from a human interlocutor using the same interface? Today, there is substantial ongoing research and development in such areas as machine translation and spoken dialogue, and significant commercial systems are in widespread use. The following *dialogue* illustrates a typical application:

..ex:

```
| S: How may I help you?
| U: When is Saving Private Ryan playing?
| S: For what theater?
| U: The Paramount theater.
| S: Saving Private Ryan is not playing at the Paramount theater, but
|  it's playing at the Madison theater at 3:00, 5:30, 8:00, and 10:30.
```

Today's commercial dialogue systems are strictly limited to narrowly-defined domains. We could not ask the above system to provide driving instructions or details of nearby restaurants unless the requisite information had already been stored and suitable question and answer sentences had been incorporated into the language processing system. Observe that the above system appears to understand the user's goals: the user asks when a movie is showing and the system correctly determines from this that the user wants to see the movie. This inference seems so obvious to humans that we usually do not even notice it has been made, yet a natural language system needs to be endowed with this capability in order to interact naturally. Without it, when asked *Do you know when Saving Private Ryan is playing*, a system might simply — and unhelpfully — respond with a cold *Yes*. While it appears that this dialogue system can perform simple inferences, such sophistication is only found in cutting edge research prototypes. Instead, the developers of commercial dialogue systems use contextual assumptions and simple business logic to ensure that the different ways in which a user might express requests or provide information are handled in a way that makes sense for the particular

application. Thus, whether the user says *When is ...*, or *I want to know when ...*, or *Can you tell me when ...*, simple rules will always yield screening times. This is sufficient for the system to provide a useful service.

Despite some recent advances, it is generally true that those natural language systems which have been fully deployed still cannot perform common-sense reasoning or draw on world knowledge. We can wait for these difficult artificial intelligence problems to be solved, but in the meantime it is necessary to live with some severe limitations on the reasoning and knowledge capabilities of natural language systems. Accordingly, right from the beginning, an important goal of NLP research has been to make progress on the holy grail of natural linguistic interaction *without* recourse to this unrestricted knowledge and reasoning capability. This is an old challenge, and so it is instructive to review the history of the field.

1.3.2 Language and Symbol Processing

The very notion that natural language could be treated in a computational manner grew out of a research program, dating back to the early 1900s, to reconstruct mathematical reasoning using logic, most clearly manifested in work by Frege, Russell, Wittgenstein, Tarski, Lambek and Carnap. This work led to the notion of language as a formal system amenable to automatic processing. Three later developments laid the foundation for natural language processing. The first was **formal language theory**. This defined a language as a set of strings accepted by a class of automata, such as context-free languages and pushdown automata, and provided the underpinnings for computational syntax.

The second development was **symbolic logic**. This provided a formal method for capturing selected aspects of natural language that are relevant for expressing logical proofs. A formal calculus in symbolic logic provides the syntax of a language, together with rules of inference and, possibly, rules of interpretation in a set-theoretic model; examples are propositional logic and First Order Logic. Given such a calculus, with a well-defined syntax and semantics, it becomes possible to associate meanings with expressions of natural language by translating them into expressions of the formal calculus. For example, if we translate *John saw Mary* into a formula `saw(j,m)`, we (implicitly or explicitly) intepret the English verb *saw* as a binary relation, and *John* and *Mary* as denoting individuals. More general statements like *All birds fly* require quantifiers, in this case $\forall$, meaning *for all*: $\forall x(bird(x) \rightarrow fly(x))$. This use of logic provided the technical machinery to perform inferences that are an important part of language understanding.

A closely related development was the **principle of compositionality**, namely that the meaning of a complex expression is comprised of the meaning of its parts and their mode of combination. This principle provided a useful correspondence between syntax and semantics, namely that the meaning of a complex expression could be computed recursively. Consider the sentence *It is not true that p*, where *p* is a proposition. We can represent the meaning of this sentence as $not(p)$. Similarly, we can represent the meaning of *John saw Mary* as $saw(j,m)$. Now we can compute the interpretation of *It is not true that John saw Mary* recursively, using the above information, to get $not(saw(j,m))$.

The approaches just outlined share the premise that computing with natural language crucially relies on rules for manipulating symbolic representations. For a certain period in the development of NLP, particularly during the 1980s, this premise provided a common starting point for both linguists and practioners of NLP, leading to a family of grammar formalisms known as unification-based (or feature-based) grammar, and to NLP applications implemented in the Prolog programming language. Although grammar-based NLP is still a significant area of research, it has become somewhat eclipsed in the last

15–20 years dues to a variety of factors. One significant influence came from automatic speech recognition. Although early work in speech processing adopted a model which emulated the kind of rule-based phonological processing typified by the *Sound Pattern of English* [Chomsky and Halle, 1968], this turned out to be hopelessly inadequate in dealing with the hard problem of recognizing actual speech in anything like real time. By contrast, systems which involved learning patterns from large bodies of speech data were significantly more accurate, efficient and robust. In addition, the speech community found that progress in building better systems was hugely assisted by the construction of shared resources for quantitatively measuring performance against common test data. Eventually, much of the NLP community embraced a **data intensive** orientation to language processing, coupled with a growing use of machine-learning techniques and evaluation-led methodology.

1.3.3 Philosophical Divides

The contrasting approaches to NLP described in the preceding section relate back to early metaphysical debates about **rationalism** versus **empiricism** and **realism** versus **idealism** that occurred in the Enlightenment period of Western philosophy. These debates took place against a backdrop of orthodox thinking in which the source of all knowledge was believed to be divine revelation. During this period of the seventeenth and eighteenth centuries, philosophers argued that human reason or sensory experience has priority over revelation. Descartes and Leibniz, amongst others, took the rationalist position, asserting that all truth has its origins in human thought, and in the existence of "innate ideas" implanted in our minds from birth. For example, they argued that the principles of Euclidean geometry were developed using human reason, and were not the result of supernatural revelation or sensory experience. In contrast, Locke and others took the empiricist view, that our primary source of knowledge is the experience of our faculties, and that human reason plays a secondary role in reflecting on that experience. Prototypical evidence for this position was Galileo's discovery — based on careful observation of the motion of the planets — that the solar system is heliocentric and not geocentric. In the context of linguistics, this debate leads to the following question: to what extent does human linguistic experience, versus our innate "language faculty", provide the basis for our knowledge of language? In NLP this matter surfaces as differences in the priority of corpus data versus linguistic introspection in the construction of computational models. We will return to this issue later in the book.

A further concern, enshrined in the debate between realism and idealism, was the metaphysical status of the constructs of a theory. Kant argued for a distinction between phenomena, the manifestations we can experience, and "things in themselves" which can never been known directly. A linguistic realist would take a theoretical construct like **noun phrase** to be real world entity that exists independently of human perception and reason, and which actually *causes* the observed linguistic phenomena. A linguistic idealist, on the other hand, would argue that noun phrases, along with more abstract constructs like semantic representations, are intrinsically unobservable, and simply play the role of useful fictions. The way linguists write about theories often betrays a realist position, while NLP practitioners occupy neutral territory or else lean towards the idealist position. Thus, in NLP, it is often enough if a theoretical abstraction leads to a useful result; it does not matter whether this result sheds any light on human linguistic processing.

These issues are still alive today, and show up in the distinctions between symbolic vs statistical methods, deep vs shallow processing, binary vs gradient classifications, and scientific vs engineering goals. However, such contrasts are now highly nuanced, and the debate is no longer as polarized as it once was. In fact, most of the discussions — and most of the advances even — involve a "balancing act". For example, one intermediate position is to assume that humans are innately endowed

with analogical and memory-based learning methods (weak rationalism), and to use these methods to identify meaningful patterns in their sensory language experience (empiricism). For a more concrete illustration, consider the way in which statistics from large corpora may serve as evidence for binary choices in a symbolic grammar. For instance, dictionaries describe the words *absolutely* and *definitely* as nearly synonymous, yet their patterns of usage are quite distinct when combined with a following verb, as shown in Table 1.1.

Google hits	*adore*	*love*	*like*	*prefer*
absolutely	289,000	905,00	16,200	644
definitely	1,460	51,000	158,000	62,600
ratio	198:1	18:1	1:10	1:97

Table 1.1: *Absolutely* vs *Definitely* (Liberman 2005, LanguageLog.org)

As you will see, *absolutely adore* is about 200 times as popular as *definitely adore*, while *absolutely prefer* is about 100 times rarer then *definitely prefer*. This information is used by statistical language models, but it also counts as evidence for a symbolic account of word combination in which *absolutely* can only modify extreme actions or attributes, a property that could be represented as a binary-valued feature of certain lexical items. Thus, we see statistical data informing symbolic models. Once this information has been codified symbolically, it is available to be exploited as a contextual feature for statistical language modelling, alongside many other rich sources of symbolic information, like hand-constructed parse trees and semantic representations. Now the circle is closed, and we see symbolic information informing statistical models.

This new rapprochement is giving rise to many exciting new developments. We will touch on some of these in the ensuing pages. We too will perform this balancing act, employing approaches to NLP that integrate these historically-opposed philosophies and methodologies.

1.4 The Architecture of linguistic and NLP systems

1.4.1 Generative Grammar and Modularity

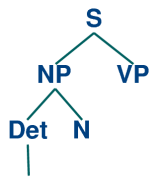
One of the intellectual descendants of formal language theory was the linguistic framework known as **generative grammar**. Such a grammar contains a set of rules that recursively specify (or *generate*) the set of well-formed strings in a language. While there is a wide spectrum of models which owe some allegiance to this core, Chomsky's transformational grammar, in its various incarnations, is probably the best known. In the Chomskyan tradition, it is claimed that humans have distinct kinds of linguistic knowledge, organized into different modules: for example, knowledge of a language's sound structure (**phonology**), knowledge of word structure (**morphology**), knowledge of phrase structure (**syntax**), and knowledge of meaning (**semantics**). In a formal linguistic theory, each kind of linguistic knowledge is made explicit as different **module** of the theory, consisting of a collection of basic elements together with a way of combining them into complex structures. For example, a phonological module might provide a set of phonemes together with an operation for concatenating phonemes into phonological strings. Similarly, a syntactic module might provide labelled nodes as primitives together wih a mechanism for assembling them into trees. A set of linguistic primitives, together with some operators for defining complex elements, is often called a **level of representation**.

As well as defining modules, a generative grammar will prescribe how the modules interact. For example, well-formed phonological strings will provide the phonological content of words, and words will provide the terminal elements of syntax trees. Well-formed syntactic trees will be mapped to semantic representations, and contextual or pragmatic information will ground these semantic representations in some real-world situation.

As we indicated above, an important aspect of theories of generative grammar is that they are intended to model the linguistic knowledge of speakers and hearers; they are not intended to explain how humans actually process linguistic information. This is, in part, reflected in the claim that a generative grammer encodes the **competence** of an idealized native speaker, rather than the speaker's **performance**. A closely related distinction is to say that a generative grammar encodes **declarative** rather than **procedural** knowledge. Declarative knowledge can be glossed as "knowing what", whereas procedural knowledge is "knowing how". As you might expect, computational linguistics has the crucial role of proposing procedural models of language. A central example is parsing, where we have to develop computational mechanisms which convert strings of words into structural representations such as syntax trees. Nevertheless, it is widely accepted that well-engineered computational models of language contain both declarative and procedural aspects. Thus, a full account of parsing will say how declarative knowledge in the form of a grammar and lexicon combines with procedural knowledge which determines how a syntactic analysis should be assigned to a given string of words. This procedural knowledge will be expressed as an **algorithm**: that is, an explicit recipe for mapping some input into an appropriate output in a finite number of steps.

A simple parsing algorithm for context-free grammars, for instance, looks first for a rule of the form $\texttt{S} \rightarrow \texttt{X}_1 \ldots \texttt{X}_n$, and builds a partial tree structure. It then steps through the grammar rules one-by-one, looking for a rule of the form $\texttt{X}_1 \rightarrow \texttt{Y}_1 \ldots \texttt{Y}_j$ which will expand the leftmost daughter introduced by the S rule, and further extends the partial tree. This process continues, for example by looking for a rule of the form $\texttt{Y}_1 \rightarrow \texttt{Z}_1 \ldots \texttt{Z}_k$ and expanding the partial tree appropriately, until the leftmost node label in the partial tree is a lexical category; the parser then checks to see if the first word of the input can belong to the category. To illustrate, let's suppose that the first grammar rule chosen by the parser is S → NP VP and the second rule chosen is NP → Det N; then the partial tree will be as follows:

(8)



If we assume that the input string we are trying to parse is *the cat slept*, we will succeed in identifying *the* as a word which can belong to the category DET. In this case, the parser goes on to the next node of the tree, N, and next input word, *cat*. However, if we had built the same partial tree with an input string *did the cat sleep*, the parse would fail at this point, since *did* is not of category DET. The parser would throw away the structure built so far and look for an alternative way of going from the S node down to a leftmost lexical category (e.g., using a rule S → V NP VP). The important point for now is not the details of this or other parsing algorithms; we discuss this topic much more fully in the chapter on parsing. Rather, we just want to illustrate the idea that an algorithm can be broken down into a fixed number of steps which produce a definite result at the end.

In Figure 1.1 we further illustrate some of these points in the context of a spoken dialogue system, such as our earlier example of an application that offers the user information about movies currently on show.

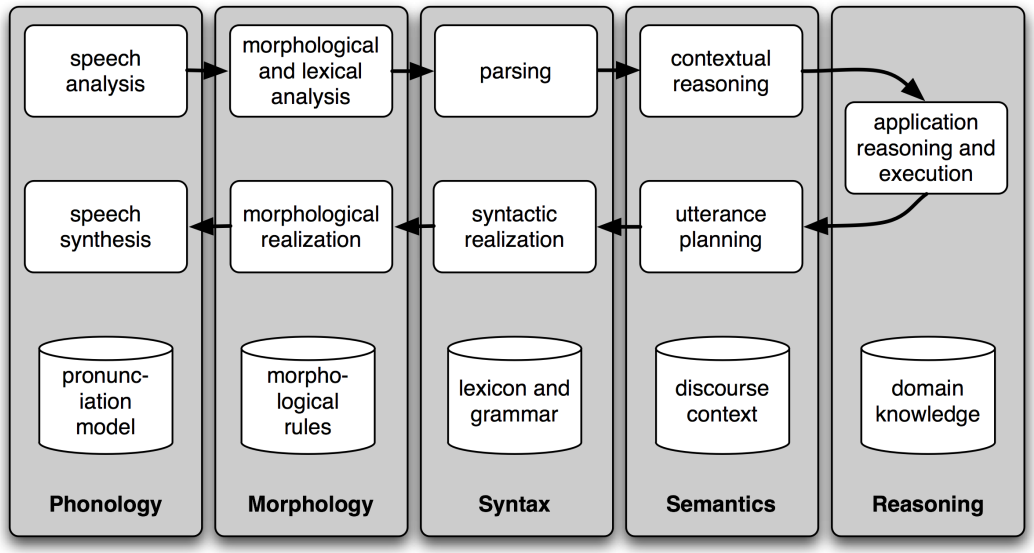


Figure 1.1: Simple Pipeline Architecture for a Spoken Dialogue System

Down the lefthand side of the diagram is a "pipeline" of some representative speech understanding **components**. These map from speech input *via* syntactic parsing to some kind of meaning representation. Up the righthand side is an inverse pipeline of components for concept-to-speech generation. These components constitute the dynamic or procedural aspect of the system's natural language processing. In the central column of the diagram are some representative bodies of static information: the repositories of language-related data which are called upon by the processing components.

The diagram illustrates that linguistically-motivated ways of modularizing linguistic knowledge are often reflected in computational systems. That is, the various components are organized so that the data which they exchange corresponds roughly to different levels of representation. For example, the output of the speech analysis component will contain sequences of phonological representations of words, and the output of the parser will be a semantic representation. Of course the parallel is not precise, in part because it is often a matter of practical expedience where to place the boundaries between different processing components. For example, we can assume that within the parsing component there is a level of syntactic representation, although we have chosen not to expose this at the level of the system diagram. Despite such idiosyncracies, most NLP systems break down their work into a series of discrete steps. In the process of natural language understanding, these steps go from more concrete levels to more abstract ones, while in natural language production, the direction is reversed.

1.5 Before Proceeding Further...

An important aspect of learning NLP using these materials is to experience both the challenge and — we hope — the satisfaction of creating software to process natural language. The accompanying software, NLTK, is available for free and runs on most operating systems including Linux/Unix, Mac OSX and Microsoft Windows. You can download NLTK from <http://nltk.sourceforge.net/>, along with extensive documentation. We encourage you to install Python and NLTK on your machine before reading beyond the end of this chapter.

1.6 Further Reading

Several websites have useful information about NLP, including conferences, resources, and special-interest groups, e.g. `www.lt-world.org`, `www.aclweb.org`, `www.elsnet.org`. The website of the *Association for Computational Linguistics*, at `www.aclweb.org`, contains an overview of computational linguistics, including copies of introductory chapters from recent textbooks. Wikipedia has entries for NLP and its subfields (but don't confuse natural language processing with the other NLP: neuro-linguistic programming.) Three books provide comprehensive surveys of the field: [Cole, 1997], [Dale et al., 2000], [Mitkov, 2002]. Several NLP systems have online interfaces that you might like to experiment with, e.g.:

- WordNet: `http://wordnet.princeton.edu/`
- Translation: `http://world.altavista.com/`
- ChatterBots: `http://www.loebner.net/Prizef/loebner-prize.html`
- Question Answering: `http://www.answerbus.com/`
- Summarization: `http://newsblaster.cs.columbia.edu/`

Part I

BASICS

Chapter 2

Programming Fundamentals and Python

This chapter provides a non-technical overview of Python and will cover the basic programming knowledge needed for the rest of the chapters in Part 1. It contains many examples and exercises; there is no better way to learn to program than to dive in and try these yourself. You should then feel confident in adapting the example for your own purposes. Before you know it you will be programming!

2.1 Python the Calculator

One of the friendly things about Python is that it allows you to type directly into the interactive **interpreter** — the program that will be running your Python programs. We want you to be completely comfortable with this before we begin, so let's start it up:

```
Python 2.4.3 (#1, Mar 30 2006, 11:02:16)
[GCC 4.0.1 (Apple Computer, Inc. build 5250)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

This blurb depends on your installation; the main thing to check is that you are running Python 2.4 or greater (here it is 2.4.3). The `>>>` prompt indicates that the Python interpreter is now waiting for input. If you are using the Python interpreter through the Interactive DeveLopment Environment (IDLE) then you should see a colorized version. We have colorized our examples in the same way, so that you can tell if you have typed the code correctly. Let's begin by using the Python prompt as a calculator:

```
>>> 3 + 2 * 5 - 1
12
>>>
```

There are several things to notice here. First, once the interpreter has finished calculating the answer and displaying it, the prompt reappears. This means the Python interpreter is waiting for another instruction. Second, notice that Python deals with the order of operations correctly (unlike some older calculators), so the multiplication `2 * 5` is calculated before it is added to `3`.

Try a few more expressions of your own. You can use asterisk (`*`) for multiplication and slash (`/`) for division, and parentheses for bracketing expressions. One strange thing you might come across is that division doesn't always behave how you expect:

```
>>> 3/3
1
>>> 1/3
0
>>>
```

The second case is surprising because we would expect the answer to be `0.333333`. We will come back to why that is the case later on in this chapter. For now, let's simply observe that these examples demonstrate how you can work interactively with the interpreter, allowing you to experiment and explore. Also, as you will see later, your intuitions about numerical expressions will be useful for manipulating other kinds of data in Python.

You should also try nonsensical expressions to see how the interpreter handles it:

```
>>> 1 +
Traceback (most recent call last):
  File "<stdin>", line 1
    1 +
      ^
SyntaxError: invalid syntax
>>>
```

Here we have produced a **syntax error**. It doesn't make sense to end an instruction with a plus sign. The Python interpreter indicates the line where the problem occurred.

2.2 Understanding the Basics: Strings and Variables

2.2.1 Representing text

We can't simply type text directly into the interpreter because it would try to interpret the text as part of the Python language:

```
>>> Hello World
Traceback (most recent call last):
  File "<stdin>", line 1
    Hello World
              ^
SyntaxError: invalid syntax
>>>
```

Here we see an error message. Note that the interpreter is confused about the position of the error, and points to the end of the string rather than the start.

Python represents a piece of text using a **string**. Strings are **delimited** — or separated from the rest of the program — by quotation marks:

```
>>> 'Hello World'
'Hello World'
>>> "Hello World"
'Hello World'
>>>
```

We can use either single or double quotation marks, as long as we use the same ones on either end of the string.

Now we can perform calculator-like operations on strings. For example, adding two strings together seems intuitive enough that you could guess the result:

```
>>> 'Hello' + 'World'
'HelloWorld'
>>>
```

When applied to strings, the + operation is called **concatenation**. It produces a new string which is a copy of the two original strings pasted together end-to-end. Notice that concatenation doesn't do anything clever like insert a space between the words. The Python interpreter has no way of knowing that you want a space; it does *exactly* what it is told. Given the example of +, you might be able guess what multiplication will do:

```
>>> 'Hi' + 'Hi' + 'Hi'
'HiHiHi'
>>> 'Hi' * 3
'HiHiHi'
>>>
```

The point to take from this (apart from learning about strings) is that in Python, intuition about what should work gets you a long way, so it is worth just trying things to see what happens. You are very unlikely to break something, so just give it a go.

2.2.2 Storing and reusing values

After a while, it can get quite tiresome to keep retyping Python statements over and over again. It would be nice to be able to store the **value** of an expression like `'Hi' + 'Hi' + 'Hi'` so that we can use it again. We do this by saving results to a location in the computer's memory, and giving the location a name. Such a named place is called a **variable**. In Python we create variables by **assignment**, which involves putting a value into the variable:

```
>>> msg = 'Hello World'          ①
>>> msg                          ②
'Hello World'                    ③
>>>
```

At ① we have created a variable called `msg` (short for 'message') and set it to have the string value `'Hello World'`. We used the = operation, which **assigns** the value of the expression on the right to the variable on the left. Notice the Python interpreter does not print any output; it only prints output when the statement returns a value, and an assignment statement returns no value. At ② we inspect the contents of the variable by naming it on the command line: that is, we use the name `msg`. The interpreter prints out the contents of the variable at ③.

Variables stand in for values, so instead of writing `'Hi' * 3` we could write:

```
>>> msg = 'Hi'
>>> num = 3
>>> msg * num
'HiHiHi'
>>>
```

We can also assign a new value to a variable just by using assignment again:

```
>>> msg = msg * num
>>> msg
'HiHiHi'
>>>
```

Here we have taken the value of `msg`, multiplied it by `3` and then stored that new string (`HiHiHi`) back into the variable `msg`.

2.2.3 Printing and inspecting strings

So far, when we have wanted to look at the contents of a variable or see the result of a calculation, we have just typed the variable name into the interpreter. For example, we can look at the contents of `msg` using:

```
>>> msg
'Hello World'
>>>
```

However, there are some situations where this isn't going to do what we want. To see this, open a text editor, and create a file called `test.py`, containing the single line

```
msg = 'Hello World'
```

Now, open this file in IDLE, then go to the `Run` menu, and select the command `Run Module`. The result in the main IDLE window should look like this:

```
>>> ================================ RESTART ================================
>>>
>>>
```

But where is the output showing the value of `msg`? The answer is that the program in `test.py` will only show a value if you explicitly tell it to, using the `print` command. So add another line to `test.py` so that it looks as follows:

```
msg = 'Hello World'
print msg
```

Select `Run Module` again, and this time you should get output which looks like this:

```
>>> ================================ RESTART ================================
>>>
Hello World
>>>
```

On close inspection, you will see that the quotation marks which indicate that `Hello World` is a string are missing in this case. That is because inspecting a variable (only possible within the interactive interpreter) prints out the Python **representation** of a value, whereas the `print` statement only prints out the value itself, which in this case is just the text in the string.

You will see that you get the same results if you use the `print` command in the interactive interpreter:

```
>>> print msg
Hello World
>>>
```

In fact, you can use a sequence of comma-separated expressions in a `print` statement.

```
>>> msg2 = 'Goodbye'
>>> print msg, msg2
Hello World Goodbye
>>>
```

So, if you want the users of your program to be able to see something then you need to use `print`. If you just want to check the contents of the variable while you are developing your program in the interactive interpreter, then you can just type the variable name directly into the interpreter.

2.2.4 Exercises

1. ☼ Start up the Python interpreter (e.g. by running IDLE). Try the examples in section 2.1, then experiment with using Python as a calculator.

2. ☼ Try the examples in this section, then try the following.

   a) Create a variable called `msg` and put a message of your own in this variable. Remember that strings need to be quoted, so you will need to type something like:

   ```
   >>> msg = "I like NLP!"
   ```

   b) Now print the contents of this variable in two ways, first by simply typing the variable name and pressing enter, then by using the `print` command.

   c) Try various arithmetic expressions using this string, e.g. `msg + msg`, and `5 * msg`.

   d) Define a new string `hello`, and then try `hello + msg`. Change the `hello` string so that it ends with a space character, and then try `hello + msg` again.

2.3 Slicing and Dicing

Strings are so important (especially for NLP!) that we will spend some more time on them. Here we will learn how to access the individual **characters** that make up a string, how to pull out arbitrary **substrings**, and how to reverse strings.

2.3.1 Accessing individual characters

The positions within a string are numbered, starting from zero. To access a position within a string, we specify the position inside square brackets:

```
>>> msg = 'Hello World'
>>> msg[0]
'H'
>>> msg[3]
'l'
>>> msg[5]
' '
>>>
```

This is called **indexing** or **subscripting** the string. The position we specify inside the square brackets is called the **index**. We can retrieve not only letters but any character, such as the space at index `5`.

> **Note**
>
> Be careful to distinguish between the string `' '`, which is a single whitespace character, and `''`, which is the empty string.

The fact that strings are numbered from zero may seem counter-intuitive. However, it goes back to the way variables are stored in a computer's memory. As mentioned earlier, a variable is actually the

name of a location, or **address**, in memory. Strings are arbitrarily long, and their address is taken to be the position of their first character. Thus, if we assign `three = 3` and `msg = 'Hello World'` then the location of those values will be along the lines shown in Figure 2.1.

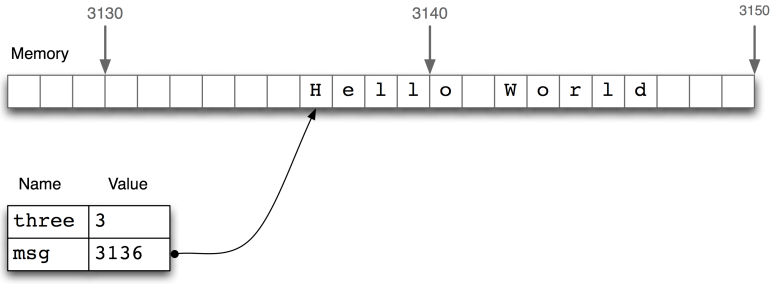


Figure 2.1: Variables and Computer Memory

When we index into a string, the computer adds the index to the string's address. Thus `msg[3]` is found at memory location `3136 + 3`. Accordingly, the first position in the string is found at `3136 + 0`, or `msg[0]`.

If you don't find Figure 2.1 helpful, you might just want to think of indexes as giving you the position in a string immediately *before* a character, as indicated in Figure 2.2.

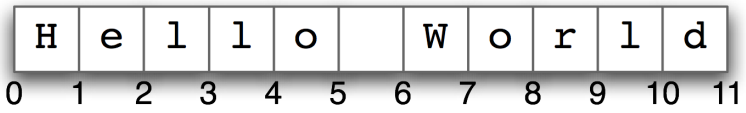


Figure 2.2: String Indexing

Now, what happens when we try to access an index that is outside of the string?

```
>>> msg[11]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: string index out of range
>>>
```

The index of `11` is outside of the range of valid indices (i.e., 0 to 10) for the string `'Hello World'`. This results in an error message. This time it is not a syntax error; the program fragment is syntactically correct. Instead, the error occurred while the program was running. The `Traceback` message indicates which line the error occurred on (line 1 of 'standard input'). It is followed by the name of the error, `IndexError`, and a brief explanation.

In general, how do we know what we can index up to? If we know the length of the string is n, the highest valid index will be $n - 1$. We can get access to the length of the string using the `len()` function.

```
>>> len(msg)
11
>>>
```

Informally, a **function** is a named snippet of code that provides a service to our program when we **call** or execute it by name. We call the `len()` function by putting parentheses after the name and giving it the string `msg` we want to know the length of. Because `len()` is built into the Python interpreter, IDLE colors it purple.

We have seen what happens when the index is too large. What about when it is too small? Let's see what happens when we use values less than zero:

```
>>> msg[-1]
'd'
>>>
```

This does not generate an error. Instead, negative indices work from the *end* of the string, so `-1` indexes the last character, which is `'d'`.

```
>>> msg[-3]
'r'
>>> msg[-6]
' '
>>>
```

Now the computer works out the location in memory relative to the string's address plus its length, e.g. `3136 + 11 - 1 = 3146`. We can also visualize negative indices as shown in Figure 2.3.

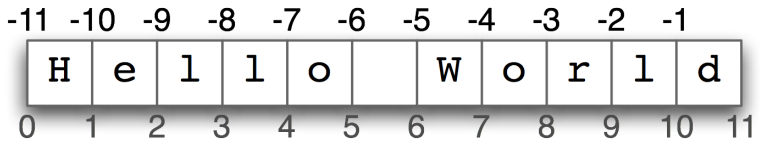


Figure 2.3: Negative Indices

Thus we have two ways to access the characters in a string, from the start or the end. For example, we can access the space in the middle of `Hello` and `World` with either `msg[5]` or `msg[-6]`; these refer to the same location, because `5 = len(msg) - 6`.

2.3.2 Accessing substrings

Next, we might want to access more than one character at a time. This is also pretty simple; we just need to specify a range of characters for indexing rather than just one. This process is called **slicing** and we indicate a slice using a colon in the square brackets to separate the beginning and end of the range:

```
>>> msg[1:4]
'ell'
>>>
```

Here we see the characters are `'e'`, `'l'` and `'l'` which correspond to `msg[1]`, `msg[2]` and `msg[3]`, but not `msg[4]`. This is because a slice *starts* at the first index but finishes *one before* the end index. This is consistent with indexing: indexing also starts from zero and goes up to *one before* the length of the string. We can see that by indexing with the value of `len()`:

```
>>> len(msg)
11
>>> msg[0:11]
'Hello World'
>>>
```

We can also slice with negative indices — the same basic rules of starting from the start index and stopping one before the end index applies; here we stop before the space character:

```
>>> msg[0:-6]
'Hello'
>>>
```

Python provides two shortcuts for commonly used slice values. If the start index is `0` then you can leave it out entirely, and if the end index is the length of the string then you can leave it out entirely:

```
>>> msg[:3]
'Hel'
>>> msg[6:]
'World'
>>>
```

The first example above selects the first three characters from the string, and the second example selects from the character with index 6, namely `'W'`, to the end of the string. These shortcuts lead to a couple of common Python idioms:

```
>>> msg[:-1]
'Hello Worl'
>>> msg[:]
'Hello World'
>>>
```

The first chomps off just the last character of the string, and the second makes a complete copy of the string (which is more important when we come to lists below).

2.3.3 Exercises

1. ☼ Define a string `s = 'colorless'`. Write a Python statement that changes this to "colourless" using only the slice and concatenation operations.

2. ☼ Try the slice examples from this section using the interactive interpreter. Then try some more of your own. Guess what the result will be before executing the command.

3. ☼ We can use the slice notation to remove morphological endings on words. For example, `'dogs'[:-1]` removes the last character of `dogs`, leaving `dog`. Use slice notation to remove the affixes ending from these words (we've inserted a hyphen to indicate the affix boundary, but omit this from your strings): `dish-es`, `run-ning`, `nation-ality`, `un-do`, `pre-heat`.

4. ☼ We saw how we can generate an `IndexError` by indexing beyond the end of a string. Is it possible to construct an index that goes too far to the left, before the start of the string?

5. ☼ We can also specify a step size for the slice. The following returns every second character within the slice, in a forwards or reverse direction:

```
>>> msg[6:11:2]
'Wrd'
>>> msg[10:5:-2]
'drW'
>>>
```

Experiment with different step values.

6. ☼ What happens if you ask the interpreter to evaluate `msg[::-1]`? Explain why this is a reasonable result.

2.4 Strings, Sequences, and Sentences

We have seen how words like *Hello* can be stored as a string `'Hello'`. Whole sentences can also be stored in strings, and manipulated as before, as we can see here for Chomsky's famous nonsense sentence:

```
>>> sent = 'colorless green ideas sleep furiously'
>>> sent[16:21]
'ideas'
>>> len(sent)
37
>>>
```

However, it turns out to be a bad idea to treat a sentence as a sequence of its characters, because this makes it too inconvenient to access the words or work out the length. Instead, we would prefer to represent a sentence as a sequence of its *words*; as a result, indexing a sentence accesses the words, rather than characters. We will see how to do this now.

2.4.1 Lists

A **list** is designed to store a sequence of values. A list is similar to a string in many ways except that individual items don't have to be just characters; they can be arbitrary strings, integers or even other lists.

A Python list is represented as a sequence of comma-separated items, delimited by square brackets. Let's create part of Chomsky's sentence as a list and put it in a variable `phrase1`:

```
>>> phrase1 = ['colorless', 'green', 'ideas']
>>> phrase1
['colorless', 'green', 'ideas']
>>>
```

Because lists and strings are both kinds of sequence, they can be processed in similar ways; just as strings support `len()`, indexing and slicing, so do lists. The following example applies these familiar operations to the list `phrase1`:

```
>>> len(phrase1)
3
>>> phrase1[0]
'colorless'
>>> phrase1[-1]
```

```
'ideas'
>>> phrase1[-5]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
>>>
```

Here, `phrase1[-5]` generates an error, because the fifth-last item in a three item list would occur before the list started, i.e., it is undefined. We can also slice lists in exactly the same way as strings:

```
>>> phrase1[1:3]
['green', 'ideas']
>>> phrase1[-2:]
['green', 'ideas']
>>>
```

Lists can be concatenated just like strings. Here we will put the resulting list into a new variable `phrase2`. The original variable `phrase1` is not changed in the process:

```
>>> phrase2 = phrase1 + ['sleep', 'furiously']
>>> phrase2
['colorless', 'green', 'ideas', 'sleep', 'furiously']
>>> phrase1
['colorless', 'green', 'ideas']
>>>
```

Now, lists and strings do not have exactly the same functionality. Lists have the added power that you can change their elements. Let's imagine that we want to change the 0th element of `phrase1` to `'colorful'`, we can do that by assigning to the index `phrase1[0]`:

```
>>> phrase1[0] = 'colorful'
>>> phrase1
['colorful', 'green', 'ideas']
>>>
```

On the other hand if we try to do that with a string (for example changing the 0th character in `msg` to `'J'` we get:

```
>>> msg[0] = 'J'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object does not support item assignment
>>>
```

This is because strings are **immutable** — you can't change a string once you have created it. However, lists are **mutable**, and their contents can be modified at any time. As a result, lists support a number of operations, or **methods**, which modify the original value rather than returning a new value.

> **Note**
>
> Methods are functions, so they can be called in a similar manner. However, as we will see later on in this book, methods are tightly associated with objects of that belong to specific **classes** (for example, strings and lists). A method is called on a particular object using the object's name, then a period, then the name of the method, and finally the parentheses containing any arguments.

Two of these methods are **sorting** and **reversing**:

```
>>> phrase2.sort()
>>> phrase2
['colorless', 'furiously', 'green', 'ideas', 'sleep']
>>> phrase2.reverse()
>>> phrase2
['sleep', 'ideas', 'green', 'furiously', 'colorless']
>>>
```

As you will see, the prompt reappears immediately on the line after `phrase2.sort()` and `phrase2.reverse()`. That is because these methods do not return a new list, but instead modify the original list stored in the variable `phrase2`.

Lists also support an `append()` method for adding items to the end of the list and an `index` method for finding the index of particular items in the list:

```
>>> phrase2.append('said')
>>> phrase2.append('Chomsky')
>>> phrase2
['sleep', 'ideas', 'green', 'furiously', 'colorless', 'said', 'Chomsky']
>>> phrase2.index('green')
2
>>>
```

Finally, just as a reminder, you can create lists of any values you like. They don't even have to be the same type, although this is rarely a good idea:

```
>>> bat = ['bat', [[1, 'n', 'flying mammal'], [2, 'n', 'striking instrument']]]
>>>
```

2.4.2 Working on sequences one item at a time

We have shown you how to create lists, and how to index and manipulate them in various ways. Often it is useful to step through a list and process each item in some way. We do this using a `for` loop. This is our first example of a **control structure** in Python, a statement that *controls* how other statements are run:

```
>>> for word in phrase2:
...     print len(word), word
5 sleep
5 ideas
5 green
9 furiously
9 colorless
4 said
7 Chomsky
```

This program runs the statement `print len(word), word` for every item in the list of words. This process is called **iteration**. Each iteration of the `for` loop starts by assigning the next item of the list `phrase2` to the **loop variable** `word`. Then the indented **body** of the loop is run. Here the body consists of a single command, but in general the body can contain as many lines of code as you want, so long as they are all indented by the same amount.

> **Note**
>
> The interactive interpreter changes the prompt from `>>>` to the `...` prompt after encountering a colon (`:`). This indicates that the interpreter is expecting an indented block of code to appear next. However, it is up to you to do the indentation. To finish the indented block just enter a blank line.

We can run another `for` loop over the Chomsky nonsense sentence, and calculate the average word length. As you will see, this program uses the `len()` function in two ways: to count the number of characters in a word, and to count the number of words in a phrase. Note that `x += y` is shorthand for `x = x + y`; this idiom allows us to **increment** the `total` variable each time the loop is run.

```
>>> total = 0
>>> for word in phrase2:
...     total += len(word)
...
>>> total / len(phrase2)
6
>>>
```

Finally, note that we can write `for` loops to iterate over the characters in strings.

```
>>> sent = 'colorless green ideas sleep furiously'
>>> for char in sent:
...     print char,
c o l o r l e s s  g r e e n  i d e a s  s l e e p  f u r i o u s l y
```

2.4.3 Tuples

Python tuples are just like lists, except that there is one important difference: tuples cannot be changed in place, for example by `sort()` or `reverse()`. In other words, like strings they are immutable. Tuples are formed with enclosing parentheses rather than square brackets, and items are separated by commas. Like lists, tuples can be indexed and sliced.

```
>>> t = ('walk', 'fem', 3)
>>> t[0]
'walk'
>>> t[1:]
('fem', 3)
>>> t[0] = 'run'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object does not support item assignment
>>>
```

2.4.4 String Formatting

The output of a program is usually structured to make the information easily digestible by a reader. Instead of running some code and then manually inspecting the contents of a variable, we would like the code to tabulate some output. We already saw this above in the first `for` loop example, where each line of output was similar to `5 sleep`, consisting of a word length, followed by the word in question.

There are many ways we might want to format such output. For instance, we might want to place the length value in parentheses *after* the word, and print all the output an a single line:

```
>>> for word in phrase2:
...     print word, '(', len(word), '),',
sleep ( 5 ), ideas ( 5 ), green ( 5 ), furiously ( 9 ), colorless ( 9 ),
said ( 4 ), Chomsky ( 7 ),
```

Notice that this `print` statement ends with a trailing comma, which is how we tell Python not to print a newline at the end.

However, this approach has a couple of problems. First, the `print` statement intermingles variables and punctuation, making it a little difficult to read. Second, the output has spaces around every item that was printed. A cleaner way to produce structured output uses Python's **string-formatting expressions**. Here's an example:

```
>>> for word in phrase2:
...     print "%s (%d)," % (word, len(word)),
sleep (5), ideas (5), green (5), furiously (9), colorless (9),
said (4), Chomsky (7),
```

Here, the `print` command is followed by a three-part object having the syntax: *`format`* `%` *`values`*. The *`format`* section is a string containing **format specifiers** such as `%s` and `%d` which Python will replace with the supplied values. The `%s` specifier tells Python that the corresponding variable is a string (or should be converted into a string), while the `%d` specifier indicates that the corresponding variable should be converted into a decimal representation. Finally, the *`values`* section of a formatting string is a tuple containing exactly as many items as there are format specifiers in the *`format`* section. (We will discuss Python's string-formatting expressions in more detail in Section 6.3.2).

In the above example, we used a trailing comma to suppress the printing of a newline. Suppose, on the other hand, that we want to introduce some additional newlines in our output. We can accomplish this by inserting the 'special' character `\n` into the `print` string:

```
>>> for word in phrase2:
...     print "Word = %s\nIndex = %s\n*****" % (word, phrase2.index(word))
...
Word = sleep
Index = 0
*****
Word = ideas
Index = 1
*****
Word = green
Index = 2
*****
Word = furiously
Index = 3
*****
Word = colorless
Index = 4
*****
Word = said
Index = 5
*****
Word = Chomsky
Index = 6
```

```
*****
>>>
```

2.4.5 Character encoding and Unicode

Our programs will often need to deal with different languages, and different character sets. The concept of "plain text" is a fiction. If you live in the English-speaking world you probably use ASCII. If you live in Europe you might use one of the extended Latin character sets, containing such characters as "ø" for Danish and Norwegian, "ő" for Hungarian, "ñ" for Spanish and Breton, and "ň" for Czech and Slovak.

[See `http://www.amk.ca/python/howto/unicode` for more information on Unicode and Python.]

2.4.6 Converting between strings and lists

Often we want to convert between a string containing a space-separated list of words and a list of strings. Let's first consider turning a list into a string. One way of doing this is as follows:

```
>>> str = ''
>>> for word in phrase2:
...     str += ' ' + word
...
>>> str
' sleep ideas green furiously colorless said Chomsky'
>>>
```

One drawback of this approach is that we have an unwanted space at the start of `str`. It is more convenient to use the `string.join()` method:

```
>>> import string
>>> phrase3 = string.join(phrase2)
>>> phrase3
'sleep ideas green furiously colorless said Chomsky'
>>>
```

Now let's try to reverse the process: that is, we want to convert a string into a list. Again, we could start off with an empty list `[]` and `append()` to it within a `for` loop. But as before, there is a more succinct way of achieving the same goal. This time, we will *split* the new string `phrase3` on the whitespace character:

```
>>> phrase3.split(' ')
['sleep', 'ideas', 'green', 'furiously', 'colorless', 'said', 'Chomsky']
>>> phrase3.split('s')
['', 'leep idea', ' green furiou', 'ly colorle', '', ' ', 'aid Chom', 'ky']
>>>
```

We can also split on any character, so we tried splitting on `'s'` as well.

2.4.7 Exercises

1. ☼ Using the Python interactive interpreter, experiment with the examples in this section. Think of a sentence and represent it as a list of strings, e.g. ['Hello', 'world']. Try the various operations for indexing, slicing and sorting the elements of your list. Extract individual items (strings), and perform some of the string operations on them.

2. ☼ We pointed out that when `phrase` is a list, `phrase.reverse()` returns a modified version of `phrase` rather than a new list. On the other hand, we can use the slice trick mentioned in the exercises for the previous section, `[::-1]` to create a *new* reversed list without changing `phrase`. Show how you can confirm this difference in behaviour.

3. ☼ We have seen how to represent a sentence as a list of words, where each word is a sequence of characters. What does `phrase1[2][2]` do? Why? Experiment with other index values.

4. ☼ Write a `for` loop to print out the characters of a string, one per line.

5. ☼ What happens if you call `split` on a string, with no argument, e.g. `phrase3.split()`? What happens when the string being split contains tab characters, consecutive space characters, or a sequence of tabs and spaces?

6. ☼ Create a variable `words` containing a list of words. Experiment with `words.sort()` and `sorted(words)`. What is the difference?

7. ◑ Process the list `phrase2` using a `for` loop, and store the result in a new list `lengths`. Hint: begin by assigning the empty list to `lengths`, using `lengths = []`. Then each time through the loop, use `append()` to add another length value to the list.

8. ◑ Define a variable `silly` to contain the string: `'newly formed bland ideas are unexpressible in an infuriating way'`. (This happens to be the legitimate interpretation that bilingual English-Spanish speakers can assign to Chomsky's famous phrase, according to Wikipedia). Now write code to perform the following tasks:

   a) Split `silly` into a list of strings, one per word, using Python's `split()` operation.

   b) Extract the second letter of each word in `silly` and join them into a string, to get `'eoldrnnnna'`.

   c) Combine the words in `phrase4` back into a single string, using `join()`. Make sure the words in the resulting string are separated with whitespace.

   d) Print the words of `silly` in alphabetical order, one per line.

9. ◑ The `index()` function can be used to look up items in sequences. For example, `'unexpressible'.index('e')` tells us the index of the first position of the letter `e`.

   a) What happens when you look up a substring, e.g. `'unexpressible'.index('re')`?

   b) Define a variable `words` containing a list of words. Now use `words.index()` to look up the position of an individual word.

c) Define a variable `silly` as in the exercise above. Use the `index()` function in combination with list slicing to build a list `phrase` consisting of all the words up to (but not including) `in` in `silly`.

2.5 Making Decisions

So far, our simple programs have been able to manipulate sequences of words, and perform some operation on each one. We applied this to lists consisting of a few words, but the approach works the same for lists of arbitrary size, containing thousands of items. Thus, such programs have some interesting qualities: (i) the ability to work with language, and (ii) the potential to save human effort through automation. Another useful feature of programs is their ability to *make decisions* on our behalf; this is our focus in this section.

2.5.1 Making simple decisions

Most programming languages permit us to execute a block of code when a **conditional expression**, or `if` statement, is satisfied. In the following program, we have created a variable called `word` containing the string value `'cat'`. The `if` statement then checks whether the condition `len(word) < 5` is true. Because the conditional expression is true, the body of the `if` statement is invoked and the `print` statement is executed.

```
>>> word = "cat"
>>> if len(word) < 5:
...     print 'word length is less than 5'
...
word length is less than 5
>>>
```

If we change the conditional expression to `len(word) >= 5` — the length of `word` is greater than or equal to `5` — then the conditional expression will no longer be true, and the body of the `if` statement will not be run:

```
>>> if len(word) >= 5:
...     print 'word length is greater than or equal to 5'
...
>>>
```

The `if` statement, just like the `for` statement above is a **control structure**. An `if` statement is a control structure because it controls whether the code in the body will be run. You will notice that both `if` and `for` have a colon at the end of the line, before the indentation begins. That's because all Python control structures end with a colon.

What if we want to do something when the conditional expression is not true? The answer is to add an `else` clause to the `if` statement:

```
>>> if len(word) >= 5:
...     print 'word length is greater than or equal to 5'
... else:
...     print 'word length is less than 5'
...
word length is less than 5
>>>
```

Finally, if we want to test multiple conditions in one go, we can use an `elif` clause which acts like an `else` and an `if` combined:

```
>>> if len(word) < 3:
...   print 'word length is less than three'
... elif len(word) == 3:
...   print 'word length is equal to three'
... else:
...   print 'word length is greater than three'
...
word length is equal to three
>>>
```

2.5.2 Conditional expressions

Python supports a wide range of operators like < and >= for testing the relationship between values. The full set of these **relational operators** are shown in Table inequalities.

Operator	Relationship
<	less than
<=	less than or equal to
==	equal to (note this is two not one = sign)
!=	not equal to
>	greater than
>=	greater than or equal to

Table 2.1:

Normally we use conditional expressions as part of an `if` statement. However, we can test these relational operators directly at the prompt:

```
>>> 3 < 5
True
>>> 5 < 3
False
>>> not 5 < 3
True
>>>
```

Here we see that these expressions have **Boolean** values, namely `True` or `False`. `not` is a Boolean operator, and flips the truth value of Boolean statement.

Strings and lists also support conditional operators:

```
>>> word = 'sovereignty'
>>> 'sovereign' in word
True
>>> 'gnt' in word
True
>>> 'pre' not in word
```

```
True
>>> 'Hello' in ['Hello', 'World']
True
>>> 'Hell' in ['Hello', 'World']
False
>>>
```

Strings also have methods for testing what appears at the beginning and the end of a string (as opposed to just anywhere in the string:

```
>>> word.startswith('sovereign')
True
>>> word.endswith('ty')
True
>>>
```

Note

Integers, strings and lists are all kinds of **data types** in Python. In fact, every value in Python has a type. The type determines what operations you can perform on the data value. So, for example, we have seen that we can index strings and lists, but we can't index integers:

```
>>> one = 'cat'
>>> one[0]
'c'
>>> two = [1, 2, 3]
>>> two[1]
2
>>> three = 3
>>> three[2]
Traceback (most recent call last):
  File "<pyshell#95>", line 1, in -toplevel-
    three[2]
TypeError: 'int' object is unsubscriptable
>>>
```

You can use Python's `type()` function to check what the type of an object is:

```
>>> data = [one, two, three]
>>> for item in data:
...     print "item '%s' belongs to %s" % (item, type(item))
...
item 'cat' belongs to <type 'str'>
item '[1, 2, 3]' belongs to <type 'list'>
item '3' belongs to <type 'int'>
>>>
```

Because strings and lists (and tuples) have so much in common, they are grouped together in a higher level type called **sequences**.

2.5.3 Iteration, items, and `if`

Now it is time to put some of the pieces together. We are going to take the string `'how now brown cow'` and print out all of the words ending in `'ow'`. Let's build the program up in stages. The first step is to split the string into a list of words:

```
>>> sentence = 'how now brown cow'
>>> words = sentence.split()
>>> words
['how', 'now', 'brown', 'cow']
>>>
```

Next, we need to iterate over the words in the list. Just so we don't get ahead of ourselves, let's print each word, one per line:

```
>>> for word in words:
...     print word
...
how
now
brown
cow
```

The next stage is to only print out the words if they end in the string `'ow'`. Let's check that we know how to do this first:

```
>>> 'how'.endswith('ow')
True
>>> 'brown'.endswith('ow')
False
>>>
```

Now we are ready to put an `if` statement inside the `for` loop. Here is the complete program:

```
>>> sentence = 'how now brown cow'
>>> words = sentence.split()
>>> for word in words:
...     if word.endswith('ow'):
...         print word
...
how
now
cow
>>>
```

As you can see, even with this small amount of Python knowledge it is possible to develop useful programs. The key idea is to develop the program in pieces, testing that each one does what you expect, and then combining them to produce whole programs. This is why the Python interactive interpreter is so invaluable, and why you should get comfortable using it.

2.5.4 Exercises

1. ☼ Assign a new value to `sentence`, namely the string `'she sells sea shells by the sea shore'`, then write code to perform the following tasks:

a) Print all words beginning with `'sh'`:

b) Print all words longer than 4 characters.

c) Generate a new sentence that adds the popular hedge word `'like'` before every word beginning with `'se'`. Your result should be a single string.

2. ☼ Write code to abbreviate text by removing all the vowels. Define `sentence` to hold any string you like, then initialize a new string `result` to hold the empty string `''`. Now write a `for` loop to process the string, one character at a time, and append any non-vowel characters to the result string.

3. ◑ Write conditional expressions, such as `'H' in msg`, but applied to lists instead of strings. Check whether particular words are included in the Chomsky nonsense sentence.

4. ◑ Write code to convert text into *hAck3r*, where characters are mapped according to the following table:

Input:	e	i	o	l	s	.	ate
Output:	3	1	0	|	5	5w33t!	8

Table 2.2:

2.6 Getting organized

Strings and lists are a simple way to organize data. In particular, they **map** from integers to values. We can 'look up' a string using an integer to get one of its letters, and we can also look up a list of words using an integer to get one of its strings. These cases are shown in Figure 2.4.

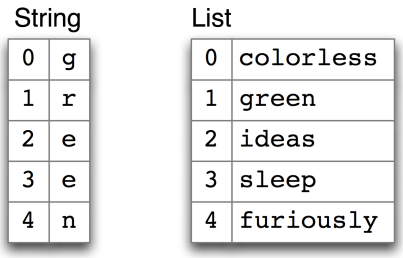


Figure 2.4: Sequence Look-up

However, we need a more flexible way to organize and access our data. Consider the examples in Figure 2.5.

In the case of a phone book, we look up an entry using a *name*, and get back a number. When we type a domain name in a web browser, the computer looks this up to get back an IP address. A word frequency table allows us to look up a word and find its frequency in a text collection. In all these cases, we are mapping from names to numbers, rather than the other way round as with indexing into sequences. In general, we would like to be able to map between arbitrary types of information. The following table lists a variety of linguistic objects, along with what they map.

Phone List

Alex	x154
Dana	x642
Kim	x911
Les	x120
Sandy	x124

Domain Name Resolution

aclweb.org	128.231.23.4
amazon.com	12.118.92.43
google.com	28.31.23.124
pythonb.org	18.21.3.144
sourceforge.net	51.98.23.53

Word Frequency Table

computational	25
language	196
linguistics	17
natural	56
processing	57

Figure 2.5: Dictionary Look-up

Linguistic Object	**Maps**	
	from	**to**
Document Index	Word	List of pages (where word is found)
Thesaurus	Word sense	List of synonyms
Dictionary	Headword	Entry (part of speech, sense definitions, etymology)
Comparative Wordlist	Gloss term	Cognates (list of words, one per language)
Morph Analyzer	Surface form	Morphological analysis (list of component morphemes)

Table 2.3:

Most often, we are mapping from a string to some structured object. For example, a document index maps from a word (which we can represent as a string), to a list of pages (represented as a list of integers). In this section, we will see how to represent such mappings in Python.

2.6.1 Accessing data with data

Python provides a **dictionary** data type, which can be used for mapping between arbitrary types.

> **Note**
>
> A Python dictionary is somewhat like a linguistic dictionary — they both give you a systematic means of looking things up, and so there is some potential for confusion. However, we hope that it will usually be clear from the context which kind of dictionary we are tallking about.

Here we define `pos` to be an empty dictionary and then add three entries to it, specifying the part-of-speech of some words. We add entries to a dictionary using the familiar square bracket notation:

```
>>> pos = {}
>>> pos['colorless'] = 'adj'
>>> pos['furiously'] = 'adv'
>>> pos['ideas'] = 'n'
>>>
```

So, for example, `pos['colorless'] = 'adj'` says that the look-up value of `'colorless'` in `pos` is the string `'adj'`.

To look up a value in `pos`, we again use indexing notation, except now the thing inside the square brackets is the item whose value we want to recover:

```
>>> pos['ideas']
'n'
>>> pos['colorless']
'adj'
>>>
```

The item used for look-up is called the **key**, and the data that is returned is known as the **value**. As with indexing a list or string, we get an exception when we try to access the value of a key that does not exist:

```
>>> pos['missing']
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
KeyError: 'missing'
>>>
```

This raises an important question. Unlike lists and strings, where we can use `len()` to work out which integers will be legal indices, how do we work out the legal keys for a dictionary? Fortunately, we can check whether a key exists in a dictionary using the `in` operator:

```
>>> 'colorless' in pos
True
>>> 'missing' in pos
False
>>> 'missing' not in pos
True
>>>
```

Notice that we can use `not in` to check if a key is *missing*. Be careful with the `in` operator for dictionaries: it only applies to the keys and not their values. If we check for a value, e.g. `'adj' in pos`, the result is `False`, since `'adj'` is not a key. We can loop over all the entries in a dictionary using a `for` loop.

```
>>> for word in pos:
...     print "%s (%s)" % (word, pos[word])
...
colorless (adj)
furiously (adv)
ideas (n)
>>>
```

We can see what the contents of the dictionary look like by inspecting the variable `pos`:

```
>>> pos
{'furiously': 'adv', 'ideas': 'n', 'colorless': 'adj'}
>>>
```

Here, the contents of the dictionary are shown as **key-value pairs**. As you can see, the order of the key-value pairs is different from the order in which they were originally entered. This is because dictionaries are not sequences but mappings. The keys in a mapping are not inherently ordered, and any ordering that we might want to impose on the keys exists independently of the mapping. As we shall see later, this gives us a lot of flexibility.

We can use the same key-value pair format to create a dictionary:

```
>>> pos = {'furiously': 'adv', 'ideas': 'n', 'colorless': 'adj'}
>>>
```

Using the dictionary methods `keys()`, `values()` and `items()`, we can access the keys and values as separate lists, and also the key-value pairs:

```
>>> pos.keys()
['colorless', 'furiously', 'ideas']
>>> pos.values()
['adj', 'adv', 'n']
>>> pos.items()
[('colorless', 'adj'), ('furiously', 'adv'), ('ideas', 'n')]
>>>
```

2.6.2 Counting with dictionaries

The values stored in a dictionary can be any kind of object, not just a string — the values can even be dictionaries. The most common kind is actually an integer. It turns out that we can use a dictionary to store **counters** for many kinds of data. For instance, we can have a counter for all the letters of the alphabet; each time we get a certain letter we increment its corresponding counter:

```
>>> phrase = 'colorless green ideas sleep furiously'
>>> count = {}
>>> for letter in phrase:
...     if letter not in count:
...         count[letter] = 0
...     count[letter] += 1
>>> count
{'a': 1, ' ': 4, 'c': 1, 'e': 6, 'd': 1, 'g': 1, 'f': 1, 'i': 2,
 'l': 4, 'o': 3, 'n': 1, 'p': 1, 's': 5, 'r': 3, 'u': 2, 'y': 1}
```

Observe that `in` is used here in two different ways: `for letter in phrase` iterates over every letter, running the body of the `for` loop. Inside this loop, the conditional expression `if letter not in count` checks whether the letter is missing from the dictionary. If it is missing, we create a new entry and set its value to zero: `count[letter] = 0`. Now we are sure that the entry exists, and it may have a zero or non-zero value. We finish the body of the `for` loop by incrementing this particular counter using the += assignment operator. Finally, we print the dictionary, to see the letters and their counts. This method of maintaining many counters will find many uses, and you will become very familiar with it.

There are other useful ways to display the result, such as sorting alphabetically by the letter:

```
>>> sorted(count.items())
[(' ', 4), ('a', 1), ('c', 1), ('d', 1), ('e', 6), ('f', 1), ...,
...('y', 1)]
```

> **Note**
>
> The function `sorted()` is similar to the `sort()` method on sequences, but rather than sorting in-place, it produces a new sorted copy of its argument. Moreover, as we will see very soon, `sorted()` will work on a wider variety of data types, including dictionaries.

2.6.3 Getting unique entries

Sometimes, we don't want to count at all, but just want to make a record of the items that we have seen, regardless of repeats. For example, we might want to compile a vocabulary from a document. This is a sorted list of the words that appeared, regardless of frequency. At this stage we have two ways to do this. The first uses lists.

```
>>> sentence = "she sells sea shells by the sea shore".split()
>>> words = []
>>> for word in sentence:
...     if word not in words:
...         words.append(word)
...
>>> sorted(words)
['by', 'sea', 'sells', 'she', 'shells', 'shore', 'the']
```

We can write this using a dictionary as well. Each word we find is entered into the dictionary as a key. We use a value of `1`, but it could be anything we like. We extract the keys from the dictionary simply by converting the dictionary to a list:

```
>>> found = {}
>>> for word in sentence:
...     found[word] = 1
...
>>> sorted(found)
['by', 'sea', 'sells', 'she', 'shells', 'shore', 'the']
```

There is a third way to do this, which is best of all: using Python's **set** data type. We can convert `sentence` into a set, using `set(sentence)`:

```
>>> set(sentence)
set(['shells', 'sells', 'shore', 'she', 'sea', 'the', 'by'])
```

The order of items in a set is not significant, and they will usually appear in a different order to the one they were entered in. The main point here is that converting a list to a set removes any duplicates. We convert it back into a list, sort it, and print. Here is the complete program:

```
>>> sentence = "she sells sea shells by the sea shore".split()
>>> sorted(set(sentence))
['by', 'sea', 'sells', 'she', 'shells', 'shore', 'the']
```

Here we have seen that there is sometimes more than one way to solve a problem with a program. In this case, we used three different built-in data types, a list, a dictionary, and a set. The set data type mostly closely modelled our task, so it required the least amount of work.

2.6.4 Scaling it up

We can use dictionaries to count word occurrences. For example, the following code reads *Macbeth* and counts the frequency of each word:

```
>>> from nltk_lite.corpora import gutenberg
>>> count = {}                                          # initialize a dictionary
>>> for word in gutenberg.raw('shakespeare-macbeth'): # tokenize Macbeth
...     word = word.lower()                             # normalize to lowercase
...     if word not in count:                           # seen this word before?
...         count[word] = 0                             # if not, set count to zero
...     count[word] += 1                                # increment the counter
...
>>>
```

This example demonstrates some of the convenience of NLTK in accessing corpora. We will see much more of this later. For now, all you need to know is that `gutenberg.raw()` returns a list of words, in this case from Shakespeare's play *Macbeth*, which we are iterating over using a `for` loop. We convert each word to lowercase using the string method `word.lower()`, and use a dictionary to maintain a set of counters, one per word. Now we can inspect the contents of the dictionary to get counts for particular words:

```
>>> count['scotland']
12
>>> count['the']
692
>>>
```

2.6.5 Exercises

1. ☼ Using the Python interpreter in interactive mode, experiment with the examples in this section. Create a dictionary `d`, and add some entries. What happens if you try to access a non-existent entry, e.g. `d['xyz']`?

2. ☼ Try deleting an element from a dictionary, using the syntax `del d['abc']`. Check that the item was deleted.

3. ☼ Create a dictionary `e`, to represent a single lexical entry for some word of your choice. Define keys like `headword`, `part-of-speech`, `sense`, and `example`, and assign them suitable values.

4. ☼ Create two dictionaries, `d1` and `d2`, and add some entries to each. Now issue the command `d1.update(d2)`. What did this do? What might it be useful for?

5. ◑ Write a program that takes a sentence expressed as a single string, splits it and counts up the words. Get it to print out each word and the word's frequency, one per line, in alphabetical order.

2.7 Defining Functions

It often happens that part of a program needs to be used several times over. For example, suppose we were writing a program that needed to be able to form the plural of a singular noun, and that this needed to be done at various places during the program. Rather than repeating the same code several times over, it is more efficient (and reliable) to localize this work inside a **function**. A function is a programming construct which can be called with one or more inputs, and which returns an output. We define a function using the keyword `def` followed by the function name and any input parameters, followed by a colon; this in turn is followed by the body of the function. We use the keyword `return` to indicate the value that is produced as output by the function. The best way to convey this is with an example. Our function `plural()` in Listing 2.1 takes a singular noun as input, and generates a plural form as output.

(There is much more to be said about ways of defining in functions, but we will defer this until Section 6.4.)

2.8 Regular Expressions

For a moment, imagine that you are editing a large text, and you have strong dislike of repeated occurrences of the word *very*. How could you find all such cases in the text? To be concrete, let's suppose that the variable `str` is bound to the text shown below:

```
>>> str = """Google Analytics is very very very nice (now)
... By Jason Hoffman 18 August 06
... Google Analytics, the result of Google's acquisition of the San
... Diego-based Urchin Software Corporation, really really opened it's
... doors to the world a couple of days ago, and it allows you to
... track up to 10 sites within a single google account.
... """
>>>
```

Listing 1

```
def plural(word):
    if word[-1] == 'y':
        return word[:-1] + 'ies'
    elif word[-1] in 'sx':
        return word + 'es'
    elif word[-2:] in ['sh', 'ch']:
        return word + 'es'
    elif word[-2:] == 'an':
        return word[:-2] + 'en'
    return word + 's'

>>> plural('fairy')
'fairies'
>>> plural('woman')
'women'
```

The triple quotes `"""` are useful here, since they allow us to break a string across lines.

One approach to our task would be to convert the string into a list, and look for adjacent items which are both equal to the string `'very'`. We use the `range(n)` function in this example to create a list of consecutive integers from 0 up to, but not including, `n`:

```
>>> text = str.split(' ')
>>> for n in range(len(text)):
...     if text[n] == 'very' and text[n+1] == 'very':
...                     print n, n+1
...
3 4
4 5
>>>
```

However, such an approach is not very flexible or convenient. In this section, we will present Python's **regular expression** module `re`, which supports powerful search and substitution inside strings. As a gentle introduction, we will start out using a utility function `re_show()` to illustrate how regular expressions match against substrings. `re_show()` takes two arguments, a pattern that it is looking for, and a string in which the pattern might occur.

```
>>> import re
>>> from nltk_lite.utilities import re_show
>>> re_show('very very', str)
Google Analytics is {very very} very nice (now)
...
>>>
```

(We have only displayed first part of `str` that is returned, since the rest is irrelevant for the moment.) As you can see, `re_show` places curly braces around the first occurrence it has found of the string `'very very'`. So an important part of what `re_show` is doing is searching for any substring of `str` which **matches** the pattern in its first argument.

Now we might want to modify the example so that `re_show` highlights cases where there are two *or more* adjacent sequences of `'very'`. To do this, we need to use a **regular expression operator**,

namely '+'. If *s* is a string, then *s*+ means: 'match one or more occurrences of *s*'. Let's first look at the case where *s* is a single character, namely the letter 'o':

```
>>> re_show('o+', str)
G{oo}gle Analytics is very very very nice (n{o}w)
...
>>>
```

'o+' is our first proper regular expression. You can think of it as matching an *infinite set* of strings, namely the set {'o', 'oo', 'ooo', ...}. But we would really like to match against the set which contains strings of least two 'o's; for this, we need the regular expression 'oo+', which matches any string consisting of 'o' followed by one or more occurrences of o.

```
>>> re_show('oo+', str)
G{oo}gle Analytics is very very very nice (now)
...
>>>
```

Let's return to the task of identifying multiple occurrences of 'very'. Some initially plausible candidates won't do what we want. For example, 'very+' would match 'veryyy' (but not 'very very'), since the + scopes over the immediately preceding expression, in this case 'y'. To widen the scope of +, we need to use parentheses, as in '(very)+'. Will this match 'very very'? No, because we've forgotten about the whitespace between the two words; instead, it will match strings like 'veryvery'. However, the following *does* work:

```
>>> re_show('(very\s)+', str)
Google Analytics is {very very very }nice (now)
>>>
```

Characters which are preceded by a \, such as '\s', have a special interpretation inside regular expressions; thus, '\s' matches a whitespace character. We could have used ' ' in our pattern, but '\s' is better practice in general. One reason is that the sense of 'whitespace' we are using is more general than you might have imagined; it includes not just inter-word spaces, but also tabs and newlines. If you try to inspect the variable str, you might initially get a shock:

```
>>> str
"Google Analytics is very very very nice (now)\nBy Jason Hoffman
18 August 06\n\nGoogle
...
>>>
```

You might recall that '\n' is a special character that corresponds to a newline in a string. The following example shows how newline is matched by '\s'.

```
>>> str2 = "I'm very very\nvery happy"
>>> re_show('very\s', str2)
I'm {very }{very
}{very }happy
>>>
```

Python's re.findall(*patt*, *str*) is a useful function which returns a list of all the substrings in *str* that are matched by *patt*. Before illustrating, let's introduce two further special characters, '\d' and '\w': the first will match any digit, and the second will match any alphanumeric character.

```
>>> re.findall('\d\d', str)
['18', '06', '10']
>>> re.findall('\s\w\w\w\s', str)
[' the ', ' the ', ' the ', ' and ', ' you ']
>>>
```

As you will see, the second example matches three-letter words. However, this regular expression is not quite what we want. First, the leading and trailing spaces are extraneous. Second, it will fail to match against strings such as `'the San'`, where two three-letter words are adjacent. To solve this problem, we can use another special character, namely `'\b'`. This is sometimes called a 'zero-width' character; it matches against the empty string, but only at the begining and ends of words:

```
>>> re.findall(r'\b\w\w\w\b', str)
['now', 'the', 'the', 'San', 'the', 'ago', 'and', 'you']
```

> **Note**
>
> This example uses a Python **raw string**: `r'\b\w\w\w\b'`. The specific justification here is that in an ordinary string, `\b` is interpreted as a backspace character. Python will convert it to a backspace in a regular expression unless you use the `r` prefix to create a raw string as shown above. Another use for raw strings is to match strings which include backslashes. Suppose we want to match 'either\or'. In order to create a regular expression, the backslash needs to be escaped, since it is a special character; so we want to pass the pattern `\\` to the regular expression interpreter. But to express this as a Python string literal, each backslash must be escaped again, yielding the string `'\\\\'`. However, with a raw string, this reduces down to `r'\\'`.

Returning to the case of repeated words, we might want to look for cases involving `'very'` or `'really'`, and for this we use the disjunction operator `|`.

```
>>> re_show('((very|really)\s)+', str)
Google Analytics is {very very very }nice (now)
By Jason Hoffman 18 August 06
Google Analytics, the result of Google's acquisition of the San
Diego-based Urchin Software Corporation, {really really }opened it's
doors to the world a couple of days ago, and it allows you to
track up to 10 sites within a single google account.
>>>
```

In addition to the matches just illustrated, the regular expression `'((very|really)\s)+'` will also match cases where the two disjuncts occur with each other, such as the string `'really very really '`.

Let's now look at how to perform substitutions, using the `re.sub()` function. In the first instance we replace all instances of `l` with `s`. Note that this generates a string as output, and doesn't modify the original string. Then we replace any instances of `green` with `red`.

```
>>> sent = "colorless green ideas sleep furiously"
>>> re.sub('l', 's', sent)
'cosorsess green ideas sseep furioussy'
>>> re.sub('green', 'red', sent)
'colorless red ideas sleep furiously'
>>>
```

We can also disjoin individual characters using a square bracket notation. For example, `[aeiou]` matches any of `a`, `e`, `i`, `o`, or `u`, that is, any vowel. The expression `[^aeiou]` matches anything that is *not* a vowel. In the following example, we match sequences consisting of non-vowels followed by vowels.

```
>>> re_show('[^aeiou][aeiou]', sent)
{co}{lo}r{le}ss g{re}en{ i}{de}as s{le}ep {fu}{ri}ously
>>>
```

Using the same regular expression, the function `re.findall()` returns a list of all the substrings in `sent` that are matched:

```
>>> re.findall('[^aeiou][aeiou]', sent)
['co', 'lo', 'le', 're', ' i', 'de', 'le', 'fu', 'ri']
>>>
```

2.8.1 Groupings

Returning briefly to our earlier problem with unwanted whitespace around three-letter words, we note that `re.findall()` behaves slightly differently if we create **groups** in the regular expression using parentheses; it only returns strings which occur within the groups:

```
>>> re.findall('\s(\w\w\w)\s', str)
['the', 'the', 'the', 'and', 'you']
>>>
```

The same device allows us to select only the non-vowel characters which appear before a vowel:

```
>>> re.findall('([^aeiou])[aeiou]', sent)
['c', 'l', 'l', 'r', ' ', 'd', 'l', 'f', 'r']
>>>
```

By delimiting a second group in the regular expression, we can even generate pairs (or **tuples**), which we may then go on and tabulate.

```
>>> re.findall('([^aeiou])([aeiou])', sent)
[('c', 'o'), ('l', 'o'), ('l', 'e'), ('r', 'e'), (' ', 'i'),
    ('d', 'e'), ('l', 'e'), ('f', 'u'), ('r', 'i')]
>>>
```

Our next example also makes use of groups. One further special character is the so-called wildcard element, `'.'`; this has the distinction of matching any single character (except `'\n'`). Given the string `str3`, our task is to pick out login names and email domains:

```
>>> str3 = """
... <hart@vmd.cso.uiuc.edu>
... Final editing was done by Martin Ward <Martin.Ward@uk.ac.durham>
... Michael S. Hart <hart@pobox.com>
... Prepared by David Price, email <ccx074@coventry.ac.uk>"""
```

The task is made much easier by the fact that all the email addresses in the example are delimited by angle brackets, and we can exploit this feature in our regular expression:

```
>>> re.findall(r'<(.+)@(.+)>', str3)
[('hart', 'vmd.cso.uiuc.edu'), ('Martin.Ward', 'uk.ac.durham'),
('hart', 'pobox.com'), ('ccx074', 'coventry.ac.uk')]
>>>
```

Since `'.'` matches any single character, `'.+'` will match any non-empty *string* of characters, including punctuation symbols such as the period.

One question which might occur to you is how do we specify a match against a period? The answer is that we have to place a `'\'` immediately before the `'.'` in order to escape its special interpretation.

```
>>> re.findall(r'(\w+\.)', str3)
['vmd.', 'cso.', 'uiuc.', 'Martin.', 'uk.', 'ac.', 'S.',
'pobox.', 'coventry.', 'ac.']
>>>
```

Now, let's suppose that we wanted to match occurrences of both `'Google'` and `'google'` in our sample text. If you have been following up till now, you would reasonably expect that this regular expression with a disjunction would do the trick: `'(G|g)oogle'`. But look what happens when we try this with `re.findall()`:

```
>>> re.findall('(G|g)oogle', str)
['G', 'G', 'G', 'g']
>>>
```

What is going wrong? We innocently used the parentheses to indicate the scope of the operator `'|'`, but `re.findall()` has interpreted them as marking a group. In order to tell `re.findall()` "don't try to do anything special with these parentheses", we need an extra piece of notation:

```
>>> re.findall('(?:G|g)oogle', str)
['Google', 'Google', 'Google', 'google']
>>>
```

Placing `'?:'` immediately after the opening parenthesis makes it explicit that the parentheses are just being used for scoping.

2.8.2 Practice Makes Perfect

Regular expressions are very flexible and very powerful. However, they often don't do what you expect. For this reason, you are strongly encouraged to try out a variety of tasks using `re_show()` and `re.findall()` in order to develop your intuitions further; the exercises below should help get you started. One tip is to build up a regular expression in small pieces, rather than trying to get it completely right first time.

As you will see, we will be using regular expressions quite frequently in the following chapters, and we will describe further features as we go along.

2.8.3 Exercises

1. ☼ Describe the class of strings matched by the following regular expressions. Note that `'*'` means: match zero or more occurrences of the preceding regular expression.

   a) `[a-zA-Z]+`

   b) `[A-Z][a-z]*`

c) `\d+(\.\d+)?`

d) `([bcdfghjklmnpqrstvwxyz][aeiou][bcdfghjklmnpqrstvwxyz])*`

e) `\w+|[^\w\s]+`

Test your answers using `re_show()`.

2. ☼ Write regular expressions to match the following classes of strings:

   a) A single determiner (assume that *a*, *an*, and *the* are the only determiners).

   b) An arithmetic expression using integers, addition, and multiplication, such as `2*3+8`.

3. ◑ Using `re.findall()`, write a regular expression which will extract pairs of values of the form *`login name`*, *`email domain`* from the following string:

```
>>> str = """
... austen-emma.txt:hart@vmd.cso.uiuc.edu  (internet)  hart@uiucvmd (bitnet)
... austen-emma.txt:Internet (72600.2026@compuserve.com); TEL: (212-254-5093)
... austen-persuasion.txt:Editing by Martin Ward (Martin.Ward@uk.ac.durham)
... blake-songs.txt:Prepared by David Price, email ccx074@coventry.ac.uk
... """
```

4. ◑ Write code to convert text into *hAck3r* again, this time using regular expressions and substitution, where `e` → `3`, `i` → `1`, `o` → `0`, `l` → `|`, `s` → `5`, `.` → `5w33t!`, `ate` → `8`. Normalise the text to lowercase before converting it. Add more substitutions of your own. Now try to map `s` to two different values: `$` for word-initial `s`, and `5` for word-internal `s`.

5. ◑ Write code to read a file and print it in reverse, so that the last line is listed first.

6. ◑ Write code to access a favorite webpage and extract some text from it. For example, access a weather site and extract the forecast top temperature for your town or city today.

7. ★ Read the Wikipedia entry on the *Soundex Algorithm*. Implement this algorithm in Python.

2.9 Summary

- Text is represented in Python using strings, and we type these with single or double quotes: `'Hello'`, `"World"`.
- The characters of a string are accessed using indexes, counting from zero: `'Hello World'[1]` gives the value `e`. The length of a string is found using `len()`.
- Substrings are accessed using slice notation: `'Hello World'[1:5]` gives the value `ello`. If the start index is omitted, the substring begins at the start of the string, similarly for the end index.
- Sequences of words are represented in Python using lists of strings: `['colorless', 'green', 'ideas']`. We can use indexing, slicing and the `len()` function on lists.

- Strings can be split into lists: `'Hello World'.split()` gives `['Hello', 'World']`. Lists can be joined into strings: `string.join(['Hello', 'World'], '/')` gives `'Hello/World'`.
- Lists can be sorted in-place: `words.sort()`. To produce a separate, sorted copy, use: `sorted(words)`.
- We process each item in a string or list using a `for` statement: `for word in phrase`. This must be followed by the colon character and an indented block of code, to be executed each time through the loop.
- We test a condition using an `if` statement: `if len(word) < 5`. This must be followed by the colon character and an indented block of code, to be executed only if the condition is true.
- A dictionary is used to map between arbitrary types of information, such as a string and a number: `freq['cat'] = 12`. We create dictionaries using the brace notation: `pos = {}`, `pos = {'furiously': 'adv', 'ideas': 'n', 'colorless': 'adj'}`.
- [More: regular expressions]

2.10 Further Reading

Guido Van Rossum (2003). *An Introduction to Python*, Network Theory Ltd.

Guido Van Rossum (2003). *The Python Language Reference*, Network Theory Ltd.

Guido van Rossum (2005). *Python Tutorial* http://docs.python.org/tut/tut.html

A.M. Kuchling. *Regular Expression HOWTO*, http://www.amk.ca/python/howto/regex/

Python Documentation http://docs.python.org/

Allen B. Downey, Jeffrey Elkner and Chris Meyers () *How to Think Like a Computer Scientist: Learning with Python* http://www.ibiblio.org/obp/thinkCSpy/

Chapter 3

Words: The Building Blocks of Language

3.1 Introduction

Language can be divided up into pieces of varying sizes, ranging from morphemes to paragraphs. In this chapter we will focus on words, a very important level for much work in NLP. Just what are words, and how should we represent them in a machine? These may seem like trivial questions, but it turns out that there are some important issues involved in defining and representing words.

In the following sections, we will explore the division of text into words; the distinction between types and tokens; sources of text data including files, the web, and linguistic corpora; accessing these sources using Python and NLTK; stemming and normalisation; Wordnet; and a variety of useful programming tasks involving words

3.2 Tokens, Types and Texts

In Chapter 1, we showed how a string could be split into a list of words. Once we have derived a list, the `len()` function will count the number of words for us:

```
>>> sentence = "This is the time -- and this is the record of the time."
>>> words = sentence.split()
>>> len(words)
13
```

This process of segmenting a string of characters into words is known as **tokenization**. Tokenization is a prelude to pretty much everything else we might want to do in NLP, since it tells our processing software what our basic units are. We will discuss tokenization in more detail shortly.

We also pointed out that we could compile a list of the unique vocabulary items in a string by using `set()` to eliminate duplicates:

```
>>> len(set(words))
10
```

So if we ask how many words there are in `sentence`, we get two different answers, depending on whether we count duplicates or not. Clearly we are using different senses of 'word' here. To help distinguish between them, let's introduce two terms: **token** and **type**. A word token is an individual occurrence of a word in a concrete context; it exists in time and space. A word **type** is a more abstract; it's what we're talking about when we say that the three occurrences of `the` in `sentence` are 'the same word'.

Something similar to a type/token distinction is reflected in the following snippet of Python:

```
>>> words[2]
'the'
>>> words[2] == words[8]
True
>>> words[2] is words[8]
False
>>> words[2] is words[2]
True
```

The operator == tests whether two expressions are equal, and in this case, it is testing for string-identity. This is the notion of identity that was assumed by our use of `set()` above. By contrast, the `is` operator tests whether two objects are stored in the same location of memory, and is therefore analogous to token-identity.

In effect, when we used `split()` above to turn a string into a list of words, our tokenization method was to say that any strings which are delimited by whitespace count as a word token. But this simple approach doesn't always lead to the results we want. Moreover, string-identity doesn't always give us a useful criterion for assigning tokens to types. We therefore need to address two questions in more detail:

***Tokenization*:** Which substrings of the original text should be treated as word tokens?

***Type definition*:** How do we decide whether two tokens have the same type?

To see the problems with our first stab at defining tokens and types in `sentence`, let's look more closely at what is contained in `set(words)`:

```
>>> set(words)
set(['and', 'this', 'record', 'This', 'of', 'is', '--', 'time.',
'time', 'the']
```

One point to note is that `'time'` and `'time.'` come out as distinct tokens, and of necessity, distinct types, since the trailing period has been bundled up with the rest of the word into a single token. We might also argue that although `'--'` is some kind of token, it isn't really a *word* token. Third, we would probably want to say that `'This'` and `'this'` are not distinct types, since capitalization should be ignored.

If we turn to languages other than English, segmenting words can be even more of a challenge. For example, in Chinese orthography, characters correspond to monosyllabic morphemes. Many morphemes are words in their own right, but words contain more than one morpheme. However, there is no visual representation of word boundaries in Chinese text. For example, consider the following three-character string: 爱 国 人 (in pinyin plus tones: ai4 'love' (verb), guo3 'country', ren2 'person'). This could either be segmented as [爱 国] 人 — 'country-loving person' or as 爱 [国 人] — 'love country-person'.

The terms *token* and *type* can also be applied to other linguistic entities. For example, a **sentence token** is an individual occurrence of a sentence; but a **sentence type** is an abstract sentence, without context. If I say the same sentence twice, I have uttered two sentence tokens but only used one sentence type. When the kind of token or type is obvious from context, we will simply use the terms token and type.

To summarize, although the type/token distinction is a useful one, we cannot just say that two word tokens have the same type if they are the same string of characters — we need to take into consideration

a number of other factors in determining what counts as the same word. Moreover, we also need to be more careful in how we identify tokens in the first place.

Up till now, we have relied on getting our source texts by defining a string in a fragment of Python code. However, this is an impractical approach for all but the simplest of texts, and makes it hard to present realistic examples. So how do we get larger chunks of text into our programs? In the rest of this section, we will see how to extract text from files, from the web, and from the corpora distributed with NLTK.

3.2.1 Extracting text from files

It is easy to access local files in Python. As an exercise, create a file called `corpus.txt` using a text editor, and enter the following text:

```
Hello World!
This is a test file.
```

Be sure to save the file as plain text. You also need to make sure that you have saved the file in the same directory or folder in which you are running the Python interactive interpreter.

> **Note**
>
> If you are using IDLE, you can easily create this file by selecting the *New Window* command in the *File* menu, typing in the required text into this window, and then saving the file as `corpus.txt` in the first directory that IDLE offers in the pop-up dialogue box.

The next step is to **open** a file using the built-in function `open()`, which takes two arguments, the name of the file, here `corpus.txt`, and the mode to open the file with (`'r'` means to open the file for reading, and `'U'` stands for "Universal", which lets us ignore the different conventions used for marking newlines).

```
>>> f = open('corpus.txt', 'rU')
```

> **Note**
>
> If the interpreter cannot find your file, it will give an error like this:
>
> ```
> >>> f = open('corpus.txt', 'rU')
> Traceback (most recent call last):
>     File "<pyshell#7>", line 1, in -toplevel-
>     f = open('foo.txt', 'rU')
> IOError: [Errno 2] No such file or directory: 'corpus.txt'
> ```
>
> To check that the file that you are trying to open is really in the right directory, use IDLE's *Open* command in the *File* menu; this will display a list of all the files in the directory where IDLE is running. An alternative is to examine the current directory from within Python:

```
>>> import os
>>> os.listdir('.')
```

To read the contents of the file we can use lots of different methods. The following uses the read method `read()` on the file object `f`; this reads the entire contents of a file into a string.

```
>>> f.read()
'Hello World!\nThis is a test file.\n'
```

You will recall that the strange `'\n'` character on the end of the string is a **newline** character; this is equivalent to pressing *Enter* on a keyboard and starting a new line. .. There is also a `'\t'` character for representing tab. Note that we can open and read a file in one step:

```
>>> text = open('corpus.txt', 'rU').read()
```

We can also read a file one line at a time using the `for` loop construct:

```
>>> f = open('corpus.txt', 'rU')
>>> for line in f:
...     print line[:-1]
Hello world!
This is a test file.
```

Here we use the slice `[:-1]` to remove the newline character at the end of the input line.

3.2.2 Extracting text from the Web

To read in a web page, we use `urlopen()`:

```
>>> from urllib import urlopen
>>> page = urlopen("http://news.bbc.co.uk/").read()
>>> print page[:60]
<!doctype html public "-//W3C//DTD HTML 4.0 Transitional//EN"
```

Web pages are usually in HTML format. To extract the plain text, we can strip out the HTML markup, that is remove all material enclosed in angle brackets. Let's digress briefly to consider how to carry out this task using regular expressions. Our first attempt might look as follows:

```
>>> line = '<title>BBC NEWS | News Front Page</title>'
>>> import re
>>> new = re.sub(r'<.*>', '', line)
```

So the regular expression `'<.*>'` is intended to match a pair of left and right angle brackets, with a string of any characters intervening. However, look at what the result is:

```
>>> new
''
```

What has happened here? The problem is two-fold. First, as already noted, the wildcard `'.'` matches any character other than `'\n'`, so in particular it will match `'>'` and `'<'`. Second, the `'*'` operator is 'greedy', in the sense that it matches as many characters as it can. In the example we just looked at, therefore, `'.*'` will return not the shortest match, namely `'title'`, but the longest match, `'title>BBC NEWS | News Front Page</title'`.

In order to get the results we want, we need to think about the task in a slightly different way. Our assumption is that after we have encountered a `'<'`, any character can occur within the tag except a `'>'`; once we find the latter, we know the tag is closed. Now, we have already seen how to match everything but α, for some character α; we use a negated range expression. In this case, the expression we need is `'[^<]'`: match everything except `'<'`. This range expression is then quantified with the `'*'` operator. In our revised example below, we use the improved regular expression, and we also normalise whitespace, replacing any sequence of one or more spaces, tabs or newlines (these are all matched by `'\s+'`) with a single space character.

```
>>> import re
>>> page = re.sub('<[^>]*>', '', page)
>>> page = re.sub('\s+', ' ', page)
>>> print page[:60]
 BBC NEWS | News Front Page News Sport Weather World Service
```

You will probably find it useful to borrow the structure of this code snippet for future tasks involving regular expressions: each time through a series of substitutions, the result of operating on `page` gets assigned as the new value of `page`. This approach allows us to decompose the transformations we need into a series of simple regular expression substitutions, each of which can be tested and debugged on its own.

3.2.3 Extracting text from NLTK Corpora

NLTK is distributed with several corpora and corpus samples and many are supported by the `corpora` package. Here we import `gutenberg`, a selection of texts from the Project Gutenberg electronic text archive, and list the items it contains:

```
>>> from nltk_lite.corpora import gutenberg
>>> gutenberg.items
['austen-emma', 'austen-persuasion', 'austen-sense', 'bible-kjv',
'blake-poems', 'blake-songs', 'chesterton-ball', 'chesterton-brown',
'chesterton-thursday', 'milton-paradise', 'shakespeare-caesar',
'shakespeare-hamlet', 'shakespeare-macbeth', 'whitman-leaves']
```

Next we iterate over the text content to find the number of word tokens:

```
>>> count = 0
>>> for word in gutenberg.raw('whitman-leaves'):
...     count += 1
>>> print count
154873
```

NLTK also includes the Brown Corpus, the first million-word, part-of-speech tagged electronic corpus of English, created in 1961 at Brown University. Each of the sections `a` through `r` represents a different genre.

```
>>> from nltk_lite.corpora import brown
>>> brown.items
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'j', 'k', 'l', 'm', 'n', 'p', 'r']
```

We can extract individual sentences (as lists of words) from the corpus using the `extract()` function. This is called below with `0` as an argument, indicating that we want the first sentence of the corpus to be returned; `1` will return the second sentence, and so on. `brown.raw()` is an iterator which gives us the words without their part-of-speech tags.

```
>>> from nltk_lite.corpora import extract
>>> print extract(0, brown.raw())
['The', 'Fulton', 'County', 'Grand', 'Jury', 'said', 'Friday', 'an',
'investigation', 'of', "Atlanta's", 'recent', 'primary', 'election',
'produced', '``', 'no', 'evidence', "''", 'that', 'any', 'irregularities',
'took', 'place', '.']
```

3.2.4 Exercises

1. ☼ Create a small text file, and write a program to read it and print it with a line number at the start of each line.

2. ☼ Use the corpus module to read `austin-persuasion.txt`. How many word tokens does this book have? How many word types?

3. ☼ Use the Brown corpus reader `brown.raw()` to access some sample text in two different genres.

4. ☼ Read in the texts of the *State of the Union* addresses, using the `state_union` corpus reader. Count occurrences of `men`, `women`, and `people` in each document. What has happened to the usage of these words over time?

5. ◑ Write a program to generate a table of token/type ratios, as we saw above. Include the full set of Brown Corpus genres. Use the dictionary `brown.item_name` to find out the genre of each section of the corpus. Which genre has the lowest diversity (greatest number of tokens per type)? Is this what you would have expected?

6. ◑ Read in some text from a corpus, tokenize it, and print the list of all *wh*-word types that occur. (*wh*-words in English are questions used in questions, relative clauses and exclamations: *who*, *which*, *what*, and so on.) Print them in order. Are any words duplicated in this list, because of the presence of case distinctions or punctuation?

7. ◑ Examine the results of processing the URL `http://news.bbc.co.uk/` using the regular expressions suggested above. You will see that there is still a fair amount of non-textual data there, particularly Javascript commands. You may also find that sentence breaks have not been properly preserved. Define further regular expressions which improve the extraction of text from this web page.

8. ◑ Take a copy of the `http://news.bbc.co.uk/` over three different days, say at two-day intervals. This should give you three different files, `bbc1.txt`, `bbc2.txt` and `bbc3.txt`, each corresponding to a different snapshot of world events. Collect the 100 most frequent word tokens for each file. What can you tell from the changes in frequency?

9. ◑ Define a function `ghits()`, which takes a word as its argument, and builds a Google query string of the form `http://www.google.com/search?q=word`. Strip the HTML markup and normalize whitespace. Search for a substring of the form `Results 1 - 10 of about`, followed by some number *n*, and extract *n*. Convert this to an integer and return it.

10. ◑ Try running the various chatbots. How *intelligent* are these programs? Take a look at the program code and see if you can discover how it works. You can find the code online at: `http://nltk.sourceforge.net/lite/nltk_lite/chat/`.

3.3 Tokenization and Normalization

Tokenization, as we saw, is the task of extracting a sequence of elementary tokens that constitute a piece of language data. In our first attempt to carry out this task, we started off with a string

of characters, and used the `split()` method to break the string at whitespace characters. (Recall that 'whitespace' covers not only interword space, but also tabs and newlines.) We pointed out that tokenization based solely on whitespace is too simplistic for most applications. In this section we will take a more sophisticated approach, using regular expression to specify which character sequences should be treated as words. We will also consider important ways to normalize tokens.

3.3.1 Tokenization with Regular Expressions

The function `tokenize.regexp()` takes a text string and a regular expression, and returns the list of substrings that match the regular expression. To define a tokenizer that includes punctuation as separate tokens, we could do the following:

```
>>> from nltk_lite import tokenize
>>> text = '''Hello.  Isn't this fun?'''
>>> pattern = r'\w+|[^\w\s]+'
>>> list(tokenize.regexp(text, pattern))
['Hello', '.', 'Isn', "'", 't', 'this', 'fun', '?']
```

The regular expression in this example will match a sequence consisting of one or more word characters `\w+`. It will also match a sequence consisting of one or more punctuation characters (or non-word, non-space characters `[^\w\s]+`). This is another negated range expression; it matches one or more characters which are not word characters (i.e., not a match for `\w`) and not a whitespace character (i.e., not a match for `\s`). We use the disjunction operator `|` to combine these into a single complex expression `\w+|[^\w\s]+`.

There are a number of ways we might want to improve this regular expression. For example, it currently breaks *$22.50* into four tokens; but we might want it to treat this as a single token. Similarly, we would want to treat *U.S.A.* as a single token. We can deal with these by adding further clauses to the tokenizer's regular expression. For readability we break it up and insert comments, and use the `re.VERBOSE` flag, so that Python knows to strip out the embedded whitespace and comments.

```
>>> import re
>>> text = 'That poster costs $22.40.'
>>> pattern = re.compile(r'''
...     \w+                # sequences of 'word' characters
...   | \$?\d+(\.\d+)?     # currency amounts, e.g. $12.50
...   | [\A\.]+            # abbreviations, e.g. U.S.A.
...   | [^\w\s]+           # sequences of punctuation
... ''', re.VERBOSE)
>>> list(tokenize.regexp(text, pattern))
['That', 'poster', 'costs', '$22.40', '.']
```

It is sometimes more convenient to write a regular expression matching the material that appears *between* tokens, such as whitespace and punctuation. The `tokenize.regexp()` function permits an optional boolean parameter `gaps`; when set to `True` the pattern is matched against the gaps. For example, here is how `tokenize.whitespace()` is defined:

```
>>> list(tokenize.regexp(text, pattern=r'\s+', gaps=True))
['That', 'poster', 'costs', '$22.40.']
```

Of course, we can also invoke the `whitespace()` method directly:

```
>>> text = 'That poster costs $22.40.'
>>> list(tokenize.whitespace(text))
['That', 'poster', 'costs', '$22.40.']
```

3.3.2 Lemmatization and Normalization

Earlier we talked about counting word tokens, and completely ignored the rest of the sentence in which these tokens appeared. Thus, for an example like *I saw the saw*, we would have treated both *saw* tokens as instances of the same type. However, one is a form of the verb *see*, and the other is the name of a cutting instrument. How do we know that these two forms of *saw* are unrelated? One answer is that as speakers of English, we know that these would appear as different entries in a dictionary. Another, more empiricist, answer is that if we looked at a large enough number of texts, it would become clear that the two forms have very different distributions. For example, only the noun *saw* will occur immediately after determiners such as *the*. Distinct words which have the same written form are called **homographs**. We can distinguish homographs with the help of context; often the previous word suffices. We will explore this idea of context briefly, before addressing the main topic of this section.

A **bigram** is simply a pair of words. For example, in the sentence *She sells sea shells by the sea shore*, the bigrams are *She sells*, *sells sea*, *sea shells*, *shells by*, *by the*, *the sea*, *sea shore*.

As a first approximation to discovering the distribution of a word, we can look at all the bigrams it occurs in. Let's consider all bigrams from the Brown Corpus which have the word *often* as first element. Here is a small selection, ordered by their counts:

```
often ,            16
often a            10
often in           8
often than         7
often the          7
often been         6
often do           5
often called       4
often appear       3
often were         3
often appeared     2
often are          2
often did          2
often is           2
often appears      1
often call         1
```

In the topmost entry, we see that *often* is frequently followed by a comma. This suggests that *often* is common at the end of phrases. We also see that *often* precedes verbs, presumably as an adverbial modifier. We might infer from this that if we come across *saw* in the context *often* __, then *saw* is being used as a verb.

You will also see that this list includes different grammatical forms of the same verb. We can form separate groups consisting of *appear ~ appears ~ appeared*; *call ~ called*; *do ~ did*; and and *been ~ were ~ are ~ is*. It is common in linguistics to say that two forms such as *appear* and *appeared* belong to a more abstract notion of a word called a **lexeme**; by contast, *appeared* and *called* belong to different lexemes. You can think of a lexeme as corresponding to an entry in a dictionary, and a **lemma** as the headword for that entry. By convention, small capitals are used when referring to a lexeme or lemma: APPEAR.

Although *appeared* and *called* belong to different lexemes, they do have something in common: they are both past tense forms. This is signalled by the segment *-ed*, which we call a morphological **suffix**. We also say that such morphologically complex forms are **inflected**. If we strip off the suffix, we get something called the **stem**, namely *appear* and *call* respectively. While *appeared*, *appears* and

appearing are all morphologically inflected, *appear* lacks any morphological inflection and is therefore termed the **base** form. In English, the base form is conventionally used as the **lemma** for a word.

Our notion of context would be more compact if we could group different forms of the various verbs into their lemmas; then we could study which verb lexemes are typically modified by a particular adverb. **Lemmatization** — the process of mapping grammatical forms into their lemmas — would yield the following picture of the distribution of *often*.

```
often ,             16
often a             10
often be            13
often in            8
often than          7
often the           7
often do            7
often appear        6
often call          5
```

Lemmatization is a rather sophisticated process which requires a mixture of rules for regular inflections and table look-up for irregular morphological patterns. Within NLTK, a simpler approach is offered by the **Porter Stemmer** and the **Lancaster Stemmer**, which strip inflectional suffixes from words, collapsing the different forms of APPEAR and CALL. Given the simple nature of the stemming algorithms, you may not be surprised to learn that this stemmer does not attempt to identify *were* as a form of the lexeme BE.

```
>>> from nltk_lite import stem
>>> stemmer = stem.Porter()
>>> verbs = ['appears', 'appear', 'appeared', 'calling', 'called']
>>> stems = []
>>> for verb in verbs:
...     stemmed_verb = stemmer.stem(verb)
...     if stemmed_verb not in stems:
...         stems.append(stemmed_verb)
>>> stems
['appear', 'call']
```

Lemmatization and stemming can be regarded as special cases of **normalization**. They identify a canonical representative for a group of related word forms. By its nature, normalization collapses distinctions. An example is case normalization, where all variants are mapped into a single format. What counts as the normalized form will vary according to context. Often, we convert everything into lower case, so that words which were capitalized by virtue of being sentence-initial are treated the same as those which occur elsewhere in the sentence. The Python string method `lower()` will accomplish this for us:

```
>>> str = 'This is THE time'
>>> str.lower()
'this is the time'
```

We need to be careful, however; case normalization will also collapse the *New* of *New York* with the *new* of *my new car*.

A final issue for normalization is the presence of contractions, such as *didn't*. If we are analyzing the meaning of a sentence, it would probably be more useful to normalize this form to two separate forms: *did* and *not*.

3.3.3 Aside: List Comprehensions

Lemmatization and normalization involve applying the same operation to each word token in a text. **List comprehensions** are a convenient Python construct for doing this. Here we lowercase each word:

```
>>> sent = ['The', 'dog', 'gave', 'John', 'the', 'newspaper']
>>> [word.lower() for word in sent]
['the', 'dog', 'gave', 'john', 'the', 'newspaper']
```

Here we rewrite the loop for identifying verb stems, from the previous section:

```
>>> [stemmer.stem(verb) for verb in verbs]
['appear', 'appear', 'appear', 'call', 'call']
>>> set(stemmer.stem(verb) for verb in verbs)
set(['call', 'appear'])
```

This syntax might be reminiscent of the notation used for building sets, e.g. $\{(x,y) \mid x^2 + y^2 = 1\}$. Just as this set definition incorporates a constraint, list comprehensions can constrain the items they include. In the next example we remove all determiners from a list of words:

```
>>> def is_lexical(word):
...     return word.lower() not in ('a', 'an', 'the', 'that', 'to')
>>> [word for word in sent if is_lexical(word)]
['dog', 'gave', 'John', 'newspaper']
```

Now we can combine the two ideas, to pull out the content words and normalize them.

```
>>> [word.lower() for word in sent if is_lexical(word)]
['dog', 'gave', 'john', 'newspaper']
```

List comprehensions can build nested structures too. For example, the following code builds a list of tuples, where each tuple consists of a word and its length.

```
>>> sent = extract(0, brown.raw())
>>> [(x, stemmer.stem(x).lower()) for x in sent]
[('The', 'the'), ('Fulton', 'fulton'), ('County', 'counti'),
('Grand', 'grand'), ('Jury', 'juri'), ('said', 'said'), ('Friday', 'friday'),
('an', 'an'), ('investigation', 'investig'), ('of', 'of'),
("Atlanta's", "atlanta'"), ('recent', 'recent'), ('primary', 'primari'),
('election', 'elect'), ('produced', 'produc'), ('``', '``'), ('no', 'no'),
('evidence', 'evid'), ("''", "''"), ('that', 'that'), ('any', 'ani'),
('irregularities', 'irregular'), ('took', 'took'), ('place', 'place'), ('.', '.')]
```

3.3.4 Exercises

1. ☼ **Regular expression tokenizers:** Save some text into a file `corpus.txt`. Define a function `load(f)` that reads from the file named in its sole argument, and returns a string containing the text of the file.

    a) Use `tokenize.regexp()` to create a tokenizer which tokenizes the various kinds of punctuation in this text. Use a single regular expression, with inline comments using the `re.VERBOSE` flag.

b) Use `tokenize.regexp()` to create a tokenizer which tokenizes the following kinds of expression: monetary amounts; dates; names of people and companies.

2. ☼ Rewrite the following loop as a list comprehension:

```
>>> sent = ['The', 'dog', 'gave', 'John', 'the', 'newspaper']
>>> result = []
>>> for word in sent:
...     word_len = (word, len(word))
...     result.append(word_len)
>>> result
[('The', 3), ('dog', 3), ('gave', 4), ('John', 4), ('the', 3), ('newspaper', 9)]
```

3. ◑ Use the Porter Stemmer to normalize some tokenized text, calling the stemmer on each word.

4. ◑ Readability measures are used to score the reading difficulty of a text, for the purposes of selecting texts of appropriate difficulty for language learners. For example, the Automated Readability Index (ARI) of a text is defined to be: `4.71 *` μ_w `+ 0.5 *` μ_s `- 21.43`, where μ_w is the mean word length (in letters), and where μ_s is the mean sentence length (in words). With the help of your word and sentence tokenizers, compute the ARI scores for a collection of texts.

5. ★ Rewrite the following nested loop as a nested list comprehension:

```
>>> words = ['attribution', 'confabulation', 'elocution',
...          'sequoia', 'tenacious', 'unidirectional']
>>> vsequences = set()
>>> for word in words:
...     vowels = []
...     for char in word:
...         if char in 'aeiou':
...             vowels.append(char)
...     vsequences.add(vowels)
>>> sorted(vsequences)
['aiuio', 'eaiou', 'eouio', 'euoia', 'oauaio', 'uiieioa']
```

1. ★ **Sentence tokenizers:** Develop a sentence tokenizer. Test it on the Brown Corpus, which has been grouped into sentences.

3.4 Lexical Resources (INCOMPLETE)

[This section will contain a discussion of lexical resources, focusing on Wordnet, but also including the `cmudict` and `timit` corpus readers.]

3.4.1 Pronunciation Dictionary

Here we access the pronunciation of words...

```
>>> from nltk_lite.corpora import cmudict
>>> from string import join
>>> for word, num, pron in cmudict.raw():
...     if pron[-4:] == ('N', 'IH0', 'K', 'S'):
...         print word.lower(),
atlantic's audiotronics avionics beatniks calisthenics centronics
chetniks clinic's clinics conics cynics diasonics dominic's
ebonics electronics electronics' endotronics endotronics' enix
environics ethnics eugenics fibronics flextronics harmonics
hispanics histrionics identics ionics kibbutzniks lasersonics
lumonics mannix mechanics mechanics' microelectronics minix minnix
mnemonics mnemonics molonicks mullenix mullenix mullinix mulnix
munich's nucleonics onyx panic's panics penix pennix personics
phenix philharmonic's phoenix phonics photronics pinnix
plantronics pyrotechnics refuseniks resnick's respironics sconnix
siliconix skolniks sonics sputniks technics tectonics tektronix
telectronics telephonics tonics unix vinick's vinnick's vitronics
```

3.4.2 WordNet Semantic Network

> **Note**
>
> Before using Wordnet it must be installed on your machine, along with NLTK version 0.8 Please see the instructions on the NLTK website. Help on the `wordnet` interface is available using `help(wordnet)`.

Consider the following sentence:

(9) Benz is credited with the invention of the motorcar.

If we replace *motorcar* in (9) by *automobile*, the meaning of the sentence stays pretty much the same:

(10) Benz is credited with the invention of the automobile.

Since everything else in the sentence has remained unchanged, we can conclude that the words *motorcar* and *automobile* have the same meaning. More technically, we say that they are **synonyms**.

Wordnet is a semantically-oriented dictionary which will allow us to find the set of synonyms — or **synset** — for any word. However, in order to look up the senses of a word, we need to pick a part of speech for the word. Wordnet contains four dictionaries: `N` (nouns), `V` (verbs), `ADJ` (adjectives), and `ADV` (adverbs). To simplify our discussion, we will focus on the `N` dictionary here. Let's look up *motorcar* in the `N` dictionary.

```
>>> from nltk_lite.wordnet import *
>>> car = N['motorcar']
>>> car
motorcar (noun)
```

The variable `car` is now bound to a `Word` object. Words will often have more than sense, where each sense is represented by a synset. However, *motorcar* only has one sense in Wordnet, as we can discover by checking the length of `car`. We can then find the synset (a set of lemmas), the words it contains, and a gloss.

```
>>> len(car)
1
>>> car[0]
{noun: car, auto, automobile, machine, motorcar}
>>> [word for word in car[0]]
['car', 'auto', 'automobile', 'machine', 'motorcar']
>>> car[0].gloss
'a motor vehicle with four wheels; usually propelled by an
internal combustion engine;
"he needs a car to get to work"'
```

The `wordnet` module also defines `Synsets`. Let's look at a word which is **polysemous**; that is, which has multiple synsets:

```
>>> poly = N['pupil']
>>> for synset in poly:
...     print synset
{noun: student, pupil, educatee}
{noun: schoolchild, school-age child, pupil}
{noun: pupil}
>>> poly[2].gloss
'the contractile aperture in the center of the iris of the eye;
resembles a large black dot'
```

We can think of synsets as being concrete manifestations of the concepts that are **lexicalized** by words in a particular language. In principle, we can postulate concepts that have no lexical realization in English. Wordnet builds on a tradition which claims that concepts are linked together in a hierarchy. Some concepts are very general, such as *Entity*, *State*, *Event* — these are called **unique beginners** in Wordnet. Others, such as *gas guzzler* and *hatchback*, are much more specific. A small portion of a concept hierarchy is illustrated in Figure 3.1. The edges between nodes indicate the hypernym/hyponym relation; the dotted line at the top is intended to indicate that *artefact* is non-immediate hypernym of *motorcar*.

Wordnet has been designed to make it easy to navigate between concepts. For example, given a concept like *motorcar*, we can look at the concepts which are more specific; these are usually called **hyponyms**. Here is one way to carry out this navigation:

```
>>> for concept in car[0][HYPONYM][:10]:
...         print concept
{noun: ambulance}
{noun: beach wagon, station wagon, wagon, estate car, beach waggon, station waggon, waggon}
{noun: bus, jalopy, heap}
{noun: cab, hack, taxi, taxicab}
{noun: compact, compact car}
{noun: convertible}
{noun: coupe}
{noun: cruiser, police cruiser, patrol car, police car, prowl car, squad car}
{noun: electric, electric automobile, electric car}
{noun: gas guzzler}
```

We can also move up the hierarchy, by looking at broader concepts than *motorcar*. The `wordnet` package offers a shortcut for finding the immediate **hypernym** of a concept:

```
>>> car[0][HYPERNYM]
[{noun: motor vehicle, automotive vehicle}]
```

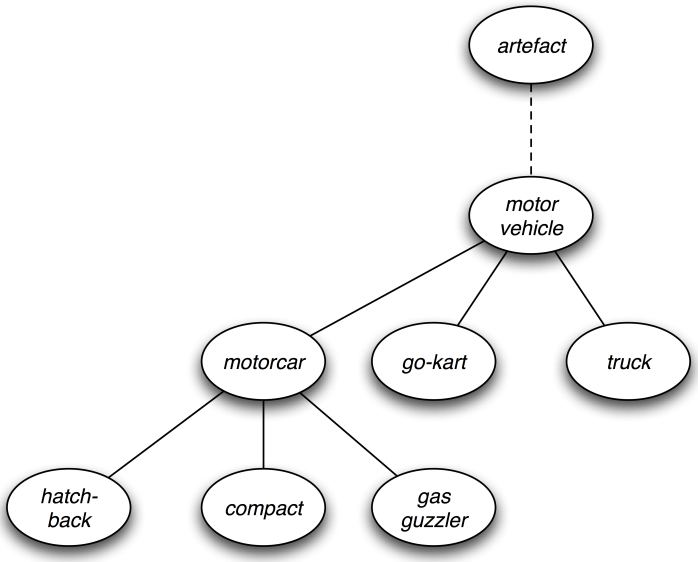


Figure 3.1: Fragment of Wordnet Concept Hierarchy

Of course, we can also look for the hypernyms of hypernyms; from any synset we can trace (multiple) paths back to a unique beginner. Synsets have a method `tree()` which produces a nested list structure.

```
>>> from pprint import pprint
>>> pprint(N['car'][0].tree(HYPERNYM))
[{noun: car, auto, automobile, machine, motorcar},
 [{noun: motor_vehicle, automotive_vehicle},
  [{noun: self-propelled_vehicle},
   [{noun: wheeled_vehicle},
    [{noun: vehicle},
     [{noun: conveyance, transport},
      [{noun: instrumentality, instrumentation},
       [{noun: artifact, artefact},
        [{noun: whole, unit},
         [{noun: object, physical_object},
          [{noun: physical_entity}, [{noun: entity}]]]]]]]],
   [{noun: container},
    [{noun: instrumentality, instrumentation},
     [{noun: artifact, artefact},
      [{noun: whole, unit},
       [{noun: object, physical_object},
        [{noun: physical_entity}, [{noun: entity}]]]]]]]]]
```

A related method `closure()` produces a flat version of this structure, with any repeats eliminated. Both of these functions take an optional `depth` argument which permit us to limit the number of steps to take, which is important when using unbounded relations like `SIMILAR`. Table 3.1 lists the most important lexical relations supported by Wordnet. See `dir(wordnet)` for a full list.

Hypernym	more general	*animal* is a hypernym of *dog*

Hyponym	more specific	*dog* is a hypernym of *animal*
Meronym	part of	*door* is a meronym of *house*
Holonym	has part	*house* is a holonym of *door*
Synonym	similar meaning	*car* is a synonym of *automobile*
Antonym	opposite meaning	*like* is an antonym of *dislike*
Entailment	necessary action	*step* is an entailment of *walk*

Table 3.1: Major WordNet Lexical Relations

Recall that we can iterate over the words of a synset, with `for word in synset`. We can also test if a word is in a dictionary, e.g. `of word in V`. Let's put these together to find "animal words" that are used as verbs. Since there are a lot of these, we will cut this off at depth 4.

```
>>> animals = N['animal'][0].closure(HYPONYM, depth=4)
>>> [word for synset in animals for word in synset if word in V]
['pet', 'stunt', 'prey', 'quarry', 'game', 'mate', 'head', 'dam',
 'sire', 'steer', 'orphan', 'spat', 'sponge', 'worm', 'grub', 'baby',
 'pup', 'whelp', 'cub', 'kit', 'kitten', 'foal', 'lamb', 'fawn',
 'bird', 'grouse', 'stud', 'hog', 'fish', 'cock', 'parrot', 'frog',
 'beetle', 'bug', 'bug', 'queen', 'leech', 'snail', 'slug', 'clam',
 'cockle', 'oyster', 'scallop', 'scollop', 'escallop', 'quail']
```

3.4.3 WordNet Similarity

It is often useful to be able to tell whether two lexical concepts are **semantically related**. For example, in order to check whether a particular instance of the word *bank* means *financial institution*, we can count the number of nearby words that are semantically related to this sense. Using WordNet, we can investigate whether semantic relatedness can be expressed in terms of the graph structure of the concept hierarchy. More specifically, we would expect that the semantic relatedness of two concepts correlates with the length of the path between them. The `wordnet` package includes a variety of measures which incorporate this basic insight. For example, `path_similarity` assigns a score in the range 0–1, based on the shortest path that connects the concepts in the hypernym hierarchy (-1 is returned in those cases where a path cannot be found). A score of 1 represents identity, i.e., comparing a sense with itself will return 1.

```
>>> from nltk_lite.wordnet import *
>>> N['poodle'][0].path_similarity(N['dalmatian'][1])
0.33333333333333331
>>> N['dog'][0].path_similarity(N['cat'][0])
0.20000000000000001
>>> V['run'][0].path_similarity(V['walk'][0])
0.25
>>> V['run'][0].path_similarity(V['think'][0])
-1
```

Several other similarity measures are provided in `nltk_lite.wordnet`: Leacock-Chodorow, Wu-Palmer, Resnik, Jiang-Conrath, and Lin. For a detailed comparison of various measures, see [Budanitsky and Hirst, 2006].

3.4.4 Exercises

1. ☼ Familiarize yourself with the Wordnet interface, by reading the documentation available via `help(wordnet)`.

2. ☼ Investigate the holonym / meronym pointers for some nouns. Note that there are three kinds (member, part, substance), so access is more specific, e.g., `MEMBER_MERONYM`, `SUBSTANCE_HOLONYM`.

3. ◑ Write a program to score the similarity of two nouns as the depth of their first common hypernym.

4. ★ Use one of the predefined similarity measures to score the similarity of each of the following pairs of words. Rank the pairs in order of decreasing similarity. How close is your ranking to the order given here? (Note that this order was established experimentally by [Miller and Charles, 1998].)

:: car-automobile, gem-jewel, journey-voyage, boy-lad, coast-shore, asylum-madhouse, magician-wizard, midday-noon, furnace-stove, food-fruit, bird-cock, bird-crane, tool-implement, brother-monk, lad-brother, crane-implement, journey-car, monk-oracle, cemetery-woodland, food-rooster, coast-hill, forest-graveyard, shore-woodland, monk-slave, coast-forest, lad-wizard, chord-smile, glass-magician, rooster-voyage, noon-string.

3.5 Simple Statistics with Tokens

3.5.1 Example: Stylistics

So far, we've seen how to count the number of tokens or types in a document. But it's much more interesting to look at *which* tokens or types appear in a document. We can use a Python dictionary to count the number of occurrences of each word type in a document:

```
>>> counts = {}
>>> for word in text.split():
...     if word not in counts:
...         counts[word] = 0
...     counts[word] += 1
```

The first statement, `counts = {}`, initializes the dictionary, while the next four lines successively add entries to it and increment the count each time we encounter a new token of a given type. To view the contents of the dictionary, we can iterate over its keys and print each entry (here just for the first 10 entries):

```
>>> for word in sorted(counts)[:10]:
...     print counts[word], word
1 $1.1
2 $130
1 $36
1 $45
1 $490
1 $5
1 $62.625,
```

```
1 $620
1 $63
2 $7
```

We can also print the number of times that a specific word we're interested in appeared:

```
>>> print counts['might']
3
```

Applying this same approach to document collections that are categorized by genre, we can learn something about the patterns of word usage in those genres. For example, Table 3.2 was constructed by counting the number of times various modal words appear in different genres in the Brown Corpus:

Genre	can	could	may	might	must	will
skill and hobbies	273	59	130	22	83	259
humor	17	33	8	8	9	13
fiction: science	16	49	4	12	8	16
press: reportage	94	86	66	36	50	387
fiction: romance	79	195	11	51	46	43
religion	84	59	79	12	54	64

Table 3.2: Use of Modals in Brown Corpus, by Genre

Observe that the most frequent modal in the reportage genre is *will*, suggesting a focus on the future, while the most frequent modal in the romance genre is *could*, suggesting a focus on possibilities.

We can also measure the lexical diversity of a genre, by calculating the ratio of word types and word tokens, as shown in Table 3.3. (Genres with lower diversity have a higher number of tokens per type.)

Genre	Token Count	Type Count	Ratio
skill and hobbies	82345	11935	6.9
humor	21695	5017	4.3
fiction: science	14470	3233	4.5
press: reportage	100554	14394	7.0
fiction: romance	70022	8452	8.3
religion	39399	6373	6.2

Table 3.3: Word Types and Tokens in Brown Corpus, by Genre

We can carry out a variety of interesting explorations simply by counting words. In fact, the field of *Corpus Linguistics* focuses almost exclusively on creating and interpreting such tables of word counts. So far, our method for identifying word tokens has been a little primitive, and we have not been able to separate punctuation from the words. We will take up this issue in the next section.

3.5.2 Example: Lexical Dispersion

Word tokens vary in their distribution throughout a text. We can visualize word distributions, to get an overall sense of topics and topic shifts. For example, consider the pattern of mention of the main characters in Jane Austen's *Sense and Sensibility*: Elinor, Marianne, Edward and Willoughby. The following plot contains four rows, one for each name, in the order just given. Each row contains a series of lines, drawn to indicate the position of each token.

Figure 3.2: Lexical Dispersion

As you can see, *Elinor* and *Marianne* appear rather uniformly throughout the text, while *Edward* and *Willoughby* tend to appear separately. Here is the program that generated the above plot. [NB. Requires NLTK-Lite 0.6.7].

```
>>> from nltk_lite.corpora import gutenberg
>>> from nltk_lite.draw import dispersion
>>> words = ['Elinor', 'Marianne', 'Edward', 'Willoughby']
>>> dispersion.plot(gutenberg.raw('austen-sense'), words)
```

3.5.3 Frequency Distributions

We can do more sophisticated counting using *frequency distributions*. Abstractly, a frequency distribution is a record of the number of times each *outcome* of an *experiment* has occurred. For instance, a frequency distribution could be used to record the frequency of each word in a document (where the "experiment" is examining a word, and the "outcome" is the word's type). Frequency distributions are generally created by repeatedly running an experiment, and incrementing the count for a sample every time it is an outcome of the experiment. The following program produces a frequency distribution that records how often each word type occurs in a text. It increments a separate counter for each word, and prints the most frequently occurring word:

```
>>> from nltk_lite.probability import FreqDist
>>> from nltk_lite.corpora import genesis
>>> fd = FreqDist()
>>> for token in genesis.raw():
...     fd.inc(token)
>>> fd.max()
'the'
```

Once we construct a frequency distribution that records the outcomes of an experiment, we can use it to examine a number of interesting properties of the experiment. Some of these properties are summarized in Table 3.4.

Name	Sample	Description
Count	`fd.count('the')`	number of times a given sample occurred
Frequency	`fd.freq('the')`	frequency of a given sample
N	`fd.N()`	number of samples
Samples	`fd.samples()`	list of distinct samples recorded

Name	Sample	Description
Max	`fd.max()`	sample with the greatest number of outcomes

Table 3.4: Frequency Distribution Module

We can also use a `FreqDist` to examine the distribution of word lengths in a corpus. For each word, we find its length, and increment the count for words of this length.

```
>>> def length_dist(text):
...     fd = FreqDist()                          # initialize frequency distribution
...     for token in genesis.raw(text):          # for each token
...         fd.inc(len(token))                   # found a word with this length
...     for i in range(1,15):                    # for each length from 1 to 14
...         print "%2d" % int(100*fd.freq(i)), # print the percentage of words with this le
...     print
```

Now we can call `length_dist` on a text to print the distribution of word lengths. We see that the most frequent word length for the English sample is 3 characters, while the most frequent length for the Finnish sample is 5-6 characters.

```
>>> length_dist('english-kjv')
 2 14 28 21 13  7  5  2  2  0  0  0  0  0
>>> length_dist('finnish')
 0  9  6 10 16 16 12  9  6  3  2  2  1  0
```

3.5.4 Conditional Frequency Distributions

A *condition* specifies the context in which an experiment is performed. Often, we are interested in the effect that conditions have on the outcome for an experiment. A *conditional frequency distribution* is a collection of frequency distributions for the same experiment, run under different conditions. For example, we might want to examine how the distribution of a word's length (the outcome) is affected by the word's initial letter (the condition).

```
>>> from nltk_lite.corpora import genesis
>>> from nltk_lite.probability import ConditionalFreqDist
>>> cfdist = ConditionalFreqDist()
>>> for text in genesis.items:
...     for word in genesis.raw(text):
...         cfdist[word[0]].inc(len(word))
```

To plot the results, we construct a list of points, where the x coordinate is the word length, and the y coordinate is the frequency with which that word length is used:

```
>>> for cond in cfdist.conditions():
...     wordlens = cfdist[cond].samples()
...     wordlens.sort()
...     points = [(i, cfdist[cond].freq(i)) for i in wordlens]
```

We can plot these points using the `Plot` function defined in `nltk_lite.draw.plot`, as follows:

```
>>> Plot(points).mainloop()
```

3.5.5 Predicting the Next Word

Conditional frequency distributions are often used for prediction. *Prediction* is the problem of deciding a likely outcome for a given run of an experiment. The decision of which outcome to predict is usually based on the context in which the experiment is performed. For example, we might try to predict a word's text (outcome), based on the text of the word that it follows (context).

To predict the outcomes of an experiment, we first examine a representative *training corpus*, where the context and outcome for each run of the experiment are known. When presented with a new run of the experiment, we simply choose the outcome that occurred most frequently for the experiment's context.

We can use a `ConditionalFreqDist` to find the most frequent occurrence for each context. First, we record each outcome in the training corpus, using the context that the experiment was run under as the condition. Then, we can access the frequency distribution for a given context with the indexing operator, and use the `max()` method to find the most likely outcome.

We will now use a `ConditionalFreqDist` to predict the most likely next word in a text. To begin, we load a corpus from a text file, and create an empty `ConditionalFreqDist`:

```
>>> from nltk_lite.corpora import genesis
>>> from nltk_lite.probability import ConditionalFreqDist
>>> cfdist = ConditionalFreqDist()
```

We then examine each token in the corpus, and increment the appropriate sample's count. We use the variable `prev` to record the previous word.

```
>>> prev = None
>>> for word in genesis.raw():
...     cfdist[prev].inc(word)
...     prev = word
```

> **Note**
>
> Sometimes the context for an experiment is unavailable, or does not exist. For example, the first token in a text does not follow any word. In these cases, we must decide what context to use. For this example, we use `None` as the context for the first token. Another option would be to discard the first token.

Once we have constructed a conditional frequency distribution for the training corpus, we can use it to find the most likely word for any given context. For example, taking the word *living* as our context, we can inspect all the words that occurred in that context.

```
>>> word = 'living'
>>> cfdist[word].samples()
['creature,', 'substance', 'soul.', 'thing', 'thing,', 'creature']
```

We can set up a simple loop to generate text: we set an initial context, picking the most likely token in that context as our next word, and then using that word as our new context:

```
>>> word = 'living'
>>> for i in range(20):
...     print word,
...     word = cfdist[word].max()
living creature that he said, I will not be a wife of the land
of the land of the land
```

This simple approach to text generation tends to get stuck in loops, as demonstrated by the text generated above. A more advanced approach would be to randomly choose each word, with more frequent words chosen more often.

3.5.6 Exercises

1. ☼ Pick a text, and explore the dispersion of particular words. What does this tell you about the words, or the text?

2. ☼ Use the `Plot` function defined in `nltk_lite.draw.plot` plot word-initial character against word length, as discussed in this section.

3. ◑ Write a program to create a table of word frequencies by genre, like the one given above for modals. Choose your own words and try to find words whose presence (or absence) is typical of a genre. Discuss your findings.

4. ◑ **Zipf's Law**: Let *f(w)* be the frequency of a word *w* in free text. Suppose that all the words of a text are ranked according to their frequency, with the most frequent word first. Zipf's law states that the frequency of a word type is inversely proportional to its rank (i.e. *f.r* = *k*, for some constant *k*). For example, the 50th most common word type should occur three times as frequently as the 150th most common word type.

   a) Write a function to process a large text and plot word frequency against word rank using the `nltk_lite.draw.plot` module. Do you confirm Zipf's law? (Hint: it helps to use logarithmic axes, by including `scale='log'` as a second argument to `Plot()`). What is going on at the extreme ends of the plotted line?

   b) Generate random text, e.g. using `random.choice("abcdefg ")`, taking care to include the space character. You will need to `import random` first. Use the string concatenation operator to accumulate characters into a (very) long string. Then tokenize this string, and generate the Zipf plot as before, and compare the two plots. What do you make of Zipf's Law in the light of this?

5. ◑ **Predicting the next word**: The word prediction program we saw in this chapter quickly gets stuck in a cycle. Modify the program to choose the next word randomly, from a list of the *n* most likely words in the given context. (Hint: store the *n* most likely words in a list `lwords` then randomly choose a word from the list using `random.choice()`.)

   a) Select a particular genre, such as a section of the Brown Corpus, or a genesis translation, or one of the Gutenberg texts. Train your system on this corpus and get it to generate random text. You may have to experiment with different start words. How intelligible is the text? Discuss the strengths and weaknesses of this method of generating random text.

   b) Try the same approach with different genres, and with different amounts of training data. What do you observe?

   c) Now train your system using two distinct genres and experiment with generating text in the hybrid genre. As before, discuss your observations.

6. ◑ **Exploring text genres:** Investigate the table of modal distributions and look for other patterns. Try to explain them in terms of your own impressionistic understanding of the different genres. Can you find other closed classes of words that exhibit significant differences across different genres?

7. ★ **Authorship identification:** Reproduce some of the results of [Zhao and Zobel, 2007].

8. ★ **Gender-specific lexical choice:** Reproduce some of the results of `http://www.clintoneast.com/articles/words.php`

3.6 Conclusion

In this chapter we saw that we can do a variety of interesting language processing tasks that focus solely on words. Tokenization turns out to be far more difficult than expected. Other kinds of tokenization, such as sentence tokenization, are left for the exercises. No single solution works well across-the-board, and we must decide what counts as a token depending on the application domain. We also looked at normalization (including lemmatization) and saw how it collapses distinctions between tokens. In the next chapter we will look at word classes and automatic tagging.

3.7 Further Reading

Chapter 4

Categorizing and Tagging Words

4.1 Introduction

In Chapter 3 we dealt with words in their own right. We saw that some distinctions can be collapsed using normalization, but we did not make any further generalizations. We looked at the distribution of *often*, identifying the words that follow it; we noticed that *often* frequently modifies verbs. We also assumed that you knew that words such as *was*, *called* and *appears* are all verbs, and that you knew that *often* is an adverb. In fact, we take it for granted that most people have a rough idea about how to group words into different categories.

There is a long tradition of classifying words into categories called **parts of speech**. These are sometimes also called word classes or **lexical categories**. Apart from verb and adverb, other familiar examples are **noun**, **preposition**, and **adjective**. One of the notable features of the Brown corpus is that all the words have been **tagged** for their part-of-speech. Now, instead of just looking at the words that immediately follow *often*, we can look at the **part-of-speech tags** (or **POS tags**). Table 4.1 lists the top eight, ordered by frequency, along with explanations of each tag. As we can see, the majority of words following *often* are verbs.

Tag	Freq	Example	Comment
vbn	61	*burnt*, *gone*	verb: past participle
vb	51	*make*, *achieve*	verb: base form
vbd	36	*saw*, *looked*	verb: simple past tense
jj	30	*ambiguous*, *acceptable*	adjective
vbz	24	*sees*, *goes*	verb: third-person singular present
in	18	*by*, *in*	preposition
at	18	*a*, *this*	article
,	16	,	comma

Table 4.1: Part of Speech Tags Following *often* in the Brown Corpus

The process of classifying words into their parts-of-speech and labeling them accordingly is known as **part-of-speech tagging**, **POS-tagging**, or simply **tagging**. The collection of tags used for a particular task is known as a **tag set**. Our emphasis in this chapter is on exploiting tags, and tagging text automatically.

Automatic tagging can bring a number of benefits. We have already seen an example of how to exploit tags in corpus analysis — we get a clear understanding of the distribution of *often* by looking at the tags of adjacent words. Automatic tagging also helps predict the behavior of previously unseen words. For example, if we encounter the word *blogging* we can probably infer that it is a verb, with the root *blog*, and likely to occur after forms of the auxiliary `to be` (e.g. *he was blogging*). Parts of speech are also used in speech synthesis and recognition. For example, `wind/nn`, as in *the wind blew*, is pronounced with a short vowel, whereas `wind/vb`, as in *wind the clock*, is pronounced with a long vowel. Other examples can be found where the stress pattern differs depending on whether the word is a noun or a verb, e.g. *contest*, *insult*, *present*, *protest*, *rebel*, *suspect*. Without knowing the part of speech we cannot be sure of pronouncing the word correctly.

In the next section we will see how to access and explore the Brown Corpus. Following this we will take a more in depth look at the linguistics of word classes. The rest of the chapter will deal with automatic tagging: simple taggers, evaluation, n-gram taggers, and the Brill tagger.

4.2 Getting Started with Tagging

Several large corpora, such as the Brown Corpus and portions of the Wall Street Journal, have been tagged for part-of-speech, and we will be able to process this tagged data. Tagged corpus files typically contain text of the following form (this example is from the Brown Corpus):

```
The/at grand/jj jury/nn commented/vbd on/in a/at number/nn of/in
other/ap topics/nns ,/, among/in them/ppo the/at Atlanta/np and/cc
Fulton/np-tl County/nn-tl purchasing/vbg departments/nns which/wdt it/pps
said/vbd ``/`` are/ber well/ql operated/vbn and/cc follow/vb generally/rb
accepted/vbn practices/nns which/wdt inure/vb to/in the/at best/jjt
interest/nn of/in both/abx governments/nns ''/'' ./.
```

4.2.1 Representing Tags and Reading Tagged Corpora

By convention in NLTK, a tagged token is represented using a Python tuple. A tuple is just like a list, only it cannot be modified. We can access the components of a tuple using indexing:

```
>>> tok = ('fly', 'nn')
>>> tok
('fly', 'nn')
>>> tok[0]
'fly'
>>> tok[1]
'nn'
```

We can create one of these special tuples from the standard string representation of a tagged token, using the function `tag2tuple()`:

```
>>> from nltk_lite.tag import tag2tuple
>>> tag2tuple('fly/nn')
('fly', 'nn')
```

We can construct tagged tokens directly from a string. The first step is to tokenize the string (using `tokenize.whitespace()`) to access the individual `word/tag` strings, and then to convert each of these into a tuple (using `tag2tuple()`). We do this in two ways. The first method, starting at line

①, initializes an empty list `tagged_words`, loops over the `word/tag` tokens, converts them into tuples, appends them to `tagged_words`, and finally displays the result. The second method, on line ②, uses a list comprehension to do the same work in a way that is not only more compact, but also more readable. (List comprehensions were introduced in section 3.3.3).

```
>>> from nltk_lite import tokenize
>>> sent = '''
... The/at grand/jj jury/nn commented/vbd on/in a/at number/nn of/in
... other/ap topics/nns ,/, among/in them/ppo the/at Atlanta/np and/cc
... Fulton/np-tl County/nn-tl purchasing/vbg departments/nns which/wdt it/pps
... said/vbd ``/`` are/ber well/ql operated/vbn and/cc follow/vb generally/rb
... accepted/vbn practices/nns which/wdt inure/vb to/in the/at best/jjt
... interest/nn of/in both/abx governments/nns ''/'' ./.
... '''
>>> tagged_words = []                                          ①
>>> for t in tokenize.whitespace(sent):
...     tagged_words.append(tag2tuple(t))
>>> tagged_words
[('The', 'at'), ('grand', 'jj'), ('jury', 'nn'), ('commented', 'vbd'),
('on', 'in'), ('a', 'at'), ('number', 'nn'), ... ('.', '.')]
>>> [tag2tuple(t) for t in tokenize.whitespace(sent)]    ②
[('The', 'at'), ('grand', 'jj'), ('jury', 'nn'), ('commented', 'vbd'),
('on', 'in'), ('a', 'at'), ('number', 'nn'), ... ('.', '.')]
```

We can also conveniently access tagged corpora directly from Python. The first step is to load the Brown Corpus reader, `brown`. We then use one of its functions, `brown.tagged()` to produce a sequence of sentences, where each sentence is a list of tagged words.

```
>>> from nltk_lite.corpora import brown, extract
>>> extract(6, brown.tagged('a'))
[('The', 'at'), ('grand', 'jj'), ('jury', 'nn'), ('commented', 'vbd'),
('on', 'in'), ('a', 'at'), ('number', 'nn'), ('of', 'in'), ('other', 'ap'),
('topics', 'nns'), (',', ','), ... ('.', '.')]
```

Tagged corpora are available for several other languages. For example, we can access tagged corpora for some Indian languages with `nltk_lite.corpora.indian`. Figure 4.1 shows the output of the demonstration code `indian.demo()`. The tags used with these corpora are documented in the `README` file that comes with this data.

```
Bangla: কুঁড়েঘরগুলি/'NN' আকার/'NN' বাংলার/'NNP' বা/'CC' ভারতের/'NNP' ?/None
নয়/'JJ' ?/None এঁচলের/'NN' প্রচলিত/'JJ' কুঁড়ে/'NN' ঘর/'NN' নয়/'VM' ৷/'SYM'
Hindi: पाकिस्तान/'NNP' की/'PREP' पूर्व/'JJ' प्रधानमंत्री/'NN' बेनजीर/'NNPC' भुट्टो/'NNP'
पर/'PREP' लगे/'VFM' भ्रष्टाचार/'NN' के/'PREP' आरोपों/'NN' के/'PREP' खिलाफ/'PREP' भुट्टो/'NNP'
द्वारा/'PREP' दायर/'NVB' की/'VFM' गई/'VAUX' याचिका/'NN' की/'PREP' सुनवाई/'NN'
मंगलवार/'NN' को/'PREP' वकीलों/'NN' की/'PREP' हड़ताल/'NN' के/'PREP' कारण/'PREP'
स्थगित/'JVB' कर/'VFM' दी/'VAUX' गई/'VAUX' ।/'PUNC'
Marathi: ग्रामीण/'JJ' जिल्हाध्यक्ष/'NN' बाळासाहेब/'NNPC' भोसले/'NNP' यांच्या/'PRP' ?/None
दयक्षतेखाली/'NN' पक्षाची/'NN' आज/'NN' बै?/None क/'NN' झाली/'VM' ./'SYM'
Telugu: ఖజురుల/'NN' నుంచి/'PREP' వచ్చిన/'VJJ' పళ్లాల/'NN' ను/'PREP' సొఖ్ఖా/'NN'
```

Figure 4.1: POS-Tagged Data from Four Indian Languages

Note

Contributions of tagged data for other languages are invited. These will be incorporated into NLTK.

4.2.2 Nouns and Verbs

Linguists recognize several major categories of words in English, such as nouns, verbs, adjectives and determiners. In this section we will discuss the most important categories, namely nouns and verbs.

Nouns generally refer to people, places, things, or concepts, e.g.: *woman, Scotland, book, intelligence*. Nouns can appear after determiners and adjectives, and can be the subject or object of the verb, as shown in Table 4.2.

Word	After a determiner	Subject of the verb
woman	*the* woman who I saw yesterday ...	the woman *sat* down
Scotland	*the* Scotland I remember as a child ...	Scotland *has* five million people
book	*the* book I bought yesterday ...	this book *recounts* the colonization of Australia
intelligence	*the* intelligence displayed by the child ...	Mary's intelligence *impressed* her teachers

Table 4.2: Syntactic Patterns involving some Nouns

Nouns can be classified as **common nouns** and **proper nouns**. Proper nouns identify particular individuals or entities, e.g. *Moses* and *Scotland*. Common nouns are all the rest. Another distinction exists between **count nouns** and **mass nouns**. Count nouns are thought of as distinct entities which can be counted, such as *pig* (e.g. *one pig, two pigs, many pigs*). They cannot occur with the word *much* (i.e. **much pigs*). Mass nouns, on the other hand, are not thought of as distinct entities (e.g. *sand*). They cannot be pluralized, and do not occur with numbers (e.g. **two sands*, **many sands*). However, they can occur with *much* (i.e. *much sand*).

Verbs are words which describe events and actions, e.g. *fall*, *eat* in Table 4.3. In the context of a sentence, verbs express a relation involving the referents of one or more noun phrases.

Word	Simple	With modifiers and adjuncts (italicized)
fall	Rome fell	Dot com stocks *suddenly* fell *like a stone*
eat	Mice eat cheese	John ate the pizza *with gusto*

Table 4.3: Syntactic Patterns involving some Verbs

Verbs can be classified according to the number of arguments (usually noun phrases) that they require. The word *fall* is **intransitive**, requiring exactly one argument (the entity which falls). The word *eat* is **transitive**, requiring two arguments (the eater and the eaten). Other verbs are more complex; for instance *put* requires three arguments, the agent doing the putting, the entity being put somewhere, and a location. We will return to this topic when we come to look at grammars and parsing (see Chapter 7).

In the Brown Corpus, verbs have a range of possible tags, e.g.: `give/vb` (present), `gives/vbz` (present, 3ps), `giving/vbg` (present continuous; gerund) `gave/vbd` (simple past), and `given/vbn` (past participle). We will discuss these tags in more detail in a later section.

4.2.3 Nouns and verbs in tagged corpora

Now that we are able to access tagged corpora, we can write simple simple programs to garner statistics about the tags. In this section we will focus on the nouns and verbs.

What are the 10 most common verbs? We can write a program to find all words tagged with VB, VBZ, VBG, VBD or VBN.

```
>>> from nltk_lite.probability import FreqDist
>>> fd = FreqDist()
>>> for sent in brown.tagged():
...     for word, tag in sent:
...         if tag[:2] == 'vb':
...             fd.inc(word+"/"+tag)
>>> fd.sorted_samples()[:20]
['said/vbd', 'make/vb', 'see/vb', 'get/vb', 'know/vb', 'made/vbn',
'came/vbd', 'go/vb', 'take/vb', 'went/vbd', 'say/vb', 'used/vbn',
'made/vbd', 'United/vbn-tl', 'think/vb', 'took/vbd', 'come/vb',
'knew/vbd', 'find/vb', 'going/vbg']
```

Let's study nouns, and find the most frequent nouns of each noun part-of-speech type. The program in Listing 4.1 finds all tags starting with `nn`, and provides a few example words for each one. Observe that there are many noun tags; the most important of these have `$` for possessive nouns, `s` for plural nouns (since plural nouns typically end in *s*), `p` for proper nouns.

Some tags contain a plus sign; these are compound tags, and are assigned to words that contain two parts normally treated separately. Some tags contain a minus sign; this indicates disjunction [MORE].

4.2.4 The Default Tagger

The simplest possible tagger assigns the same tag to each token. This may seem to be a rather banal step, but it establishes an important baseline for tagger performance. In order to get the best result, we tag each word with the most likely word. (This kind of tagger is known as a **majority class classifier**). What then, is the most frequent tag? We can find out using a simple program:

```
>>> fd = FreqDist()
>>> for sent in brown.tagged('a'):
...     for word, tag in sent:
...         fd.inc(tag)
>>> fd.max()
'nn'
```

Now we can create a tagger, called `default_tagger`, which tags everything as `nn`.

```
>>> from nltk_lite import tag
>>> tokens = tokenize.whitespace('John saw 3 polar bears .')
>>> default_tagger = tag.Default('nn')
>>> list(default_tagger.tag(tokens))
[('John', 'nn'), ('saw', 'nn'), ('3', 'nn'), ('polar', 'nn'),
('bears', 'nn'), ('.', 'nn')]
```

> **Note**
>
> The tokenizer is a *generator* over tokens. We cannot print it directly, but we can convert it to a list for printing, as shown in the above program. Note that we can only use a generator once, but if we save it as a list, the list can be used many times over.

Listing 2 Program to Find the Most Frequent Noun Tags

```
from nltk_lite.probability import ConditionalFreqDist
def findtags(tag_prefix, tagged_text):
    cfd = ConditionalFreqDist()
    for sent in tagged_text:
        for word, tag in sent:
            if tag.startswith(tag_prefix):
                cfd[tag].inc(word)
    tagdict = {}
    for tag in cfd.conditions():
        tagdict[tag] = cfd[tag].sorted_samples()[:5]
    return tagdict

>>> tagdict = findtags('nn', brown.tagged('a'))
>>> for tag in sorted(tagdict):
...     print tag, tagdict[tag]
nn ['year', 'time', 'state', 'week', 'home']
nn$ ["year's", "world's", "state's", "city's", "company's"]
nn$-hl ["Golf's", "Navy's"]
nn$-tl ["President's", "Administration's", "Army's", "Gallery's", "League's"]
nn-hl ['Question', 'Salary', 'business', 'condition', 'cut']
nn-nc ['aya', 'eva', 'ova']
nn-tl ['President', 'House', 'State', 'University', 'City']
nn-tl-hl ['Fort', 'Basin', 'Beat', 'City', 'Commissioner']
nns ['years', 'members', 'people', 'sales', 'men']
nns$ ["children's", "women's", "janitors'", "men's", "builders'"]
nns$-hl ["Dealers'", "Idols'"]
nns$-tl ["Women's", "States'", "Giants'", "Bombers'", "Braves'"]
nns-hl ['$12,500', '$14', '$37', 'A135', 'Arms']
nns-tl ['States', 'Nations', 'Masters', 'Bears', 'Communists']
nns-tl-hl ['Nations']
```

This is a simple algorithm, and it performs poorly when used on its own. On a typical corpus, it will tag only about an eighth of the tokens correctly:

```
>>> tag.accuracy(default_tagger, brown.tagged('a'))
0.13089484257215028
```

Default taggers assign their tag to every single word, even words that have never been encountered before. As it happens, most new words are nouns. Thus, default taggers they help to improve the robustness of a language processing system. We will return to them later, in the context of our discussion of *backoff*.

4.2.5 Exercises

1. ☼ Working with someone else, take turns to pick a word which can be either a noun or a verb (e.g. *contest*); the opponent has to predict which one is likely to be the most frequent in the Brown corpus; check the opponents prediction, and tally the score over several turns.

2. ◑ Write programs to process the Brown Corpus and find answers to the following questions:

    1) Which nouns are more common in their plural form, rather than their singular form? (Only consider regular plurals, formed with the *-s* suffix.)
    2) Which word has the greatest number of distinct tags. What are they, and what do they represent?
    3) List tags in order of decreasing frequency. What do the 20 most frequent tags represent?
    4) Which tags are nouns most commonly found after? What do these tags represent?

3. ◑ Generate some statistics for tagged data to answer the following questions:

    a) What proportion of word types are always assigned the same part-of-speech tag?
    b) How many words are ambiguous, in the sense that they appear with at least two tags?
    c) What percentage of word *occurrences* in the Brown Corpus involve these ambiguous words?

4. ◑ Above we gave an example of the `tag.accuracy()` function. It has two arguments, a tagger and some tagged text, and it works out how accurately the tagger performs on this text. For example, if the supplied tagged text was `[('the', 'dt'), ('dog', 'nn')]` and the tagger produced the output `[('the', 'nn'), ('dog', 'nn')]`, then the accuracy score would be `0.5`. Can you figure out how the `tag.accuracy()` function works?

    a) A tagger takes a list of words as input, and produces a list of tagged words as output. However, `tag.accuracy()` is given correctly tagged text as its input. What must the `tag.accuracy()` function do with this input before performing the tagging?

b) Once the supplied tagger has created newly tagged text, how would `tag.accuracy()` go about comparing it with the original tagged text and computing the accuracy score?

4.3 Looking for Patterns in Words

4.3.1 Some morphology

English nouns can be morphologically complex. For example, words like *books* and *women* are plural. Words with the *-ness* suffix are nouns that have been derived from adjectives, e.g. *happiness* and *illness*. The *-ment* suffix appears on certain nouns derived from verbs, e.g. *government* and *establishment*.

English verbs can also be morphologically complex. For instance, the **present participle** of a verb ends in *-ing*, and expresses the idea of ongoing, incomplete action (e.g. *falling, eating*). The *-ing* suffix also appears on nouns derived from verbs, e.g. *the falling of the leaves* (this is known as the **gerund**). In the Brown corpus, these are tagged `vbg`.

The **past participle** of a verb often ends in *-ed*, and expresses the idea of a completed action (e.g. *fell, ate*). These are tagged `vbd`.

[MORE: Modal verbs, e.g. *would* ...]

Common tag sets often capture some **morpho-syntactic** information; that is, information about the kind of morphological markings which words receive by virtue of their syntactic role. Consider, for example, the selection of distinct grammatical forms of the word *go* illustrated in the following sentences:

(11a) *Go* away!

(11b) He sometimes *goes* to the cafe.

(11c) All the cakes have *gone*.

(11d) We *went* on the excursion.

Each of these forms — *go*, *goes*, *gone*, and *went* — is morphologically distinct from the others. Consider the form, *goes*. This cannot occur in all grammatical contexts, but requires, for instance, a third person singular subject. Thus, the following sentences are ungrammatical.

(12a) *They sometimes *goes* to the cafe.

(12b) *I sometimes *goes* to the cafe.

By contrast, *gone* is the past participle form; it is required after *have* (and cannot be replaced in this context by *goes*), and cannot occur as the main verb of a clause.

(13a) *All the cakes have *goes*.

(13b) *He sometimes *gone* to the cafe.

We can easily imagine a tag set in which the four distinct grammatical forms just discussed were all tagged as VB. Although this would be adequate for some purposes, a more fine-grained tag set will provide useful information about these forms that can be of value to other processors which try to detect syntactic patterns from tag sequences. As we noted at the beginning of this chapter, the Brown tag set does in fact capture these distinctions, as summarized in Table 4.4.

Form	Category	Tag
go	base	vb
goes	3rd singular present	vbz
gone	past participle	vbn
went	simple past	vbd

Table 4.4: Some morphosyntactic distinctions in the Brown tag set

These differences between the forms are encoded in their Brown Corpus tags: `be/be`, `being/beg`, `am/bem`, `been/ben` and `was/bedz`. This means that an automatic tagger which uses this tag set is in effect carrying out a limited amount of morphological analysis.

Most part-of-speech tag sets make use of the same basic categories, such as noun, verb, adjective, and preposition. However, tag sets differ both in how finely they divide words into categories; and in how they define their categories. For example, *is* might be just tagged as a verb in one tag set; but as a distinct form of the lexeme *BE* in another tag set (as in the Brown Corpus). This variation in tag sets is unavoidable, since part-of-speech tags are used in different ways for different tasks. In other words, there is no one 'right way' to assign tags, only more or less useful ways depending on one's goals. More details about the Brown corpus tag set can be found in the Appendix.

4.3.2 The Regular Expression Tagger

The regular expression tagger assigns tags to tokens on the basis of matching patterns. For instance, we might guess that any word ending in *ed* is the past participle of a verb, and any word ending with *'s* is a possessive noun. We can express these as a list of regular expressions:

```
>>> patterns = [
...     (r'.*ing$', 'vbg'),               # gerunds
...     (r'.*ed$', 'vbd'),                # simple past
...     (r'.*es$', 'vbz'),                # 3rd singular present
...     (r'.*ould$', 'md'),               # modals
...     (r'.*\'s$', 'nn$'),               # possessive nouns
...     (r'.*s$', 'nns'),                 # plural nouns
...     (r'^-?[0-9]+(.[0-9]+)?$', 'cd'),  # cardinal numbers
...     (r'.*', 'nn')                     # nouns (default)
... ]
```

Note that these are processed in order, and the first one that matches is applied.

Now we can set up a tagger and use it to tag some text.

```
>>> regexp_tagger = tag.Regexp(patterns)
>>> list(regexp_tagger.tag(brown.raw('a')))[3]
[('``', 'nn'), ('Only', 'nn'), ('a', 'nn'), ('relative', 'nn'),
('handful', 'nn'), ('of', 'nn'), ('such', 'nn'), ('reports', 'nns'),
('was', 'nns'), ('received', 'vbd'), ("''", 'nn'), (',', 'nn'),
('the', 'nn'), ('jury', 'nn'), ('said', 'nn'), (',', 'nn'), ('``', 'nn'),
('considering', 'vbg'), ('the', 'nn'), ('widespread', 'nn'), ..., ('.', 'nn')]
```

How well does this do?

```
>>> tag.accuracy(regexp_tagger, brown.tagged('a'))
0.20326391789486245
```

The regular expression is a catch-all, which tags everything as a noun. This is equivalent to the default tagger (only much less efficient). Instead of re-specifying this as part of the regular expression tagger, is there a way to combine this tagger with the default tagger? We will see how to do this later, under the heading of backoff taggers.

4.3.3 Exercises

1. ☼ Search the web for "spoof newspaper headlines", to find such gems as: *British Left Waffles on Falkland Islands*, and *Juvenile Court to Try Shooting Defendant*. Manually tag these headlines to see if knowledge of the part-of-speech tags removes the ambiguity.

2. ☼ Satisfy yourself that there are restrictions on the distribution of *go* and *went*, in the sense that they cannot be freely interchanged in the kinds of contexts illustrated in (11).

3. ◑ Write code to search for particular words and phrases according to tags, to answer the following questions:

   a) Produce an alphabetically sorted list of the distinct words tagged as `md`.

   b) Identify words which can be plural nouns or third person singular verbs (e.g. *deals*, *flies*).

   c) Identify three-word prepositional phrases of the form IN + DET + NN (eg. *in the lab*).

   d) What is the ratio of masculine to feminine pronouns?

4. ◑ In the introduction we saw a table involving frequency counts for the adjectives *adore*, *love*, *like*, *prefer* and preceding qualifiers such as *really*. Investigate the full range of qualifiers (Brown tag `ql`) which appear before these four adjectives.

5. ◑ We defined the `regexp_tagger`, which can be used as a fall-back tagger for unknown words. This tagger only checks for cardinal numbers. By testing for particular prefix or suffix strings, it should be possible to guess other tags. For example, we could tag any word that ends with *-s* as a plural noun. Define a regular expression tagger (using `tag.Regexp` which tests for at least five other patterns in the spelling of words. (Use inline documentation to explain the rules.)

6. ◑ Consider the regular expression tagger developed in the exercises in the previous section. Evaluate the tagger using `tag.accuracy()`, and try to come up with ways to improve its performance. Discuss your findings. How does objective evaluation help in the development process?

7. ★ There are 264 distinct words in the Brown Corpus having exactly three possible tags.

   a) Print a table with the integers 1..10 in one column, and the number of distinct words in the corpus having 1..10 distinct tags.

   b) For the word with the greatest number of distinct tags, print out sentences from the corpus containing the word, one for each possible tag.

8. ★ Write a program to classify contexts involving the word *must* according to the tag of the following word. Can this be used to discriminate between the epistemic and deontic uses of *must*?

4.4 Baselines and Backoff

So far the performance of our simple taggers has been disappointing. Before we embark on a process to get 90+% performance, we need to do two more things. First, we need to establish a more principled baseline performance than the default tagger, which was too simplistic, and the regular expression tagger, which was too arbitrary. Second, we need a way to connect multiple taggers together, so that if a more specialized tagger is unable to assign a tag, we can "back off" to a more generalized tagger.

4.4.1 The Lookup Tagger

A lot of high-frequency words do not have the `nn` tag. Let's find some of these words and their tags. The function in Listing 4.2 takes a list of sentences and counts up the words, and returns the *n* most frequent words. We'll test out this function for the 100 most frequent words:

Listing 3 Program to Find the N Most Frequent Words

```
def wordcounts(sents, n):
    "find the n most frequent words"
    fd = FreqDist()
    for sent in sents:
        for word in sent:
            fd.inc(word)    # count the word
    return fd.sorted_samples()[:n]

>>> frequent_words = wordcounts(brown.raw('a'), 100)
>>> frequent_words
['the', ',', '.', 'of', 'and', 'to', 'a', 'in', 'for', 'The',
'that', '``', 'is', 'was', "''", 'on', 'at', 'with', 'be', 'by',
'as', 'he', 'said', 'his', 'will', 'it', 'from', 'are', ';', '--',
'an', 'has', 'had', 'who', 'have', 'not', 'Mrs.', 'were', 'this',
'which', 'would', 'their', 'been', 'they', 'He', 'one', ..., 'now']
```

Next, let's inspect the tags that these words have. First we will do this in the most obvious (but highly inefficient) way:

```
>>> [(w,t) for sent in brown.tagged('a')
...        for (w,t) in sent
...        if w in frequent_words]
[('The', 'at'), ('said', 'vbd'), ('an', 'at'), ('of', 'in'),
('``', '``'), ('no', 'at'), ("''", "''"), ('that', 'cs'),
('any', 'dti'), ('.', '.'), ..., ("''", "''")]
```

A much better approach is to set up a dictionary which maps each of the 100 most frequent words to its most likely tag. We can do this by setting up a frequency distribution `cfd` over the tags, conditioned on each of the frequent words, as shown in Listing 4.3. This gives us, for each word, a count of the frequency of different tags that occur with the word.

Now for any word that appears in this section of the corpus, we can look up its most likely tag. For example, to find the tag for the word *The* we can access the corresponding frequency distribution, and ask for its most frequent event:

Listing 4 Program to Find the Most Likely Tags for the Specified Words

```
from nltk_lite.probability import ConditionalFreqDist
def wordtags(tagged_sents, words):
    "Find the most likely tag for these words in the tagged sentences"
    cfd = ConditionalFreqDist()
    for sent in tagged_sents:
        for (w,t) in sent:
            if w in words:
                cfd[w].inc(t)    # count the word's tag
    return dict((word, cfd[word].max()) for word in words)
```

```
>>> table = wordtags(brown.tagged('a'), frequent_words)
>>> table['The']
'at'
```

Now we can create and evaluate a simple tagger that assigns tags to words based on this table:

```
>>> baseline_tagger = tag.Lookup(table)
>>> tag.accuracy(baseline_tagger, brown.tagged('a'))
0.45578495136941344
```

This is surprisingly good; just knowing the tags for the 100 most frequent words enables us to tag nearly half the words correctly! Let's see how it does on some untagged input text:

```
>>> list(baseline_tagger.tag(brown.raw('a')))[3]
[('``', '``'), ('Only', None), ('a', 'at'), ('relative', None),
('handful', None), ('of', 'in'), ('such', None), ('reports', None),
('was', 'bedz'), ('received', None), ("''", "''"), (',', ','),
('the', 'at'), ('jury', None), ('said', 'vbd'), (',', ','),
('``', '``'), ('considering', None), ('the', 'at'), ('widespread', None),
('interest', None), ('in', 'in'), ('the', 'at'), ('election', None),
(',', ','), ('the', 'at'), ('number', None), ('of', 'in'),
('voters', None), ('and', 'cc'), ('the', 'at'), ('size', None),
('of', 'in'), ('this', 'dt'), ('city', None), ("''", "''"), ('.', '.')]
```

Notice that a lot of these words have been assigned a tag of `None`. That is because they were not among the 100 most frequent words. In these cases we would like to assign the default tag of `nn`, a process known as backoff.

4.4.2 Backoff

How do we combine these taggers? We want to use the lookup table first, and if it is unable to assign a tag, then use the default tagger. We do this by specifying the default tagger as an argument to the lookup tagger. The lookup tagger will call the default tagger just in case it can't assign a tag itself.

```
>>> baseline_tagger = tag.Lookup(table, backoff=tag.Default('nn'))
>>> tag.accuracy(baseline_tagger, brown.tagged('a'))
0.58177695566561249
```

4.4.3 Choosing a good baseline

We can write a simple (but somewhat inefficient) program to create and evaluate lookup taggers having a range of sizes, as shown in Listing 4.4. We include a backoff tagger that tags everything as a noun. A consequence of using this backoff tagger is that the lookup tagger only has to store word/tag pairs for words other than nouns.

Listing 5 Lookup Tagger Performace with Varying Model Size

```
def performance(size):
    frequent_words = wordcounts(brown.raw('a'), size)
    table = wordtags(brown.tagged('a'), frequent_words)
    baseline_tagger = tag.Lookup(table, backoff=tag.Default('nn'))
    return tag.accuracy(baseline_tagger, brown.tagged('a'))

>>> from pylab import *
>>> sizes = 2**arange(15)
>>> perfs = [performance(size) for size in sizes]
>>> plot(sizes, perfs, '-bo')
>>> title('Lookup Tagger Performance with Varying Model Size')
>>> xlabel('Model Size')
>>> ylabel('Performance')
>>> show()
```

Observe that performance initially increases rapidly as the model size grows, eventually reaching a plateau, when large increases in model size yield little improvement in performance. (This example used the `pylab` plotting package; we will return to this later in Section 6.3.4).

4.4.4 Exercises

1. ◑ Explore the following issues that arise in connection with the lookup tagger:

    a) What happens to the tagger performance for the various model sizes when a backoff tagger is omitted?

    b) Consider the curve in Figure 4.2; suggest a good size for a lookup tagger that balances memory and performance. Can you come up with scenarios where it would be preferable to minimize memory usage, or to maximize performance with no regard for memory usage?

2. ◑ What is the upper limit of performance for a lookup tagger, assuming no limit to the size of its table? (Hint: write a program to work out what percentage of tokens of a word are assigned the most likely tag for that word, on average.)

4.5 Getting Better Coverage

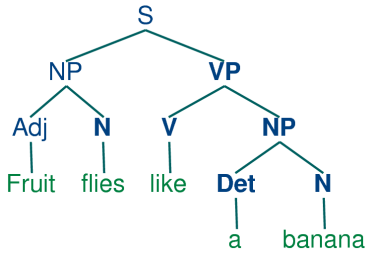
4.5.1 More English Word Classes

Two other important word classes are **adjectives** and **adverbs**. Adjectives describe nouns, and can be used as modifiers (e.g. *large* in *the large pizza*), or in predicates (e.g. *the pizza is large*). English

adjectives can be morphologically complex (e.g. *fall*$_V$*+ing* in *the falling stocks*). Adverbs modify verbs to specify the time, manner, place or direction of the event described by the verb (e.g. *quickly* in *the stocks fell quickly*). Adverbs may also modify adjectives (e.g. *really* in *Mary's teacher was really nice*).

English has several categories of closed class words in addition to prepositions, such as **articles** (also often called **determiners**) (e.g., *the*, *a*), **modals** (e.g., *should*, *may*), and **personal pronouns** (e.g., *she*, *they*). Each dictionary and grammar classifies these words differently.

Part-of-speech tags are closely related to the notion of word class used in syntax. The assumption in linguistics is that every distinct word type will be listed in a lexicon (or dictionary), with information about its pronunciation, syntactic properties and meaning. A key component of the word's properties will be its class. When we carry out a syntactic analysis of an example like *fruit flies like a banana*, we will look up each word in the lexicon, determine its word class, and then group it into a hierarchy of phrases, as illustrated in the following parse tree.



Syntactic analysis will be dealt with in more detail in Part II. For now, we simply want to make the connection between the labels used in syntactic parse trees and part-of-speech tags. Table 4.5 shows the correspondence:

Word Class Label	Brown Tag	Word Class
Det	AT	article
N	NN	noun
V	VB	verb
Adj	JJ	adjective
P	IN	preposition
Card	CD	cardinal number
--	.	Sentence-ending punctuation

Table 4.5: Word Class Labels and Brown Corpus Tags

4.5.2 Some diagnostics

Now that we have examined word classes in detail, we turn to a more basic question: how do we decide what category a word belongs to in the first place? In general, linguists use three criteria: morphological (or formal); syntactic (or distributional); semantic (or notional). A **morphological** criterion is one which looks at the internal structure of a word. For example, *-ness* is a suffix which combines with an adjective to produce a noun. Examples are *happy* → *happiness*, *ill* → *illness*. So if we encounter a word which ends in *-ness*, this is very likely to be a noun.

A **syntactic** criterion refers to the contexts in which a word can occur. For example, assume that we have already determined the category of nouns. Then we might say that a syntactic criterion for an

adjective in English is that it can occur immediately before a noun, or immediately following the words *be* or *very*. According to these tests, *near* should be categorized as an adjective:

(14a) the near window

(14b) The end is (very) near.

A familiar example of a **semantic** criterion is that a noun is "the name of a person, place or thing". Within modern linguistics, semantic criteria for word classes are treated with suspicion, mainly because they are hard to formalize. Nevertheless, semantic criteria underpin many of our intuitions about word classes, and enable us to make a good guess about the categorization of words in languages that we are unfamiliar with. For example, if we all we know about the Dutch *verjaardag* is that it means the same as the English word *birthday*, then we can guess that *verjaardag* is a noun in Dutch. However, some care is needed: although we might translate *zij is vandaag jarig* as *it's her birthday today*, the word *jarig* is in fact an adjective in Dutch, and has no exact equivalent in English!

All languages acquire new lexical items. A list of words recently added to the Oxford Dictionary of English includes *cyberslacker, fatoush, blamestorm, SARS, cantopop, bupkis, noughties, muggle*, and *robata*. Notice that all these new words are nouns, and this is reflected in calling nouns an *open class*. By contrast, prepositions are regarded as a **closed class**. That is, there is a limited set of words belonging to the class (e.g., *above, along, at, below, beside, between, during, for, from, in, near, on, outside, over, past, through, towards, under, up, with*), and membership of the set only changes very gradually over time.

With this background we are now ready to embark on our main task for this chapter, automatically assigning part-of-speech tags to words.

4.5.3 Unigram Tagging

The `tag.Unigram()` class implements a simple statistical tagging algorithm: for each token, it assigns the tag that is most likely for that particular token. For example, it will assign the tag `jj` to any occurrence of the word *frequent*, since *frequent* is used as an adjective (e.g. *a frequent word*) more often than it is used as a verb (e.g. *I frequent this cafe*).

Before a unigram tagger can be used to tag data, it must be trained on a tagged corpus. It uses this corpus to determine which tags are most common for each word. Unigram taggers are trained using the `train()` method, which takes a tagged corpus:

```
>>> from nltk_lite.corpora import brown
>>> from itertools import islice
>>> train_sents = list(islice(brown.tagged(), 500))  # sents 0..499
>>> unigram_tagger = tag.Unigram()
>>> unigram_tagger.train(train_sents)
```

Once a unigram tagger has been trained, the `tag()` method can be used to tag new text:

```
>>> text = "John saw the book on the table"
>>> tokens = list(tokenize.whitespace(text))
>>> list(unigram_tagger.tag(tokens))
[('John', 'np'), ('saw', 'vbd'), ('the', 'at'), ('book', None),
('on', 'in'), ('the', 'at'), ('table', None)]
```

The unigram tagger will assign the special tag `None` to any token that was not encountered in the training data.

4.5.4 Affix Taggers

Affix taggers are like unigram taggers, except they are trained on word prefixes or suffixes of a specified length. (NB. Here we use *prefix* and *suffix* in the string sense, not the morphological sense.) For example, the following tagger will consider suffixes of length 3 (e.g. *-ize*, *-ion*), for words having at least 5 characters.

```
>>> affix_tagger = tag.Affix(-2, 3)
>>> affix_tagger.train(train_sents)
>>> list(affix_tagger.tag(tokens))
[('John', 'np'), ('saw', 'nn'), ('the', 'at'), ('book', 'np'),
('on', None), ('the', 'at'), ('table', 'jj')]
```

4.5.5 Exercises

1. ☼ Train a unigram tagger and run it on some new text. Observe that some words are not assigned a tag. Why not?

2. ☼ Train an affix tagger `tag.Affix()` and run it on some new text. Experiment with different settings for the affix length and the minimum word length. Can you find a setting which seems to perform better than the one described above? Discuss your findings.

3. ◑ Write a program which calls `tag.Affix()` repeatedly, using different settings for the affix length and the minimum word length. What parameter values give the best overall performance? Why do you think this is the case?

4.6 N-Gram Taggers

Earlier we encountered the unigram tagger, which assigns a tag to a word based on the identity of that word. In this section we will look at taggers that exploit a larger amount of context when assigning a tag.

4.6.1 Bigram Taggers

Bigram taggers use two pieces of contextual information for each tagging decision, typically the current word together with the tag of the previous word. Given the context, the tagger assigns the most likely tag. In order to do this, the tagger uses a bigram table, a fragment of which is shown in Table 4.6. Given the tag of the previous word (down the left), and the current word (across the top), it can look up the preferred tag.

tag	ask	Congress	to	increase	grants	to	states
at				nn			
tl			to			to	
bd			to		nns	to	
md	*vb*			vb			
vb		*np*	to		*nns*	to	nns
np			*to*			to	
to	vb		*vb*				
nn		np	to	nn	nns	to	

tag	ask	Congress	to	increase	grants	to	states
nns			to			*to*	
in		np	in			in	*nns*
jj			to		nns	to	nns

Table 4.6: Fragment of Bigram Table

The best way to understand the table is to work through an example. Suppose we are processing the sentence *The President will ask Congress to increase grants to states for vocational rehabilitation* . and that we have got as far as `will/md`. We can use the table to simply read off the tags that should be assigned to the remainder of the sentence. When preceded by `md`, the tagger guesses that *ask* has the tag `vb` (italicized in the table). Moving to the next word, we know it is preceded by `vb`, and looking across this row we see that *Congress* is assigned the tag `np`. The process continues through the rest of the sentence. When we encounter the word *increase*, we correctly assign it the tag `vb` (unlike the unigram tagger which assigned it `nn`). However, the bigram tagger mistakenly assigns the infinitival tag to the word *to* immediately preceding *states*, and not the preposition tag. This suggests that we may need to consider even more context in order to get the correct tag.

4.6.2 N-Gram Taggers

As we have just seen, it may be desirable to look at more than just the preceding word's tag when making a tagging decision. An **n-gram tagger** is a generalization of a bigram tagger whose context is the current word together with the part-of-speech tags of the *n-1*: preceding tokens, as shown in the following diagram. It then picks the tag which is most likely for that context. The tag to be chosen, t_n, is circled, and the context is shaded in grey. In the example of an n-gram tagger shown in Figure 4.3, we have *n=3*; that is, we consider the tags of the two preceding words in addition to the current word.

> **Note**
>
> A 1-gram tagger is another term for a unigram tagger: i.e., the context used to tag a token is just the text of the token itself. 2-gram taggers are also called *bigram taggers*, and 3-gram taggers are called *trigram taggers*.

The `tag.Ngram` class uses a tagged training corpus to determine which part-of-speech tag is most likely for each context. Here we see a special case of an n-gram tagger, namely a bigram tagger:

```
>>> bigram_tagger = tag.Bigram()
>>> bigram_tagger.train(brown.tagged(['a','b']))
```

Once a bigram tagger has been trained, it can be used to tag untagged corpora:

```
>>> text = "John saw the book on the table"
>>> tokens = list(tokenize.whitespace(text))
>>> list(bigram_tagger.tag(tokens))
[('John', None), ('saw', None), ('the', 'at'), ('book', 'nn'),
('on', 'in'), ('the', 'at'), ('table', None)]
```

As with the other taggers, n-gram taggers assign the tag NONE to any token whose context was not seen during training.

As *n* gets larger, the specificity of the contexts increases, as does the chance that the data we wish to tag contains contexts that were not present in the training data. This is known as the *sparse data*

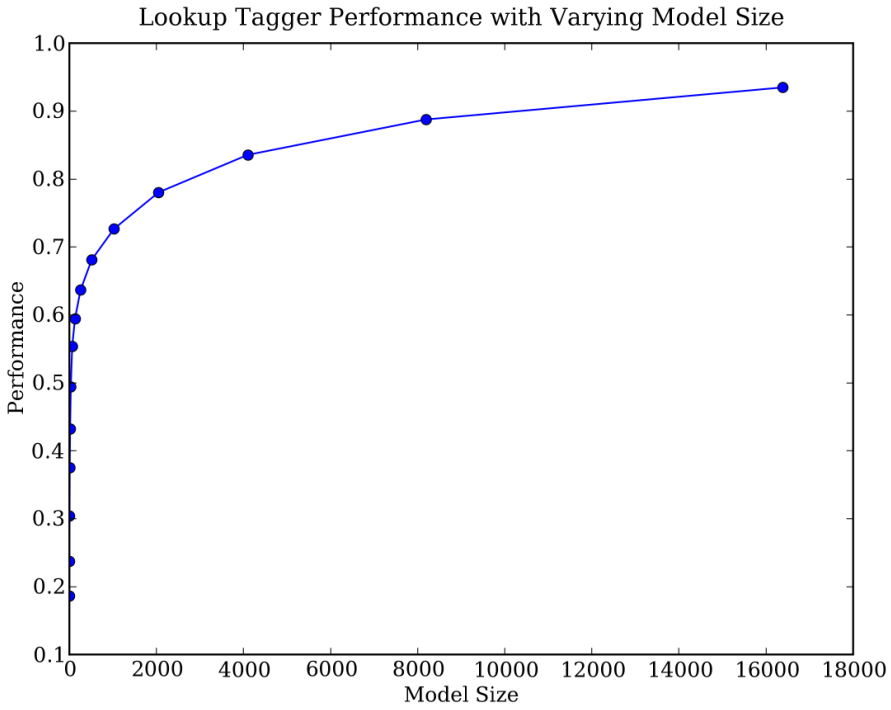


Figure 4.2: Lookup Tagger

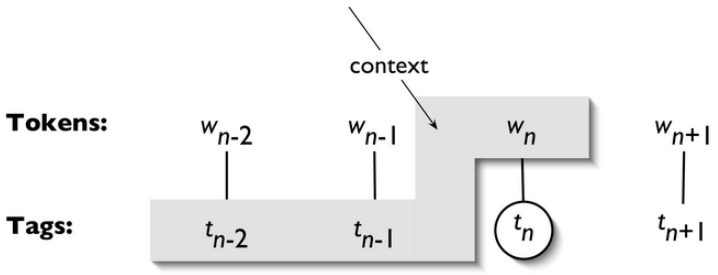


Figure 4.3: Tagger Context

problem, and is quite pervasive in NLP. Thus, there is a trade-off between the accuracy and the coverage of our results (and this is related to the **precision/recall trade-off** in information retrieval.)

> **Note**
>
> n-gram taggers should not consider context that crosses a sentence boundary. Accordingly, NLTK taggers are designed to work with lists of sentences, where each sentence is a list of words. At the start of a sentence, t_{n-1} and preceding tags are set to `None`.

4.6.3 Combining Taggers

One way to address the trade-off between accuracy and coverage is to use the more accurate algorithms when we can, but to fall back on algorithms with wider coverage when necessary. For example, we could combine the results of a bigram tagger, a unigram tagger, and a `regexp_tagger`, as follows:

1. Try tagging the token with the bigram tagger.
2. If the bigram tagger is unable to find a tag for the token, try the unigram tagger.
3. If the unigram tagger is also unable to find a tag, use a default tagger.

Each NLTK tagger other than `tag.Default` permits a backoff-tagger to be specified. The backoff-tagger may itself have a backoff tagger:

```
>>> t0 = tag.Default('nn')
>>> t1 = tag.Unigram(backoff=t0)
>>> t2 = tag.Bigram(backoff=t1)
>>> t1.train(brown.tagged('a'))    # section a: press-reportage
>>> t2.train(brown.tagged('a'))
```

> **Note**
>
> We specify the backoff tagger when the tagger is initialized, so that training can take advantage of the backing off. Thus, if the bigram tagger would assign the same tag as its unigram backoff tagger in a certain context, the bigram tagger discards the training instance. This keeps the bigram tagger model as small as possible. We can further specify that a tagger needs to see more than one instance of a context in order to retain it, e.g. `Bigram(cutoff=2, backoff=t1)` will discard contexts which have only been seen once or twice.

As before we test the taggers against unseen data. Here we will use a different segment of the corpus.

```
>>> accuracy0 = tag.accuracy(t0, brown.tagged('b')) # section b: press-editorial
>>> accuracy1 = tag.accuracy(t1, brown.tagged('b'))
>>> accuracy2 = tag.accuracy(t2, brown.tagged('b'))

>>> print 'Default Accuracy = %4.1f%%' % (100 * accuracy0)
Default Accuracy = 12.5%
>>> print 'Unigram Accuracy  = %4.1f%%' % (100 * accuracy1)
Unigram Accuracy  = 81.0%
>>> print 'Bigram Accuracy = %4.1f%%' % (100 * accuracy2)
Bigram Accuracy = 81.9%
```

4.6.4 Investigating tagger performance

What is the upper limit to tagger performance? Unfortunately perfect tagging is impossible. Consider the case of a trigram tagger. How many cases of part-of-speech ambiguity does it encounter? We can determine the answer to this question empirically:

```
>>> from nltk_lite.corpora import brown
>>> from nltk_lite.probability import ConditionalFreqDist
>>> cfdist = ConditionalFreqDist()
>>> for sent in brown.tagged('a'):
...     p = [(None, None)] # empty token/tag pair
...     trigrams = zip(p+p+sent, p+sent+p, sent+p+p)
...     for (pair1,pair2,pair3) in trigrams:
...         context = (pair1[1], pair2[1], pair3[0]) # last 2 tags, this word
...         cfdist[context].inc(pair3[1])            # current tag
>>> total = ambiguous = 0
>>> for cond in cfdist.conditions():
...     if cfdist[cond].B() > 1:
...         ambiguous += cfdist[cond].N()
...     total += cfdist[cond].N()
>>> print float(ambiguous) / total
0.0560190160471
```

Thus, one out of twenty trigrams is ambiguous. Given the current word and the previous two tags, there is more than one tag that could be legitimately assigned to the current word according to the training data. Assuming we always pick the most likely tag in such ambiguous contexts, we can derive an empirical upper bound on the performance of a trigram tagger.

Another way to investigate the performance of a tagger is to study its mistakes. Some tags may be harder than others to assign, and it might be possible to treat them specially by pre- or post-processing the data. A convenient way to look at tagging errors is the **confusion matrix**. It charts expected tags (the gold standard) against actual tags generated by a tagger:

```
>>> from nltk_lite.evaluate import ConfusionMatrix
>>> from nltk_lite.tag import untag
>>> def tag_list(tagged_sents):
...     return [tag for (word, tag) in sent for sent in tagged_sents]
>>> def apply_tagger(tagger, corpus):
...     return [tagger.tag(untag(sent)) for sent in corpus]
>>> gold = tag_list(brown.tagged('b'))
>>> test = tag_list(apply_tagger(t2, brown.tagged('b')))
>>> print ConfusionMatrix(gold, test)
```

Based on such analysis we may decide to modify the tagset. Perhaps a distinction between tags that is difficult to make can be dropped, since it is not important in the context of some larger processing task. We could collapse distinctions with the aid of a dictionary that maps the existing tagset to a simpler tagset:

```
>>> mapping = {'pps': 'nn', 'ppss': 'nn'}
>>> data = [(word, mapping.get(tag, tag))
            for (word, tag) in sent for sent in tagged_sents]
```

We can do this for both the training and test data.

4.6.5 Storing Taggers

Training a tagger on a large corpus may take several minutes. Instead of training a tagger every time we need one, it is convenient to save a trained tagger in a file for later re-use. Let's save our tagger `t2` to a file `t2.pkl`.

```
>>> from cPickle import dump
>>> output = open('t2.pkl', 'wb')
>>> dump(t2, output, -1)
>>> output.close()
```

Now, in a separate Python process, we can load our saved tagger.

```
>>> from cPickle import load
>>> input = open('t2.pkl', 'rb')
>>> tagger = load(input)
>>> input.close()
```

Now let's check that it can be used for tagging.

```
>>> text = """The board's action shows what free enterprise
...     is up against in our complex maze of regulatory laws ."""
>>> tokens = list(tokenize.whitespace(text))
>>> list(tagger.tag(tokens))
[('The', 'at'), ("board's", 'nn$'), ('action', 'nn'), ('shows', 'nns'),
('what', 'wdt'), ('free', 'jj'), ('enterprise', 'nn'), ('is', 'bez'),
('up', 'rp'), ('against', 'in'), ('in', 'in'), ('our', 'pp$'), ('complex', 'jj'),
('maze', 'nn'), ('of', 'in'), ('regulatory', 'nn'), ('laws', 'nns'), ('.', '.')]
```

4.6.6 Smoothing

[Brief discussion of NLTK's smoothing classes, for another approach to handling unknown words: Lidstone, Laplace, Expected Likelihood, Heldout, Witten-Bell, Good-Turing.]

4.6.7 Exercises

1. ☼ Train a bigram tagger with no backoff tagger, and run it on some of the training data. Next, run it on some new data. What happens to the performance of the tagger? Why?

2. ◑ Inspect the confusion matrix for the bigram tagger `t2` defined above, and identify one or more sets of tags to collapse. Define a dictionary to do the mapping, and evaluate the tagger on the simplified data.

3. ◑ How serious is the sparse data problem? Investigate the performance of n-gram taggers as n increases from 1 to 6. Tabulate the accuracy score. Estimate the training data required for these taggers, assuming a vocabulary size of 10^5 and a tagset size of 10^2.

4. ★ Create a default tagger and various unigram and n-gram taggers, incorporating backoff, and train them on part of the Brown corpus.

   a) Create three different combinations of the taggers. Test the accuracy of each combined tagger. Which combination works best?

   b) Try varying the size of the training corpus. How does it affect your results?

5. ★ Our approach for tagging an unknown word has been to consider the letters of the word (using `tag.Regexp()` and `tag.Affix()`), or to ignore the word altogether and tag it as a noun (using `tag.Default()`). These methods will not do well for texts having new words that are not nouns. Consider the sentence *I like to blog on Kim's blog*. If `blog`:lx: is a new word, then looking at the previous tag (`to` vs `np$`) would probably be helpful. I.e. we need a default tagger that is sensitive to the preceding tag.

   a) Create a new kind of unigram tagger that looks at the tag of the previous word, and ignores the current word. (The best way to do this is to modify the source code for `tag.Unigram()`, which presumes knowledge of Python classes discussed in Section 10.1.)

   b) Add this tagger to the sequence of backoff taggers (including ordinary trigram and bigram taggers that look at words), right before the usual default tagger.

   c) Evaluate the contribution of this new unigram tagger.

4.7 Conclusion

This chapter has introduced the language processing task known as tagging, with an emphasis on part-of-speech tagging. English word classes and their corresponding tags were introduced. We showed how tagged tokens and tagged corpora can be represented, then discussed a variety of taggers: default tagger, regular expression tagger, unigram tagger and n-gram taggers. We also described some objective evaluation methods. In the process, the reader has been introduced to an important paradigm in language processing, namely *language modeling*. This paradigm former is extremely general, and we will encounter it again later.

Observe that the tagging process simultaneously collapses distinctions (i.e., lexical identity is usually lost when all personal pronouns are tagged PRP), while introducing distinctions and removing ambiguities (e.g. *deal* tagged as VB or NN). This move facilitates classification and prediction. When we introduce finer distinctions in a tag set, we get better information about linguistic context, but we have to do more work to classify the current token (there are more tags to choose from). Conversely, with fewer distinctions, we have less work to do for classifying the current token, but less information about the context to draw on.

There are several other important approaches to tagging involving *Transformation-Based Learning*, *Markov Modeling*, and *Finite State Methods*. We will discuss these in a later chapter. In Chapter 5 we will see a generalization of tagging called *chunking* in which a contiguous sequence of words is assigned a single tag.

Part-of-speech tagging is just one kind of tagging, one that does not depend on deep linguistic analysis. There are many other kinds of tagging. Words can be tagged with directives to a speech synthesizer, indicating which words should be emphasized. Words can be tagged with sense numbers, indicating which sense of the word was used. Words can also be tagged with morphological features. Examples of each of these kinds of tags are shown below. For space reasons, we only show the tag for a single word. Note also that the first two examples use XML-style tags, where elements in angle brackets enclose the word that is tagged.

1. *Speech Synthesis Markup Language (W3C SSML):* `That is a <emphasis>big</emphasis> car!`

2. *SemCor: Brown Corpus tagged with WordNet senses:* `Space in any <wf pos="NN " lemma="form" wnsn="4">form</wf> is completely measured by the three dimensions.` (Wordnet form/nn sense 4: "shape, form, configuration, contour, conformation")

3. *Morphological tagging, from the Turin University Italian Treebank:* `E' italiano , come progetto e realizzazione , il primo (PRIMO ADJ ORDIN M SING ) porto turistico dell' Albania .`

Tagging exhibits several properties that are characteristic of natural language processing. First, tagging involves *classification*: words have properties; many words share the same property (e.g. `cat` and `dog` are both nouns), while some words can have multiple such properties (e.g. `wind` is a noun and a verb). Second, in tagging, disambiguation occurs via *representation*: we augment the representation of tokens with part-of-speech tags. Third, training a tagger involves *sequence learning from annotated corpora*. Finally, tagging uses *simple, general, methods* such as conditional frequency distributions and transformation-based learning.

We have seen that ambiguity in the training data leads to an upper limit in tagger performance. Sometimes more context will resolve the ambiguity. In other cases however, as noted by Abney (1996), the ambiguity can only resolved with reference to syntax, or to world knowledge. Despite these imperfections, part-of-speech tagging has played a central role in the rise of statistical approaches to natural language processing. In the early 1990s, the surprising accuracy of statistical taggers was a striking demonstration that it was possible to solve one small part of the language understanding problem, namely part-of-speech disambiguation, without reference to deeper sources of linguistic knowledge. Can this idea be pushed further? In Chapter 5, on chunk parsing, we shall see that it can.

4.8 Further Reading

Tagging: Jurafsky and Martin, Chapter 8

Brill tagging: Manning and Schutze 361ff; Jurafsky and Martin 307ff

HMM tagging: Manning and Schutze 345ff

Abney, Steven (1996). Tagging and Partial Parsing. In: Ken Church, Steve Young, and Gerrit Bloothooft (eds.), *Corpus-Based Methods in Language and Speech.* Kluwer Academic Publishers, Dordrecht. `http://www.vinartus.net/spa/95a.pdf`

Wikipedia: `http://en.wikipedia.org/wiki/Part-of-speech_tagging`

List of available taggers: `http://www-nlp.stanford.edu/links/statnlp.html`

4.9 Further Exercises

1. **Impossibility of exact tagging:** Write a program to determine the upper bound for accuracy of an n-gram tagger. Hint: how often is the context seen during training inadequate for uniquely determining the tag to assign to a word?

2. **Impossibility of exact tagging:** Consult the Abney reading and review his discussion of the impossibility of exact tagging. Explain why correct tagging of these examples requires access to other kinds of information than just words and tags. How might you estimate the scale of this problem?

3. **Application to other languages**: Obtain some tagged data for another language, and train and evaluate a variety of taggers on it. If the language is morphologically complex, or if there are any orthographic clues (e.g. capitalization) to word classes, consider developing a regular expression tagger for it (ordered after the unigram tagger, and before the default tagger). How does the accuracy of your tagger(s) compare with the same taggers run on English data? Discuss any issues you encounter in applying these methods to the language.

4. **Comparing n-gram taggers and Brill taggers** (advanced): Investigate the relative performance of n-gram taggers with backoff and Brill taggers as the size of the training data is increased. Consider the training time, running time, memory usage, and accuracy, for a range of different parameterizations of each technique.

5. **HMM taggers**: Explore the Hidden Markov Model tagger `nltk_lite.tag.hmm`.

6. (Advanced) **Estimation:** Use some of the estimation techniques in `nltk_lite.probability`, such as *Lidstone* or *Laplace* estimation, to develop a statistical tagger that does a better job than ngram backoff taggers in cases where contexts encountered during testing were not seen during training. Read up on the TnT tagger, since this provides useful technical background: `http://www.aclweb.org/anthology/A00-1031`

4.10 Appendix: Brown Tag Set

Table 4.7 gives a sample of closed class words, following the classification of the Brown Corpus. (Note that part-of-speech tags may be presented as either upper-case or lower-case strings -- the case difference is not significant.)

ap	determiner/pronoun, post-determiner	many other next more last former little several enough most least only very few fewer past same
at	article	the an no a every th' ever' ye
cc	conjunction, coordinating	and or but plus & either neither nor yet 'n' and/or minus an'
cs	conjunction, subordinating	that as after whether before while like because if since for than until so unless though providing once lest till whereas whereupon supposing albeit then
in	preposition	of in for by considering to on among at through with under into regarding than since despite ...
md	modal auxiliary	should may might will would must can could shall ought need wilt
pn	pronoun, nominal	none something everything one anyone nothing nobody everybody everyone anybody anything someone no-one nothin'
ppl	pronoun, singular, reflexive	itself himself myself yourself herself oneself ownself
pp$	determiner, possessive	our its his their my your her out thy mine thine
pp$$	pronoun, possessive	ours mine his hers theirs yours
pps	pronoun, personal, nom, 3rd pers sng	it he she thee

ppss	pronoun, personal, nom, not 3rd pers sng	they we I you ye thou you'uns
wdt	WH-determiner	which what whatever whichever
wps	WH-pronoun, nominative	that who whoever whosoever what whatsoever

Table 4.7: Some English Closed Class Words, with Brown Tag

4.10.1 Acknowledgments

Chapter 5

Chunking

5.1 Introduction

Chunking is an efficient and robust method for identifying short phrases in text, or "chunks". Chunks are *non-overlapping spans of text*, usually consisting of a head word (such as a noun) and the adjacent modifiers and function words (such as adjectives and determiners). For example, here is some Wall Street Journal text with noun phrase chunks marked using brackets (this data is distributed with NLTK):

> [The/DT market/NN] for/IN [system-management/NN software/NN] for/IN [Digital/NNP] ['s/POS hardware/NN] is/VBZ fragmented/JJ enough/RB that/IN [a/DT giant/NN] such/JJ as/IN [Computer/NNP Associates/NNPS] should/MD do/VB well/RB there/RB ./.

There are two motivations for chunking: to locate information, and to ignore information. In the former case, we may want to extract all noun phrases so they can be indexed. A text retrieval system could use such an index to support efficient retrieval for queries involving terminological expressions.

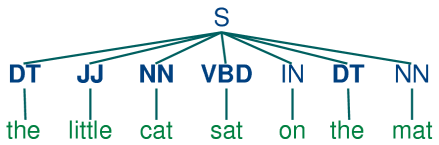
The reverse side of the coin is to *ignore* information. Suppose that we want to study syntactic patterns, finding particular verbs in a corpus and displaying their arguments. For instance, here are some uses of the verb `gave` in the Wall Street Journal (in the Penn Treebank corpus sample). After doing NP-chunking, the internal details of each noun phrase have been suppessed, allowing us to see some higher-level patterns:

```
gave NP
gave up NP in NP
gave NP up
gave NP NP
gave NP to NP
```

In this way we can acquire information about the complementation patterns of a verb like `gave`, for use in the development of a grammar (see Chapter 7).

Chunking in NLTK begins with tagged tokens, converted into a tree. (We will learn all about trees in Part II; for now its enough to know that they are hierarchical structures built over sequences of tagged tokens.)

```
>>> from nltk_lite.parse import Tree
>>> tagged_tokens = [("the", "DT"), ("little", "JJ"), ("cat", "NN"),
...      ("sat", "VBD"), ("on", "IN"), ("the", "DT"), ("mat", "NN")]
>>> Tree('S', tagged_tokens).draw()
```

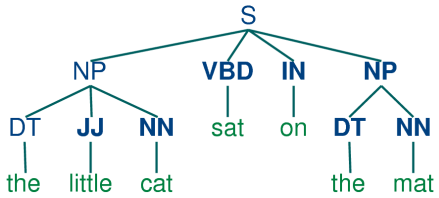


Next, we write regular expressions over tag sequences. The following example identifies noun phrases that consist of an optional determiner, followed by any number of adjectives, then a noun.

```
>>> from nltk_lite import chunk
>>> cp = chunk.Regexp("NP: {<DT>?<JJ>*<NN>}")
```

We create a chunker `cp` which can then be used repeatedly to parse tagged input. The result of chunking is also a tree, but with some extra structure:

```
>>> cp.parse(tagged_tokens).draw()
```



In this chapter we explore chunking in depth, beginning with the definition and representation of chunks. We will see regular expression and n-gram approaches to chunking, and will develop and evaluate chunkers using the CoNLL-2000 chunking corpus.

5.2 Defining and Representing Chunks

5.2.1 An Analogy

Two of the most common operations in language processing are *segmentation* and *labeling*. Recall that in tokenization, we *segment* a sequence of characters into tokens, while in tagging we *label* each of these tokens. Moreover, these two operations of segmentation and labeling go hand in hand. We break up a stream of characters into linguistically meaningful segments (e.g. words) so that we can classify those segments with their part-of-speech categories. The result of such classification is represented by adding a label to the segment in question.

In this chapter we do this segmentation and labeling at the phrase level, as illustrated in Figure 5.1. The solid boxes show word-level segmentation and labeling, while the dashed boxes show a higher-level segmentation and labeling. These larger pieces are called **chunks**, and the process of identifying them is called **chunking**.

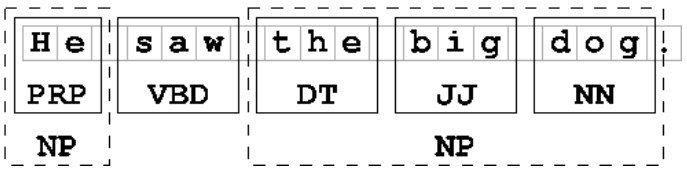


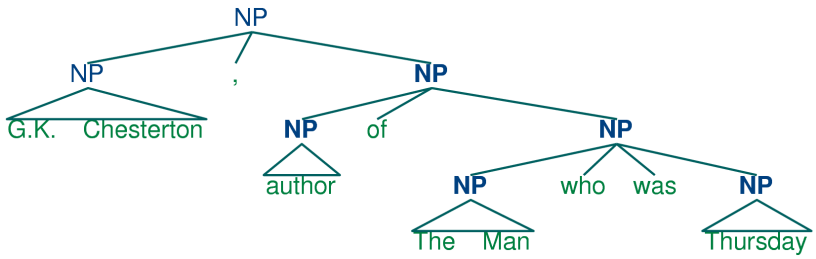
Figure 5.1: Segmentation and Labeling at both the Token and Chunk Levels

Like tokenization, chunking can skip over material in the input. Tokenization omits white space and punctuation characters. Chunking uses only a subset of the tokens and leaves others out.

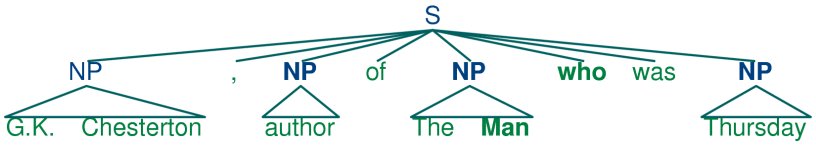
5.2.2 Chunking vs Parsing

Chunking is akin to parsing in the sense that it can be used to build hierarchical structure over text. There are several important differences, however. First, as noted above, chunking is not exhaustive, and typically omits items in the surface string. Second, where parsing constructs deeply nested structures, chunking creates structures of fixed depth, (typically depth 2). These chunks often correspond to the lowest level of grouping identified in the full parse tree, as illustrated in the parsing and chunking examples in (15) below:

(15a)



(15b)



A significant motivation for chunking is its robustness and efficiency relative to parsing. Parsing uses recursive phrase structure grammars and arbitrary-depth trees. Parsing has problems with robustness, given the difficulty in getting broad coverage and in resolving ambiguity. Parsing is also relatively inefficient: the time taken to parse a sentence grows with the cube of the length of the sentence, while the time taken to chunk a sentence only grows linearly.

5.2.3 Representing Chunks: Tags vs Trees

As befits its intermediate status between tagging and parsing, chunk structures can be represented using either tags or trees. The most widespread file representation uses so-called **IOB tags**. In this scheme, each token is tagged with one of three special chunk tags, `I` (inside), `O` (outside), or `B` (begin). A token is tagged as `B` if it marks the beginning of a chunk. Subsequent tokens within the chunk are tagged `I`. All other tokens are tagged `O`. The `B` and `I` tags are suffixed with the chunk type, e.g. `B-NP`, `I-NP`. Of course, it is not necessary to specify a chunk type for tokens that appear outside a chunk, so these are just labeled `O`. An example of this scheme is shown in Figure 5.2.

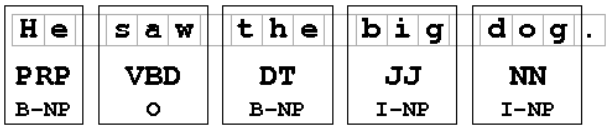


Figure 5.2: Tag Representation of Chunk Structures

IOB tags have become the standard way to represent chunk structures in files, and we will also be using this format. Here is an example of the file representation of the information in Figure 5.2:

```
He PRP B-NP
saw VBD O
the DT B-NP
big JJ I-NP
dog NN I-NP
```

In this representation, there is one token per line, each with its part-of-speech tag and its chunk tag. We will see later that this format permits us to represent more than one chunk type, so long as the chunks do not overlap. This file format was developed for NP chunking by [Ramshaw and Marcus, 1995], and was used for the shared NP bracketing task run by the *Conference on Natural Language Learning* (CoNLL) in 1999. It has come to be called the **IOB Format** (or sometimes **BIO Format**). The same format was adopted by CoNLL 2000 for annotating a section of Wall Street Journal text as part of a shared task on NP chunking.

As we saw earlier, chunk structures can also be represented using trees. These have the benefit that each chunk is a constituent that can be manipulated directly. An example is shown in Figure 5.3:

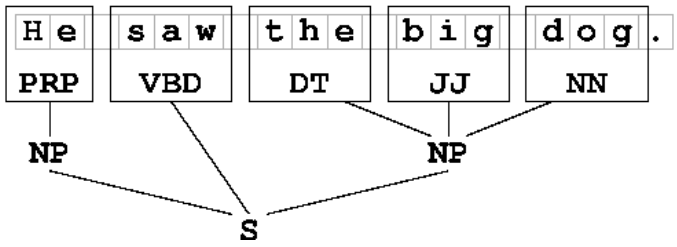


Figure 5.3: Tree Representation of Chunk Structures

NLTK uses trees for its internal representation of chunks, and provides methods for reading and writing such trees to the IOB format. By now you should understand what chunks are, and how they are represented. In the next section you will see how to build a simple chunker.

5.3 Chunking

A **chunker** finds contiguous, non-overlapping spans of related tokens and groups them together into *chunks*. Chunkers often operate on tagged texts, and use the tags to make chunking decisions. In this section we will see how to write a special type of regular expression over part-of-speech tags, and then how to combine these into a chunk grammar. Then we will set up a chunker to chunk some tagged text according to the grammar.

5.3.1 Tag Patterns

A **tag pattern** is a sequence of part-of-speech tags delimited using angle brackets, e.g. `<DT><JJ><NN>`. Tag patterns are the same as the regular expression patterns we have already seen, except for two differences which make them easier to use for chunking. First, angle brackets group their contents into atomic units, so "`<NN>+`" matches one or more repetitions of the tag `NN`; and "`<NN|JJ>`" matches the `NN` or `JJ`. Second, the period wildcard operator is constrained not to cross tag delimiters, so that "`<N.*>`" matches any single tag starting with `N`, e.g. `NN`, `NNS`.

Now, consider the following noun phrases from the Wall Street Journal:

```
another/DT sharp/JJ dive/NN
trade/NN figures/NNS
any/DT new/JJ policy/NN measures/NNS
earlier/JJR stages/NNS
Panamanian/JJ dictator/NN Manuel/NNP Noriega/NNP
```

We can match these using a slight refinement of the first tag pattern above: `<DT>?<JJ.*>*<NN.*>+`. This can be used to chunk any sequence of tokens beginning with an optional determiner `DT`, followed by zero or more adjectives of any type `JJ.*` (including relative adjectives like `earlier/JJR`), followed by one or more nouns of any type `NN.*`. It is easy to find many more difficult examples:

```
his/PRP$ Mansion/NNP House/NNP speech/NN
the/DT price/NN cutting/VBG
3/CD %/NN to/TO 4/CD %/NN
more/JJR than/IN 10/CD %/NN
the/DT fastest/JJS developing/VBG trends/NNS
's/POS skill/NN
```

Your challenge will be to come up with tag patterns to cover these and other examples.

5.3.2 Chunking with Regular Expressions

The chunker begins with a flat structure in which no tokens are chunked. Patterns are applied in turn, successively updating the chunk structure. Once all of the patterns have been applied, the resulting chunk structure is returned. Listing 5.1 shows a simple chunk grammar consisting of two patterns. The first pattern matches an optional determiner, zero or more adjectives, then a noun. We also define some tagged tokens to be chunked, and run the chunker on this input.

Listing 6 Simple Noun Phrase Chunker

```
grammar = r"""
NP:
  {<DT>?<JJ>*<NN>}     # chunk determiners, adjectives and nouns
  {<NNP>+}             # chunk sequences of proper nouns
"""
cp = chunk.Regexp(grammar)
tagged_tokens = [("the", "DT"), ("little", "JJ"), ("cat", "NN"),
     ("sat", "VBD"), ("on", "IN"), ("the", "DT"), ("mat", "NN")]

>>> cp.parse(tagged_tokens)
(S:
  (NP: ('the', 'DT') ('little', 'JJ') ('cat', 'NN'))
  ('sat', 'VBD')
  ('on', 'IN')
  (NP: ('the', 'DT') ('mat', 'NN')))
```

If a tag pattern matches at overlapping locations, the first match takes precedence. For example, if we apply a rule that matches two consecutive nouns to a text containing three consecutive nouns, then only the first two nouns will be chunked:

```
>>> nouns = [("money", "NN"), ("market", "NN"), ("fund", "NN")]
```

```
>>> grammar = "NP: {<NN><NN>}  # Chunk two consecutive nouns"
>>> cp = chunk.Regexp(grammar)
>>> print cp.parse(nouns)
(S: (NP: ('money', 'NN') ('market', 'NN')) ('fund', 'NN'))
```

Once we have created the chunk for *money market*, we have removed the context that would have permitted *fund* to be included in a chunk. This issue would have been avoided with a more permissive chunk rule, e.g. `NP: {<NN>+}`.

5.3.3 Developing Chunkers

Creating a good chunker usually requires several rounds of development and testing, during which existing rules are refined and new rules are added. In order to diagnose any problems, it often helps to trace the execution of a chunker, using its `trace` argument. The tracing output shows the rules that are applied, and uses braces to show the chunks that are created at each stage of processing. In Listing 5.2, two chunk patterns are applied to the input sentence. The first rule finds all sequences of three tokens whose tags are `DT`, `JJ`, and `NN`, and the second rule finds any sequence of tokens whose tags are either `DT` or `NN`. We set up two chunkers one for each rule ordering, and test them on the same input.

Observe that when we chunk material that is already partially chunked, the chunker will only create chunks that do not partially overlap existing chunks. In the case of `cp2`, the second rule did not find any chunks, since all chunks that matched its tag pattern overlapped with existing chunks. Therefore it is necessary to be careful to put chunk rules in the right order.

5.3.4 Exercises

1. ☼ **Chunking Demonstration:** Run the chunking demonstration:

   ```
   from nltk_lite import chunk
   chunk.demo()  # the chunker
   ```

2. ☼ **IOB Tags:** The IOB format categorizes tagged tokens as `I`, `O` and `B`. Why are three tags necessary? What problem would be caused if we used `I` and `O` tags exclusively?

3. ☼ Write a tag pattern to match noun phrases containing plural head nouns, e.g. "many/JJ researchers/NNS", "two/CD weeks/NNS", "both/DT new/JJ positions/NNS". Try to do this by generalizing the tag pattern that handled singular noun phrases.

4. ◑ Write tag pattern to cover noun phrases that contain gerunds, e.g. "the/DT receiving/VBG end/NN", "assistant/NN managing/VBG editor/NN". Add these patterns to the grammar, one per line. Test your work using some tagged sentences of your own devising.

5. ◑ Write one or more tag patterns to handle coordinated noun phrases, e.g. "July/NNP and/CC August/NNP", "all/DT your/PRP$ managers/NNS and/CC supervisors/NNS", "company/NN courts/NNS and/CC adjudicators/NNS".

6. ◑ Sometimes a word is incorrectly tagged, e.g. the head noun in "12/CD or/CC so/RB cases/VBZ". Instead of requiring manual correction of tagger output, good chunkers are able to work with the erroneous output of taggers. Look for other examples of correctly chunked noun phrases with incorrect tags.

Listing 7 Two Noun Phrase Chunkers Having Identical Rules in Different Orders

```
cp1 = chunk.Regexp(r"""
NP: {<DT><JJ><NN>}      # Chunk det+adj+noun
    {<DT|NN>+}          # Chunk sequences of NN and DT
""")
cp2 = chunk.Regexp(r"""
NP: {<DT|NN>+}          # Chunk sequences of NN and DT
    {<DT><JJ><NN>}      # Chunk det+adj+noun
""")

>>> print cp1.parse(tagged_tokens, trace=1)
# Input:
 <DT>  <JJ>  <NN>  <VBD>  <IN>  <DT>  <NN>
# Chunk det+adj+noun:
{<DT>  <JJ>  <NN>} <VBD>  <IN>  <DT>  <NN>
# Chunk sequences of NN and DT:
{<DT>  <JJ>  <NN>} <VBD>  <IN> {<DT>  <NN>}
(S:
  (NP: ('the', 'DT') ('little', 'JJ') ('cat', 'NN'))
  ('sat', 'VBD')
  ('on', 'IN')
  (NP: ('the', 'DT') ('mat', 'NN')))
>>> print cp2.parse(tagged_tokens, trace=1)
# Input:
 <DT>  <JJ>  <NN>  <VBD>  <IN>  <DT>  <NN>
# Chunk sequences of NN and DT:
{<DT>} <JJ> {<NN>} <VBD>  <IN> {<DT>  <NN>}
# Chunk det+adj+noun:
{<DT>} <JJ> {<NN>} <VBD>  <IN> {<DT>  <NN>}
(S:
  (NP: ('the', 'DT'))
  ('little', 'JJ')
  (NP: ('cat', 'NN'))
  ('sat', 'VBD')
  ('on', 'IN')
  (NP: ('the', 'DT') ('mat', 'NN')))
```

5.4 Scaling Up

Now that you have a taste of what chunking can do, you are ready to look at a chunked corpus, and use it in developing and testing more complex chunkers. We will begin by looking at the mechanics of converting IOB format into an NLTK tree, then at how this is done on a larger scale using the corpus directly. We will see how to use the corpus to score the accuracy of a chunker, then look some more flexible ways to manipulate chunks. Throughout our focus will be on scaling up the coverage of a chunker.

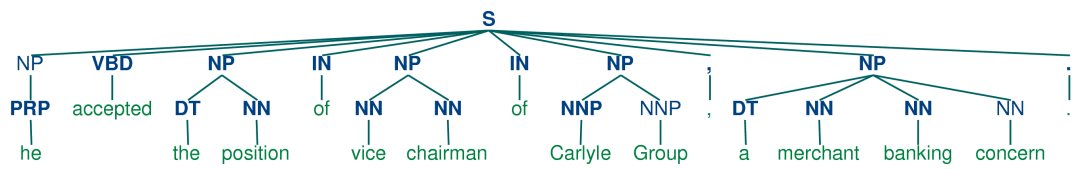
5.4.1 Reading IOB Format and the CoNLL 2000 Corpus

Using the `nltk_lite.corpora` module we can load Wall Street Journal text that has been tagged, then chunked using the IOB notation. The chunk categories provided in this corpus are NP, VP and PP. As we have seen, each sentence is represented using multiple lines, as shown below:

```
he PRP B-NP
accepted VBD B-VP
the DT B-NP
position NN I-NP
...
```

A conversion function `chunk.conllstr2tree()` builds a tree representation from one of these multi-line strings. Moreover, it permits us to choose any subset of the three chunk types to use. The example below produces only `NP` chunks:

```
>>> text = '''
... he PRP B-NP
... accepted VBD B-VP
... the DT B-NP
... position NN I-NP
... of IN B-PP
... vice NN B-NP
... chairman NN I-NP
... of IN B-PP
... Carlyle NNP B-NP
... Group NNP I-NP
... , , O
... a DT B-NP
... merchant NN I-NP
... banking NN I-NP
... concern NN I-NP
... . . O
... '''
>>> chunk.conllstr2tree(text, chunk_types=('NP',)).draw()
```



We can use the NLTK corpus module to access a larger amount of chunked text. The CoNLL 2000 corpus contains 270k words of Wall Street Journal text, with part-of-speech tags and chunk tags in the

IOB format. We can access this data using an NLTK corpus reader called `conll2000`. Here is an example:

```
>>> from nltk_lite.corpora import conll2000, extract
>>> print extract(2000, conll2000.chunked())
(S:
  (NP: ('Health-care', 'JJ') ('companies', 'NNS'))
  (VP: ('should', 'MD') ('get', 'VB'))
  ('healthier', 'JJR')
  (PP: ('in', 'IN'))
  (NP: ('the', 'DT') ('third', 'JJ') ('quarter', 'NN'))
  ('.', '.'))
```

This just showed three chunk types, for NP, VP and PP. We can also select which chunk types to read:

```
>>> from nltk_lite.corpora import conll2000, extract
>>> print extract(2000, conll2000.chunked(chunk_types=('NP',)))
(S:
  (NP: ('Health-care', 'JJ') ('companies', 'NNS'))
  ('should', 'MD')
  ('get', 'VB')
  ('healthier', 'JJR')
  ('in', 'IN')
  (NP: ('the', 'DT') ('third', 'JJ') ('quarter', 'NN'))
  ('.', '.'))
```

5.4.2 Simple Evaluation and Baselines

Armed with a corpus, it is now possible to do some simple evaluation. The first evaluation is to establish a baseline for the case where nothing is chunked:

```
>>> cp = chunk.Regexp("")
>>> print chunk.accuracy(cp, conll2000.chunked(chunk_types=('NP',)))
0.440845995079
```

Now let's try a naive regular expression chunker that looks for tags beginning with letters that are typical of noun phrase tags:

```
>>> grammar = r"NP: {<[CDJNP].*>+}"
>>> cp = chunk.Regexp(grammar)
>>> print chunk.accuracy(cp, conll2000.chunked(chunk_types=('NP',)))
0.874479872666
```

We can extend this approach, and create a function `chunked_tags()` that takes some chunked data, and sets up a conditional frequency distribution. For each tag, it counts up the number of times the tag occurs inside an NP chunk (the `True` case), or outside a chunk (the `False` case). It returns a list of those tags that occur inside chunks more often than outside chunks.

```
>>> def chunked_tags(train):
...     """Generate a list of tags that tend to appear inside chunks"""
...     from nltk_lite.probability import ConditionalFreqDist
...     cfdist = ConditionalFreqDist()
...     for t in train:
...         for word, tag, chtag in chunk.tree2conlltags(t):
```

```
...             if chtag == "O":
...                 cfdist[tag].inc(False)
...             else:
...                 cfdist[tag].inc(True)
...     return [tag for tag in cfdist.conditions() if cfdist[tag].max() == True]
```

The next step is to convert this list of tags into a tag pattern. To do this we need to "escape" all non-word characters, by preceding them with a backslash. Then we need to join them into a disjunction. This process would convert a tag list `['NN', 'NN$']` into the tag pattern `<NN|NN\$>`. The following function does this work, and returns a regular expression chunker:

```
>>> def baseline_chunker(train):
...     import re
...     chunk_tags = [re.sub(r'(\W)', r'\\\1', tag)
...                   for tag in chunked_tags(train)]
...     grammar = 'NP: {<' + '|'.join(chunk_tags) + '>+}'
...     return chunk.Regexp(grammar)
```

The final step is to train this chunker and test its accuracy (this time on data not seen during training):

```
>>> cp = baseline_chunker(conll2000.chunked(files='train', chunk_types=('NP',)))
>>> print chunk.accuracy(cp, conll2000.chunked(files='test', chunk_types=('NP',)))
0.914262194736
```

5.4.3 Splitting and Merging (incomplete)

[Notes: the above approach creates chunks that are too large, e.g. *the cat the dog chased* would be given a single NP chunk because it does not detect that determiners introduce new chunks. For this we would need a rule to split an NP chunk prior to any determiner, using a pattern like: `"NP: <.*>}{<DT>"`. We can also merge chunks, e.g. `"NP: <NN>{}<NN>"`.]

5.4.4 Chinking

Sometimes it is easier to define what we *don't* want to include in a chunk than it is to define what we *do* want to include. In these cases, it may be easier to build a chunker using a method called **chinking**.

The word **chink** initially meant a sequence of stopwords, according to a 1975 paper by Ross and Tukey [Abney, 1996a]. Following Abney, we define a *chink* is a sequence of tokens that is not included in a chunk. In the following example, `sat/VBD on/IN` is a chink:

```
[ the/DT little/JJ cat/NN ] sat/VBD on/IN [ the/DT mat/NN ]
```

Chinking is the process of removing a sequence of tokens from a chunk. If the sequence of tokens spans an entire chunk, then the whole chunk is removed; if the sequence of tokens appears in the middle of the chunk, these tokens are removed, leaving two chunks where there was only one before. If the sequence is at the beginning or end of the chunk, these tokens are removed, and a smaller chunk remains. These three possibilities are illustrated in Table 5.1.

	Entire chunk	Middle of a chunk	End of a chunk
Input	[a/DT big/JJ cat/NN]	[a/DT big/JJ cat/NN]	[a/DT big/JJ cat/NN]
Operation	Chink "DT JJ NN"	Chink "JJ"	Chink "NN"
Pattern	"}DT JJ NN{"	"}JJ{"	"}NN{"

Output	a/DT big/JJ cat/NN	[a/DT] big/JJ [cat/NN]	[a/DT big/JJ] cat/NN

Table 5.1: Three chinking rules applied to the same chunk

In the following grammar, we put the entire sentence into a single chunk, then excise the chink:

```
>>> grammar = r"""
... NP:
...   {<.*>+}          # Chunk everything
...   }<VBD|IN>+{      # Chink sequences of VBD and IN
... """
>>> cp = chunk.Regexp(grammar)
>>> print cp.parse(tagged_tokens)
(S:
  (NP: ('the', 'DT') ('little', 'JJ') ('cat', 'NN'))
  ('sat', 'VBD')
  ('on', 'IN')
  (NP: ('the', 'DT') ('mat', 'NN')))
>>> print chunk.accuracy(cp, conll2000.chunked(files='test', chunk_types=('NP',)))
0.581041433607
```

A chunk grammar can use any number of chunking and chinking patterns in any order.

5.4.5 Multiple Chunk Types (incomplete)

So far we have only developed NP chunkers. However, as we saw earlier in the chapter, the CoNLL chunking data is also annotated for PP and VP chunks. Here is an example, to show the structure we get from the corpus and the flattened version that will be used as input to the parser.

```
>>> example = extract(2000, conll2000.chunked())
>>> print example
(S:
  (NP: ('Health-care', 'JJ') ('companies', 'NNS'))
  (VP: ('should', 'MD') ('get', 'VB'))
  ('healthier', 'JJR')
  (PP: ('in', 'IN'))
  (NP: ('the', 'DT') ('third', 'JJ') ('quarter', 'NN'))
  ('.', '.'))
>>> print example.flatten()
(S:
  ('Health-care', 'JJ')
  ('companies', 'NNS')
  ('should', 'MD')
  ('get', 'VB')
  ('healthier', 'JJR')
  ('in', 'IN')
  ('the', 'DT')
  ('third', 'JJ')
  ('quarter', 'NN')
  ('.', '.'))
```

Now we can set up a multi-stage chunk grammar, as shown in Listing 5.3. It has a stage for each of the chunk types.

Listing 8

```
cp = chunk.Regexp(r"""
NP: {<DT>?<JJ>*<NN.*>+} # noun phrase chunks
VP: {<TO>?<VB.*>}       # verb phrase chunks
PP: {<IN>}              # prepositional phrase chunks
""")

>>> example = extract(2000, conll2000.chunked())
>>> print cp.parse(example.flatten(), trace=1)
# Input:
 <JJ>  <NNS>  <MD>  <VB>  <JJR>  <IN>  <DT>  <JJ>  <NN>  <.>
# noun phrase chunks:
{<JJ>  <NNS>} <MD>  <VB>  <JJR>  <IN> {<DT>  <JJ>  <NN>} <.>
# Input:
 <NP>  <MD>  <VB>  <JJR>  <IN>  <NP>  <.>
# verb phrase chunks:
 <NP>  <MD> {<VB>} <JJR>  <IN>  <NP>  <.>
# Input:
 <NP>  <MD>  <VP>  <JJR>  <IN>  <NP>  <.>
# prepositional phrase chunks:
 <NP>  <MD>  <VP>  <JJR> {<IN>} <NP>  <.>
(S:
  (NP: ('Health-care', 'JJ') ('companies', 'NNS'))
  ('should', 'MD')
  (VP: ('get', 'VB'))
  ('healthier', 'JJR')
  (PP: ('in', 'IN'))
  (NP: ('the', 'DT') ('third', 'JJ') ('quarter', 'NN'))
  ('.', '.'))
```

5.4.6 Exercises

1. ☼ Pick one of the three chunk types in the CoNLL corpus. Inspect the CoNLL corpus and try to observe any patterns in the POS tag sequences that make up this kind of chunk. Develop a simple chunker using the regular expression chunker `chunk.Regexp`. Discuss any tag sequences that are difficult to chunk reliably.

2. ☼ An early definition of *chunk* was the material that occurs between chinks. Develop a chunker which starts by putting the whole sentence in a single chunk, and then does the rest of its work solely by chinking. Determine which tags (or tag sequences) are most likely to make up chinks with the help of your own utility program. Compare the performance and simplicity of this approach relative to a chunker based entirely on chunk rules.

3. ◑ Develop a chunker for one of the chunk types in the CoNLL corpus using a regular-expression based chunk grammar `RegexpChunk`. Use any combination of rules for chunking, chinking, merging or splitting.

4. ★ We saw in the tagging chapter that it is possible to establish an upper limit to tagging performance by looking for ambiguous n-grams, n-grams that are tagged in more than one possible way in the training data. Apply the same method to determine an upper bound on the performance of an n-gram chunker.

5. ★ Pick one of the three chunk types in the CoNLL corpus. Write functions to do the following tasks for your chosen type:

   a) List all the tag sequences that occur with each instance of this chunk type.

   b) Count the frequency of each tag sequence, and produce a ranked list in order of decreasing frequency; each line should consist of an integer (the frequency) and the tag sequence.

   c) Inspect the high-frequency tag sequences. Use these as the basis for developing a better chunker.

6. ★ The baseline chunker presented in the evaluation section tends to create larger chunks than it should. For example, the phrase: `[every/DT time/NN] [she/PRP] sees /VBZ [a/DT newspaper/NN]` contains two consecutive chunks, and our baseline chunker will incorrectly combine the first two: `[every/DT time/NN she/PRP]`. Write a program that finds which of these chunk-internal tags typically occur at the start of a chunk, then devise one or more rules that will split up these chunks. Combine these with the existing baseline chunker and re-evaluate it, to see if you have discovered an improved baseline.

7. ★ Develop an NP chunker which converts POS-tagged text into a list of tuples, where each tuple consists of a verb followed by a sequence of noun phrases and prepositions, e.g. `the little cat sat on the mat` becomes `('sat', 'on', 'NP')`...

8. ★ The Penn Treebank contains a section of tagged Wall Street Journal text which has been chunked into noun phrases. The format uses square brackets, and we have encountered it several times during this chapter. It can be accessed by importing the Treebank corpus reader (`from nltk_lite.corpora import treebank`), then iterating over its

chunked items (`for sent in treebank.chunked():`). These items are flat trees, just as we got using `conll2000.chunked()`.

a) Consult the documentation for the NLTK chunk package to find out how to generate Treebank and IOB strings from a tree. Write functions `chunk2brackets()` and `chunk2iob()` which take a single chunk tree as their sole argument, and return the required multi-line string representation.

b) Write command-line conversion utilities `bracket2iob.py` and `iob2bracket.py` that take a file in Treebank or CoNLL format (resp) and convert it to the other format. (Obtain some raw Treebank or CoNLL data from the NLTK Corpora, save it to a file, and then use `for line in open(filename)` to access it from Python.)

5.5 N-Gram Chunking

Our approach to chunking has been to try to detect structure based on the part-of-speech tags. We have seen that the IOB format represents this extra structure using another kind of tag. The question arises then, as to whether we could use the same n-gram tagging methods we saw in the last chapter, applied to a different vocabulary.

The first step is to get the `word,tag,chunk` triples from the CoNLL corpus and map these to `tag,chunk` pairs:

```
>>> from nltk_lite import tag
>>> chunk_data = [[(t,c) for w,t,c in chunk.tree2conlltags(chtree)]
...               for chtree in conll2000.chunked()]
```

5.5.1 A Unigram Chunker

Now we can train and score a **unigram chunker** on this data, just as if it was a tagger:

```
>>> unigram_chunker = tag.Unigram()
>>> unigram_chunker.train(chunk_data)
>>> print tag.accuracy(unigram_chunker, chunk_data)
0.781378851068
```

This chunker does reasonably well. Let's look at the errors it makes. Consider the opening phrase of the first sentence of the chunking data, here shown with part of speech tags:

> Confidence/NN in/IN the/DT pound/NN is/VBZ widely/RB expected/VBN to/TO take/VB another/DT sharp/JJ dive/NN

We can try the unigram chunker out on this first sentence by creating some "tokens" using `[t for t,c in chunk_data[0]]`, then running our chunker over them using `list(unigram_chunker.tag(tokens))`. The unigram chunker only looks at the tags, and tries to add chunk tags. Here is what it comes up with:

> NN/I-NP IN/B-PP DT/B-NP NN/I-NP VBZ/B-VP RB/O VBN/I-VP TO/B-PP VB/I-VP DT/B-NP JJ/I-NP NN/I-NP

Notice that it tags all instances of `NN` with `I-NP`, because nouns usually do not appear at the beginning of noun phrases in the training data. Thus, the first noun `Confidence/NN` is tagged incorrectly. However, `pound/NN` and `dive` are correctly tagged as `I-NP`; they are not in the initial position that should be tagged `B-NP`. It incorrectly tags `widely/RB` as outside `O`, and it incorrectly tags the infinitival `to/TO` as `B-PP`, as if it was a preposition starting a prepositional phrase.

5.5.2 A Bigram Chunker (incomplete)

[Why these problems might go away if we look at the previous chunk tag?]

Let's run a bigram chunker:

```
>>> bigram_chunker = tag.Bigram(backoff=unigram_chunker)
>>> bigram_chunker.train(chunk_data)
>>> print tag.accuracy(bigram_chunker, chunk_data)
0.89312652614
```

We can run the bigram chunker over the same sentence as before using `list(bigram_chunker.tag(tokens))`. Here is what it comes up with:

> NN/B-NP IN/B-PP DT/B-NP NN/I-NP VBZ/B-VP RB/I-VP VBN/I-VP TO/I-VP VB/I-VP DT/B-NP JJ/I-NP NN/I-NP

This is 100% correct.

5.5.3 Exercises

1. ◑ The bigram chunker scores about 90% accuracy. Study its errors and try to work out why it doesn't get 100% accuracy.

2. ◑ Experiment with trigram chunking. Are you able to improve the performance any more?

3. ★ An n-gram chunker can use information other than the current part-of-speech tag and the $n-1$ previous chunk tags. Investigate other models of the context, such as the $n-1$ previous part-of-speech tags, or some combination of previous chunk tags along with previous and following part-of-speech tags.

4. ★ Consider the way an n-gram tagger uses recent tags to inform its tagging choice. Now observe how a chunker may re-use this sequence information. For example, both tasks will make use of the information that nouns tend to follow adjectives (in English). It would appear that the same information is being maintained in two places. Is this likely to become a problem as the size of the rule sets grows? If so, speculate about any ways that this problem might be addressed.

5.6 Cascaded Chunkers

So far, our chunk structures have been relatively flat. Trees consist of tagged tokens, optionally grouped under a chunk node such as `NP`. However, it is possible to build chunk structures of arbitrary depth, simply by creating a multi-stage chunk grammar.

So far, our chunk grammars have consisted of a single stage: a chunk type followed by one or more patterns. However, chunk grammars can have two or more such stages. These stages are processed

in the order that they appear. The patterns in later stages can refer to a mixture of part-of-speech tags and chunk types. Listing 5.4 has patterns for noun phrases, prepositional phrases, verb phrases, and sentences. This is a four-stage chunk grammar, and can be used to create structures having a depth of at most four.

Listing 9 A Chunker that Handles NP, PP, VP and S

```
cp = chunk.Regexp(r"""
NP: {<DT|JJ|NN.*>+}          # Chunk sequences of DT, JJ, NN
PP: {<IN><NP>}               # Chunk prepositions followed by NP
VP: {<VB.*><NP|PP|S>+$}      # Chunk rightmost verbs and arguments/adjuncts
S:  {<NP><VP>}               # Chunk NP, VP
""")
tagged_tokens = [("Mary", "NN"), ("saw", "VBD"), ("the", "DT"), ("cat", "NN"),
    ("sit", "VB"), ("on", "IN"), ("the", "DT"), ("mat", "NN")]

>>> print cp.parse(tagged_tokens)
(S:
  (NP: ('Mary', 'NN'))
  ('saw', 'VBD')
  (S:
    (NP: ('the', 'DT') ('cat', 'NN'))
    (VP:
      ('sit', 'VB')
      (PP: ('on', 'IN') (NP: ('the', 'DT') ('mat', 'NN'))))))
```

Unfortunately this result misses the VP headed by *saw*. It has other shortcomings too. Let's see what happens when we apply this chunker to a sentence having deeper nesting.

```
>>> tagged_tokens = [("John", "NNP"), ("thinks", "VBZ"), ("Mary", "NN"),
...     ("saw", "VBD"), ("the", "DT"), ("cat", "NN"), ("sit", "VB"),
...     ("on", "IN"), ("the", "DT"), ("mat", "NN")]
>>> print cp.parse(tagged_tokens)
(S:
  (NP: ('John', 'NNP'))
  ('thinks', 'VBZ')
  (NP: ('Mary', 'NN'))
  ('saw', 'VBD')
  (S:
    (NP: ('the', 'DT') ('cat', 'NN'))
    (VP:
      ('sit', 'VB')
      (PP: ('on', 'IN') (NP: ('the', 'DT') ('mat', 'NN'))))))
```

The solution to these problems is to get the chunker to loop over its patterns: after trying all of them, it repeats the process. We add an optional second argument `loop` to specify the number of times the set of patterns should be run:

```
>>> cp = chunk.Regexp(grammar, loop=2)
>>> print cp.parse(tagged_tokens)
(S:
  (NP: ('John', 'NNP'))
```

```
('thinks', 'VBZ')
(S:
  (NP: ('Mary', 'NN'))
  (VP:
    ('saw', 'VBD')
    (S:
      (NP: ('the', 'DT') ('cat', 'NN'))
      (VP:
        ('sit', 'VB')
        (PP: ('on', 'IN') (NP: ('the', 'DT') ('mat', 'NN'))))))))
```

This cascading process enables us to create deep structures. However, creating and debugging a cascade is quite difficult, and there comes a point where it is more effective to do full parsing (see Chapter 7).

5.7 Conclusion

In this chapter we have explored efficient and robust methods that can identify linguistic structures in text. Using only part-of-speech information for words in the local context, a "chunker" can successfully identify simple structures such as noun phrases and verb groups. We have seen how chunking methods extend the same lightweight methods that were successful in tagging. The resulting structured information is useful in information extraction tasks and in the description of the syntactic environments of words. The latter will be invaluable as we move to full parsing.

There are a surprising number of ways to chunk a sentence using regular expressions. The patterns can add, shift and remove chunks in many ways, and the patterns can be sequentially ordered in many ways. One can use a small number of very complex rules, or a long sequence of much simpler rules. One can hand-craft a collection of rules, and one can write programs to analyze a chunked corpus to help in the development of such rules. The process is painstaking, but generates very compact chunkers that perform well and that transparently encode linguistic knowledge.

It is also possible to chunk a sentence using the techniques of n-gram tagging. Instead of assigning part-of-speech tags to words, we assign IOB tags to the part-of-speech tags. Bigram tagging turned out to be particularly effective, as it could be sensitive to the chunk tag on the previous word. This statistical approach requires far less effort than rule-based chunking, but creates large models and delivers few linguistic insights.

Like tagging, chunking cannot be done perfectly. For example, as pointed out by [Abney, 1996a], we cannot correctly analyze the structure of the sentence *I turned off the spectroroute* without knowing the meaning of *spectroroute*; is it a kind of road or a type of device? Without knowing this, we cannot tell whether *off* is part of a prepositional phrase indicating direction (tagged `B-PP`), or whether *off* is part of the verb-particle construction *turn off* (tagged `I-VP`).

A recurring theme of this chapter has been *diagnosis*. The simplest kind is manual, when we inspect the tracing output of a chunker and observe some undesirable behavior that we would like to fix. Sometimes we discover cases where we cannot hope to get the correct answer because the part-of-speech tags are too impoverished and do not give us sufficient information about the lexical item. A second approach is to write utility programs to analyze the training data, such as counting the number of times a given part-of-speech tag occurs inside and outside an NP chunk. A third approach is to evaluate the system against some gold standard data to obtain an overall performance score. We can even use this to parameterize the system, specifying which chunk rules are used on a given run, and tabulating performance for different parameter combinations. Careful use of these diagnostic methods

permits us to optimize the performance of our system. We will see this theme emerge again later in chapters dealing with other topics in natural language processing.

5.8 Further Reading

Abney's Cass system: `http://www.vinartus.net/spa/97a.pdf`

Part II

PARSING

Chapter 6

Structured Programming in Python

6.1 Introduction

In Part I you had an intensive introduction to Python (Chapter 2) followed by chapters on words, tags, and chunks (Chapters 3-5). These chapters contain many examples and exercises that should have helped you consolidate your Python skills and apply them to simple NLP tasks. So far our programs — and the data we have been processing — have been relatively unstructured. In Part II we will focus on *structure*: i.e. structured programming with structured data.

In this chapter we will review key programming concepts and explain many of the minor points that could easily trip you up. More fundamentally, we will introduce important concepts in structured programming that help you write readable, well-organized programs that you and others will be able to re-use. Each section is independent, so you can easily select what you most need to learn and concentrate on that. As before, this chapter contains many examples and exercises (and as before, some exercises introduce new material). Readers new to programming should work through them carefully and consult other introductions to programming if necessary; experienced programmers can quickly skim this chapter.

6.2 Back to the Basics

Let's begin by revisiting some of the fundamental operations and data structures required for natural language processing in Python. It is important to appreciate several finer points in order to write Python programs which are not only correct but also *idiomatic* — by this, we mean using the features of the Python language in a natural and concise way. To illustrate, here is a technique for iterating over the members of a list by initializing an index `i` and then incrementing the index each time we pass through the loop:

```
>>> sent = ['I', 'am', 'the', 'Walrus']
>>> i = 0
>>> while i < len(sent):
...     sent[i].lower()
...     i += 1
>>> sent
['i', 'am', 'the', 'walrus']
```

Although this does the job, it is not idiomatic Python. By contrast, Python's `for` statement allows us to achieve the same effect much more succinctly:

```
>>> sent = ['I', 'am', 'the', 'Walrus']
>>> for s in sent:
...     s.lower()
>>> sent
['i', 'am', 'the', 'walrus']
```

We'll start with the most innocuous operation of all: *assignment*. Then we will look at sequence types in detail.

6.2.1 Assignment

Python's assignment statement operates on *values*. But what is a value? Consider the following code fragment:

```
>>> word1 = 'Monty'
>>> word2 = word1              ①
>>> word1 = 'Python'           ②
>>> word2
'Monty'
```

This code shows that when we write `word2 = word1` in line ①, the value of `word1` (the string `'Monty'`) is assigned to `word2`. That is, `word2` is a **copy** of `word1`, so when we overwrite `word1` with a new string `'Python'` in line ②, the value of `word2` is not affected.

However, assignment statements do not always involve making copies in this way. An important subtlety of Python is that the "value" of a structured object (such as a list) is actually a *reference* to the object. In the following example, line ① assigns the reference of `list1` to the new variable `list2`. When we modify something inside `list1` on line ②, we can see that the contents of `list2` have also been changed.

```
>>> list1 = ['Monty', 'Python']
>>> list2 = list1              ①
>>> list1[1] = 'Bodkin'        ②
>>> list2
['Monty', 'Bodkin']
```

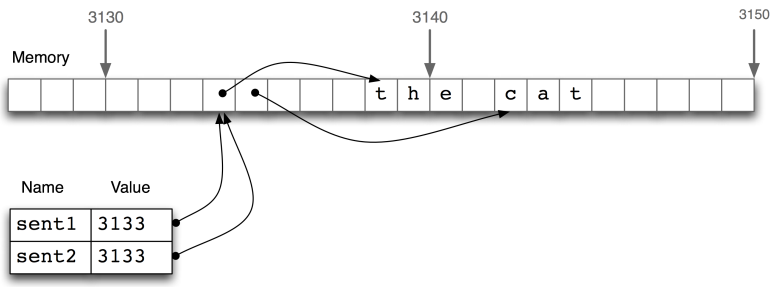


Figure 6.1: List Assignment and Computer Memory

Thus line ① does not copy the contents of the variable, only its "object reference". To understand what is going on here, we need to know how lists are stored in the computer's memory. In Figure 6.1, we see that a list `sent1` is a reference to an object stored at location 3133 (which is itself a series of pointers to other locations holding strings). When we assign `sent2 = sent1`, it is just the object reference 3133 that gets copied.

6.2.2 Sequences: Strings, Lists and Tuples

We have seen three kinds of sequence object: strings, lists, and tuples. As sequences, they have some common properties: they can be indexed and they have a length:

```
>>> string = 'I turned off the spectroroute'
>>> words = ['I', 'turned', 'off', 'the', 'spectroroute']
>>> pair = (6, 'turned')
>>> string[2], words[3], pair[1]
('t', 'the', 'turned')
>>> len(string), len(words), len(pair)
(29, 5, 2)
```

We can iterate over the items in a sequence `s` in a variety of useful ways, as shown in Table 6.1.

Python Expression	Comment
`for item in s`	iterate over the items of `s`
`for item in sorted(s)`	iterate over the items of `s` in order
`for item in set(s)`	iterate over unique elements of `s`
`for item in reversed(s)`	iterate over elements of `s` in reverse
`for item in set(s).difference(t)`	iterate over elements of `s` not in `t`

Table 6.1: Various ways to iterate over sequences

The sequence functions illustrated in Table 6.1 can be combined in various ways; for example, to get unique elements of `s` sorted in reverse, use `reversed(sorted(set(s)))`. Sometimes random order is required:

```
>>> import random
>>> random.shuffle(words)
```

We can convert between these sequence types. For example, `tuple(s)` converts any kind of sequence into a tuple, and `list(s)` converts any kind of sequence into a list. We can convert a list of strings to a single string using the `join()` function, e.g. `':'.join(words)`.

Notice in the above code sample that we computed multiple values on a single line, separated by commas. These comma-separated expressions are actually just tuples — Python allows us to omit the parentheses around tuples if there is no ambiguity. When we print a tuple, the parentheses are always displayed. By using tuples in this way, we are implicitly aggregating items together.

In the next example, we use tuples to re-arrange the contents of our list. (We can omit the parentheses because the comma has higher precedence than assignment.)

```
>>> words[2], words[3], words[4] = words[3], words[4], words[2]
>>> words
['I', 'turned', 'the', 'spectroroute', 'off']
```

This is an idiomatic and readable way to move items inside a list. It is equivalent to the following traditional way of doing such tasks that does not use tuples (notice that this method needs a temporary variable `tmp`).

```
>>> tmp = words[2]
>>> words[2] = words[3]
>>> words[3] = words[4]
>>> words[4] = tmp
```

As we have seen, Python has sequence functions such as `sorted()` and `reversed()` which rearrange the items of a sequence. There are also functions that modify the *structure* of a sequence and which can be handy for language processing. Thus, `zip()` takes the items of two sequences and 'zips' them together into a single list of pairs. Given a sequence `s`, `enumerate(s)` returns an iterator that produces a pair of an index and the item at that index.

```
>>> words = ['I', 'turned', 'off', 'the', 'spectroroute']
>>> tags = ['nnp', 'vbd', 'in', 'dt', 'nn']
>>> zip(words, tags)
[('I', 'nnp'), ('turned', 'vbd'), ('off', 'in'),
('the', 'dt'), ('spectroroute', 'nn')]
>>> list(enumerate(words))
[(0, 'I'), (1, 'turned'), (2, 'off'), (3, 'the'), (4, 'spectroroute')]
```

6.2.3 Combining different sequence types

Let's combine our knowledge of these three sequence types, together with list comprehensions, to perform the task of sorting the words in a string by their length.

```
>>> words = 'I turned off the spectroroute'.split()      ①
>>> wordlens = [(len(word), word) for word in words]     ②
>>> wordlens
[(1, 'I'), (6, 'turned'), (3, 'off'), (3, 'the'), (12, 'spectroroute')
>>> wordlens.sort()                                      ③
>>> import string
>>> string.join([word for count, word in wordlens])      ④
'I off the turned spectroroute'
```

Each of the above lines of code contains a significant feature. Line ① demonstrates that a simple string is actually an object with methods defined on it, such as `split()`. Line ② shows the construction of a list of tuples, where each tuple consists of a number (the word length) and the word, e.g. `(3, 'the')`. Line ③ sorts the list, modifying the list in-place. Finally, line ④ discards the length information then joins the words back into a single string.

We began by talking about the commonalities in these sequence types, but the above code illustrates important differences in their roles. First, strings appear at the beginning and the end: this is typical in the context where our program is reading in some text and producing output for us to read. Lists and tuples are used in the middle, but for different purposes. A list is typically a sequence of objects all having the *same type*, of *arbitrary length*. We often use lists to hold sequences of words. In contrast, a tuple is typically a collection of objects of *different types*, of *fixed length*. We often use a tuple to hold a **record**, a collection of different **fields** relating to some entity. This distinction between the use of lists and tuples takes some getting used to, so here is another example:

```
>>> lexicon = [
...     ('the', 'DT', ['Di:', 'D@']),
...     ('off', 'IN', ['Qf', 'O:f'])
... ]
```

Here, a lexicon is represented as a list because it is a collection of objects of a single type — lexical entries — of no predetermined length. An individual entry is represented as a tuple because it is a collection of objects with different interpretations, such as the orthographic form, the part of speech, and the pronunciations represented in the SAMPA computer readable phonetic alphabet. Note that these pronunciations are stored using a list. (Why?)

The distinction between lists and tuples has been described in terms of usage. However, there is a more fundamental difference: in Python, lists are **mutable**, while tuples are **immutable**. In other words, lists can be modified, while tuples cannot. Here are some of the operations on lists which do in-place modification of the list. None of these operations is permitted on a tuple, a fact you should confirm for yourself.

```
>>> lexicon.sort()
>>> lexicon[1] = ('turned', 'VBD', ['t3:nd', 't3`nd'])
>>> del lexicon[0]
```

6.2.4 Stacks and Queues

Lists are a particularly versatile data type. We can use lists to implement higher-level data types such as stacks and queues. A **stack** is a container that has a **first-in-first-out** policy for adding and removing items (see Figure 6.2).

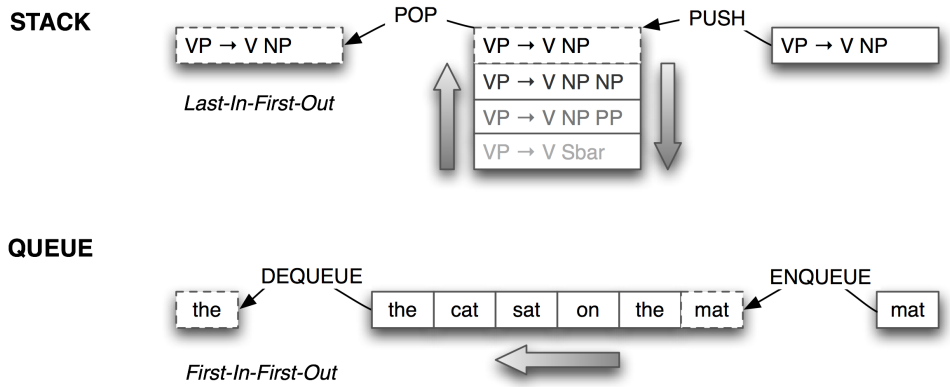


Figure 6.2: Stacks and Queues

Stacks are used to keep track of the current context in computer processing of natural languages (and programming languages too). We will seldom have to deal with stacks explicitly, as the implementation of NLTK parsers, treebank corpus readers, (and even Python functions), all use stacks behind the scenes. However, it is important to understand what stacks are and how they work.

In Python, we can treat a list as a stack by limiting ourselves to the three operations defined on stacks: `append(item)` (to push `item` onto the stack), `pop()` to pop the item off the top of the stack, and `[-1]` to access the item on the top of the stack. Listing 6.1 processes a sentence with phrase markers, and checks that the parentheses are balanced. The loop pushes material onto the stack when it gets an open parenthesis, and pops the stack when it gets a close parenthesis. We see that two are left on the stack at the end; i.e. the parentheses are not balanced.

Although Listing 6.1 is a useful illustration of stacks, it is overkill because we could have done a direct count: `phrase.count('(') == phrase.count(')')`. However, we can use stacks for more sophisticated processing of strings containing nested structure, as shown in Listing 6.2. Here we build a (potentially deeply-nested) list of lists. Whenever a token other than a parenthesis is encountered, we add it to a list at the appropriate level of nesting. The stack cleverly keeps track of this level of nesting, exploiting the fact that the item at the top of the stack is actually shared with a more deeply nested item. (Hint: add diagnostic print statements to the function to help you see what it is doing.)

Listing 10 Check parentheses are balanced

```
def check_parens(tokens):
    stack = []
    for token in tokens:
        if token == '(':      # push
            stack.append(token)
        elif token == ')':    # pop
            stack.pop()
    return stack

>>> phrase = "( the cat ) ( sat ( on ( the mat )"
>>> print check_parens(phrase.split())
['(', '(']
```

Listing 11 Convert a nested phrase into a nested list using a stack

```
def convert_parens(tokens):
    stack = [[]]
    for token in input:
        if token == '(':      # push
            sublist = []
            stack[-1].append(sublist)
            stack.append(sublist)
        elif token == ')':    # pop
            stack.pop()
        else:                 # update top of stack
            stack[-1].append(token)
    return stack[0]

>>> phrase = "( the cat ) ( sat ( on ( the mat ) ) )"
>>> print convert_parens(phrase.split())
[['the', 'cat'], ['sat', ['on', ['the', 'mat']]]]
```

Lists can be used to represent another important data structure. A **queue** is a container that has a **last-in-first-out** policy for adding and removing items (see Figure 6.2). Queues are used for scheduling activities or resources. As with stacks, we will seldom have to deal with queues explicitly, as the implementation of NLTK n-gram taggers (Section 4.6) and chart parsers (Section 8.2) use queues behind the scenes. However, we will take a brief look at how queues are implemented using lists.

```
>>> queue = ['the', 'cat', 'sat']
>>> queue.append('on')
>>> queue.append('the')
>>> queue.append('mat')
>>> queue.pop(0)
'the'
>>> queue.pop(0)
'cat'
>>> queue
['sat', 'on', 'the', 'mat']
```

6.2.5 More List Comprehensions

You may recall that in Chapter 3, we introduced list comprehensions, with examples like the following:

```
>>> sent = ['The', 'dog', 'gave', 'John', 'the', 'newspaper']
>>> [word.lower() for word in sent]
['the', 'dog', 'gave', 'john', 'the', 'newspaper']
```

List comprehensions are a convenient and readable way to express list operations in Python, and they have a wide range of uses in natural language processing. In this section we will see some more examples. The first of these takes successive overlapping slices of size `n` (a **sliding window**) from a list (pay particular attention to the range of the variable `i`).

```
>>> sent = ['The', 'dog', 'gave', 'John', 'the', 'newspaper']
>>> n = 3
>>> [sent[i:i+n] for i in range(len(sent)-n+1)]
[['The', 'dog', 'gave'],
 ['dog', 'gave', 'John'],
 ['gave', 'John', 'the'],
 ['John', 'the', 'newspaper']]
```

You can also use list comprehensions for a kind of multiplication. Here we generate all combinations of two determiners, two adjectives, and two nouns. The list comprehension is split across three lines for readability.

```
>>> [(dt,jj,nn) for dt in ('two', 'three')
...             for jj in ('old', 'blind')
...             for nn in ('men', 'mice')]
[('two', 'old', 'men'), ('two', 'old', 'mice'), ('two', 'blind', 'men'),
 ('two', 'blind', 'mice'), ('three', 'old', 'men'), ('three', 'old', 'mice'),
 ('three', 'blind', 'men'), ('three', 'blind', 'mice')]
```

The above example contains three independent `for` loops. These loops have no variables in common, and we could have put them in any order. We can also have **nested loops** with shared variables. The next example iterates over all sentences in a section of the Brown Corpus, and for each sentence, iterates over each word.

```
>>> from nltk_lite.corpora import brown
>>> [word for sent in brown.raw('a')
...      for word in sent
...      if len(word) == 17]
['September-October', 'Sheraton-Biltmore', 'anti-organization',
 'anti-organization', 'Washington-Oregon', 'York-Pennsylvania',
 'misunderstandings', 'Sheraton-Biltmore', 'neo-stagnationist',
 'cross-examination', 'bronzy-green-gold', 'Oh-the-pain-of-it',
 'Secretary-General', 'Secretary-General', 'textile-importing',
 'textile-exporting', 'textile-producing', 'textile-producing']
```

As you will see, the list comprehension in this example contains a final `if` clause which allows us to filter out any words that fail to meet the specified condition.

Another way to use loop variables is to ignore them! This is the standard method for building multidimensional structures. For example, to build an array with *m* rows and *n* columns, where each cell is a set, we would use a nested list comprehension, as shown in line ① below. Observe that the loop variables `i` and `j` are not used anywhere in the expressions preceding the `for` clauses.

```
>>> from pprint import pprint
>>> m, n = 3, 7
>>> array = [[set() for i in range(n)] for j in range(m)]      ①
>>> array[2][5].add('foo')
>>> pprint(array)
[[set([]), set([]), set([]), set([]), set([]), set([]), set([])],
 [set([]), set([]), set([]), set([]), set([]), set([]), set([])],
 [set([]), set([]), set([]), set([]), set([]), set(['foo']), set([])]]
```

Sometimes we use a list comprehension as part of a larger aggregation task. In the following example we calculate the average length of words in part of the Brown Corpus. Notice that we don't bother storing the list comprehension in a temporary variable, but use it directly as an argument to the `average()` function.

```
>>> from numpy import average
>>> average([len(word) for sent in brown.raw('a') for word in sent])
4.40154543827
```

Now that we have reviewed the sequence types, we have one more fundamental data type to revisit.

6.2.6 Dictionaries

As you have already seen, the dictionary data type can be used in a variety of language processing tasks (e.g. Section 2.6). However, we have only scratched the surface. Dictionaries have many more applications than you might have imagined.

> **Note**
>
> The dictionary data type is often known by the name **associative array**. A normal array maps from integers (the keys) to arbitrary data types (the values), while an associative array places no such constraint on keys. Keys can be strings, tuples, or other more deeply nested structure. Python places the constraint that keys must be *immutable*.

Let's begin by comparing dictionaries with tuples. Tuples allow *access by position*; to access the part-of-speech of the following lexical entry we just have to know it is found at index position `1`. However, dictionaries allow *access by name*:

```
>>> lexical_entry = ('turned', 'VBD', ['t3:nd', 't3`nd'])
>>> lexical_entry[1]
'VBD'
>>> entry_dict = {'lexeme':'turned', 'pos':'VBD', 'pron':['t3:nd', 't3`nd']}
>>> entry_dict['pos']
'VBD'
```

In this case, dictionaries are little more than a convenience. We can even simulate access by name using well-chosen constants, e.g.:

```
>>> LEXEME = 0; POS = 1; PRON = 2
>>> entry_tuple[POS]
'VBD'
```

This method works when there is a closed set of keys and the keys are known in advance. Dictionaries come into their own when we are mapping from an open set of keys, which happens when the keys are drawn from an unrestricted vocabulary or where they are generated by some procedure. Listing 6.3 illustrates the first of these. The function `mystery()` begins by initializing a dictionary called `groups`, then populates it with words. We leave it as an exercise for the reader to work out what this function computes. For now, it's enough to note that the keys of this dictionary are an open set, and it would not be feasible to use a integer keys, as would be required if we used lists or tuples for the representation.

Listing 12 Mystery program

```
from string import join
def mystery(input):
    groups = {}
    for word in input:
        key = join(sorted(list(word)), '')
        if key not in groups:                    ①
            groups[key] = set()                  ②
        groups[key].add(word)                    ③
    return sorted(join(sorted(v)) for v in groups.values() if len(v) > 1)

>>> from nltk_lite.corpora import words
>>> text = words.raw()
>>> print mystery(text)
```

Listing 6.3 illustrates two important idioms, which we already touched on in Chapter 2. First, dictionary keys are unique; in order to store multiple items in a single entry we define the value to be a list or a set, and simply update the value each time we want to store another item (line ③). Second, if a key does not yet exist in a dictionary (line ①) we must explicitly add it and give it an initial value (line ②).

The second important use of dictionaries is for mappings that involve **compound keys**. Suppose we want to categorize a series of linguistic observations according to two or more properties. We can combine the properties using a tuple and build up a dictionary in the usual way, as exemplified in Listing 6.4.

Listing 13 Illustration of compound keys

```
from nltk_lite.corpora import ppattach
attachment = {}
V, N = 0, 1
for entry in ppattach.dictionary('training'):
    key = verb, prep
    if key not in attachment:
        attachment[key] = [0,0]
    if entry['attachment'] == 'V':
        attachment[key][V] += 1
    else:
        attachment[key][N] += 1
```

6.2.7 Exercises

1. ☼ Find out more about sequence objects using Python's help facility. In the interpreter, type `help(str)`, `help(list)`, and `help(tuple)`. This will give you a full list of the functions supported by each type. Some functions have special names flanked with underscore; as the help documentation shows, each such function corresponds to something more familiar. For example `x.__getitem__(y)` is just a long-winded way of saying `x[y]`.

2. ☼ Identify three operations that can be performed on both tuples and lists. Identify three list operations that cannot be performed on tuples. Name a context where using a list instead of a tuple generates a Python error.

3. ☼ Find out how to create a tuple consisting of a single item. There are at least two ways to do this.

4. ☼ Create a list `words = ['is', 'NLP', 'fun', '?']`. Use a series of assignment statements (e.g. `words[1] = words[2]`) and a temporary variable `tmp` to transform this list into the list `['NLP', 'is', 'fun', '!']`. Now do the same transformation using tuple assignment.

5. ☼ Does the method for creating a sliding window of n-grams behave correctly for the two limiting cases: $n = 1$, and $n =$ `len(sent)`?

6. ◑ Create a list of words and store it in a variable `sent1`. Now assign `sent2 = sent1`. Modify one of the items in `sent1` and verify that `sent2` has changed.

   a) Now try the same exercise but instead assign `sent2 = sent1[:]`. Modify `sent1` again and see what happens to `sent2`. Explain.

   b) Now define `text1` to be a list of lists of strings (e.g. to represent a text consisting of multiple sentences. Now assign `text2 = text1[:]`, assign a new value to one of the words, e.g. `text1[1][1] = 'Monty'`. Check what this did to `text2`. Explain.

   c) Load Python's `deepcopy()` function (i.e. `from copy import deepcopy`), consult its documentation, and test that it makes a fresh copy of any object.

7. ◑ Write code which starts with a string of words and results in a new string consisting of the same words, but where the first word swaps places with the second, and so on. For example, `'the cat sat on the mat'` will be converted into `'cat the on sat mat the'`.

8. ◑ Initialize an *n*-by-*m* list of lists of empty strings using list multiplication, e.g. `word_table = [['']*n]*m`. What happens when you set one of its values, e.g. `word_table[1][2] = "hello"`? Explain why this happens. Now write an expression using `range()` to construct a list of lists, and show that it does not have this problem.

9. ◑ Write code to initialize a two-dimensional array of sets called `word_vowels` and process a list of words, adding each word to `word_vowels[l][v]` where `l` is the length of the word and `v` is the number of vowels it contains.

10. ◑ Write code that builds a dictionary of dictionaries of sets.

11. ◑ Use `sorted()` and `set()` to get a sorted list of tags used in the Brown corpus, removing duplicates.

12. ★ Extend the example in Listing 6.4 in the following ways:

    a) Define two sets `verbs` and `preps`, and add each verb and preposition as they are encountered. (Note that you can add an item to a set without bothering to check whether it is already present.)

    b) Create nested loops to display the results, iterating over verbs and prepositions in sorted order. Generate one line of output per verb, listing prepositions and attachment ratios as follows: `raised: about 0:3, at 1:0, by 9:0, for 3:6, from 5:0, in 5:5...`

    c) We used a tuple to represent a compound key consisting of two strings. However, we could have simply concatenated the strings, e.g. `key = verb + ":" + prep`, resulting in a simple string key. Why is it better to use tuples for compound keys?

6.3 Presenting Results

Often we write a program to report a single datum, such as a particular element in a corpus that meets some complicated criterion, or a single summary statistic such as a word-count or the performance of a tagger. More often, we write a program to produce a structured result, such as a tabulation of numbers or linguistic forms, or a reformatting of the original data. When the results to be presented are linguistic, textual output is usually the most natural choice. However, when the results are numerical, it may be preferrable to produce graphical output. In this section you will learn about a variety of ways to present program output.

6.3.1 Strings and Formats

We have seen that there are two ways to display the contents of an object:

```
>>> word = 'cat'
>>> sentence = """hello
... world"""
>>> print word
cat
>>> print sentence
hello
world
>>> word
'cat'
>>> sentence
'hello\nworld'
```

The `print` command yields Python's attempt to produce the most human-readable form of an object. The second method — naming the variable at a prompt — shows us a string that can be used to recreate this object. It is important to keep in mind that both of these are just strings, displayed for the benefit of you, the user. They do not give us any clue as to the actual internal representation of the object.

There are many other useful ways to display an object as a string of characters. This may be for the benefit of a human reader, or because we want to **export** our data to a particular file format for use in an external program.

Formatted output typically contains a combination of variables and pre-specified strings, e.g. given a dictionary `wordcount` consisting of words and their frequencies we could do:

```
>>> wordcount = {'cat':3, 'dog':4, 'snake':1}
>>> for word in wordcount:
...     print word, '->', wordcount[word], ';',
dog -> 4 ; cat -> 3 ; snake -> 1
```

Apart from the problem of unwanted whitespace, print statements that contain alternating variables and constants can be difficult to read and maintain. A better solution is to use print formatting strings:

```
>>> for word in wordcount:
...     print '%s->%d;' % (word, wordcount[word]),
dog->4; cat->3; snake->1
```

6.3.2 Lining things up

So far our formatting strings have contained specifications of fixed width, such as `%6s`, a string that is padded to width 6 and right-justified. We can include a minus sign to make it left-justified. In case we don't know in advance how wide a displayed value should be, the width value can be replaced with a star in the formatting string, then specified using a variable:

```
>>> '%6s' % 'dog'
'   dog'
>>> '%-6s' % 'dog'
'dog   '
>>> width = 6
>>> '%-*s' % (width, 'dog')
'dog   '
```

Other control characters are used for decimal integers and floating point numbers. Since the percent character `%` has a special interpretation in formatting strings, we have to precede it with another `%` to get it in the output:

```
>>> "accuracy for %d words: %2.4f%%" % (9375, 100.0 * 3205/9375)
'accuracy for 9375 words: 34.1867%'
```

An important use of formatting strings is for tabulating data. The program in Listing 6.5 iterates over five genres of the Brown Corpus. For each token having the `md` tag we increment a count. To do this we have used `ConditionalFreqDist()`, where the condition is the current genre and the event is the modal, i.e. this constructs a frequency distribution of the modal verbs in each genre. Line ① identifies a small set of modals of interest, and calls the function `tabulate()` which processes the data structure to output the required counts. Note that we have been careful to separate the language processing from the tabulation of results.

There are some interesting patterns in the table produced by Listing 6.5. For instance, compare the rows for government literature and adventure literature; the former is dominated by the use of `can , may, must, will` while the latter is characterised by the use of `could` and `might`. With some further work it might be possible to guess the genre of a new text automatically, according to its distribution of modals.

Our next example, in Listing 6.6, generates a concordance display. We use the left/right justification of strings and the variable width to get vertical alignment of a variable-width window.

[TODO: explain ValueError exception]

6.3.3 Writing results to a file

We have seen how to read text from files (Section 3.2.1). It is often useful to write output to files as well. The following code opens a file `output.txt` for writing, and saves the program output to the file.

```
>>> from nltk_lite.corpora import genesis
>>> file = open('output.txt', 'w')
>>> words = set(genesis.raw())
>>> words.sort()
>>> for word in words:
...     file.write(word + "\n")
```

When we write non-text data to a file we must convert it to a string first. We can do this conversion using formatting strings, as we saw above. We can also do it using Python's backquote notation, which converts any object into a string. Let's write the total number of words to our file, before closing it.

```
>>> len(words)
4408
>>> `len(words)`
'4408'
>>> file.write(`len(words)` + "\n")
>>> file.close()
```

6.3.4 Graphical presentation

So far we have focussed on textual presentation and the use of formatted print statements to get output lined up in columns. It is often very useful to display numerical data in graphical form, since this often

Listing 14 Frequency of Modals in Different Sections of the Brown Corpus

```
from nltk_lite.probability import ConditionalFreqDist
def count_words_by_tag(t, genres):
    cfdist = ConditionalFreqDist()
    for genre in genres:                          # each genre
        for sent in brown.tagged(genre):          # each sentence
            for (word,tag) in sent:               # each tagged token
                if tag == t:                      # found a word tagged t
                    cfdist[genre].inc(word.lower())
    return cfdist

def tabulate(cfdist, words):
    print '%-18s' % 'Genre', string.join([('%6s' % w) for w in words])
    for genre in cfdist.conditions():               # for each genre
        print '%-18s' % brown.item_name[genre],     # print row heading
        for w in words:                             # for each word
            print '%6d' % cfdist[genre].count(w),   # print table cell
        print                                       # end the row

>>> genres = ['a', 'd', 'e', 'h', 'n']
>>> cfdist = count_words_by_tag('md', genres)
>>> modals = ['can', 'could', 'may', 'might', 'must', 'will']    ①
>>> tabulate(cfdist, modals)
Genre                 can  could    may  might   must   will
press: reportage       94     86     66     36     50    387
skill and hobbies     273     59    130     22     83    259
religion               84     59     79     12     54     64
miscellaneous: gov    115     37    152     13     99    237
fiction: adventure     48    154      6     58     27     48
```

Listing 15 Simple Concordance Display

```
def concordance(word, context):
    "Generate a concordance for the word with the specified context window"
    import string
    for sent in brown.raw('a'):
        try:
            pos = sent.index(word)
            left = string.join(sent[:pos])
            right = string.join(sent[pos+1:])
            print '%*s %s %-*s' %\
                (context, left[-context:], word, context, right[:context])
        except ValueError:
            pass

>>> concordance('line', 32)
ce , is today closer to the NATO line .
n more activity across the state line in Massachusetts than in Rhode I
 , gained five yards through the line and then uncorked a 56-yard touc
                `` Our interior line and out linebackers played excep
k then moved Cooke across with a line drive to left .
chal doubled down the rightfield line and Cooke singled off Phil Shart
              -- Billy Gardner's line double , which just eluded the d
           -- Nick Skorich , the line coach for the football champion
                    Maris is in line for a big raise .
uld be impossible to work on the line until then because of the large
         Murray makes a complete line of ginning equipment except for
    The company sells a complete line of gin machinery all over the co
tter Co. of Sherman makes a full line of gin machinery and equipment .
fred E. Perlman said Tuesday his line would face the threat of bankrup
 sale of property disposed of in line with a plan of liquidation .
 little effort spice up any chow line .
es , filed through the cafeteria line .
l be particularly sensitive to a line between first and second class c
A skilled worker on the assembly line , for example , earns $37 a week
```

makes it easier to detect patterns. For example, in Listing 6.5 we saw a table of numbers showing the frequency of particular modal verbs in the Brown Corpus, classified by genre. In Listing 6.7 we present the same information in graphical format. The output is shown in Figure 6.3 (a color figure in the online version).

Note

Listing 6.7 uses the PyLab package which supports sophisticated plotting functions with a MATLAB-style interface. For more information about this package please see `http://matplotlib.sourceforge.net/`.

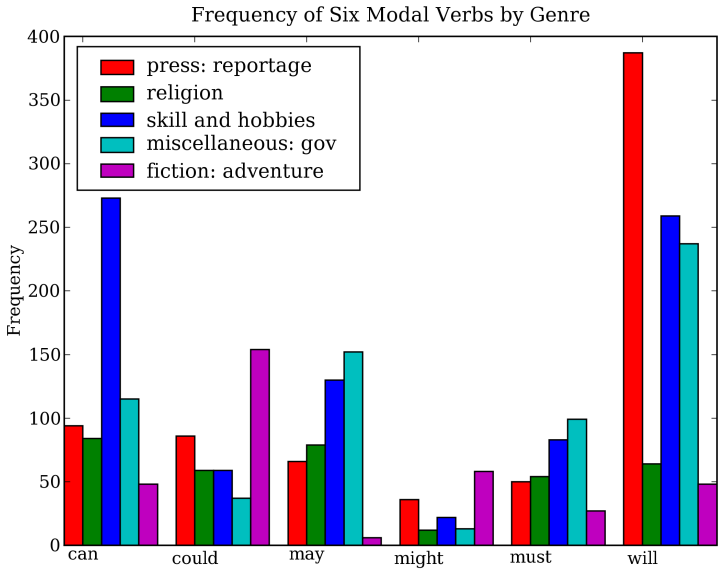


Figure 6.3: Bar Chart Showing Frequency of Modals in Different Sections of Brown Corpus

From the bar chart it is immediately obvious that *may* and *must* have almost identical relative frequencies. The same goes for *could* and *might*.

6.3.5 Exercises

1. ☼ Write code that removes whitespace at the beginning and end of a string, and normalizes whitespace between words to be a single space character.

   1) do this task using `split()` and `join()`

   2) do this task using regular expression substitutions

2. ☼ What happens when the formatting strings `%6s` and `%-6s` are used to display strings that are longer than six characters?

3. ☼ We can use a dictionary to specify the values to be substituted into a formatting string. Read Python's library documentation for formatting strings (http://docs.python.org/lib/typesseq-strings.html), and use this method to display today's date in two different formats.

Listing 16 Frequency of Modals in Different Sections of the Brown Corpus

```
from nltk_lite.corpora import brown
colors = 'rgbcmyk' # red, green, blue, cyan, magenta, yellow, black
def bar_chart(categories, words, counts):
    "Plot a bar chart showing counts for each word by category"
    import pylab
    ind = pylab.arange(len(words))
    width = 1.0 / (len(categories) + 1)
    bar_groups = []
    for c in range(len(categories)):
        bars = pylab.bar(ind+c*width, counts[categories[c]], width, color=colors[c % len(c
        bar_groups.append(bars)
    pylab.xticks(ind+width, words)
    pylab.legend([b[0] for b in bar_groups], [brown.item_name[c][:18] for c in categories]
    pylab.ylabel('Frequency')
    pylab.title('Frequency of Six Modal Verbs by Genre')
    pylab.show()

>>> genres = ['a', 'd', 'e', 'h', 'n']
>>> cfdist = count_words_by_tag('md', genres)
>>> modals = ['can', 'could', 'may', 'might', 'must', 'will']
>>> counts = {}
>>> for genre in genres:
...     counts[genre] = [cfdist[genre].count(word) for word in modals]
>>> bar_chart(genres, modals, counts)
```

4. ◑ Listing 4.4 in Chapter 4 plotted a curve showing change in the performance of a lookup tagger as the model size was increased. Plot the performance curve for a unigram tagger, as the amount of training data is varied.

6.4 Functions

Once you have been programming for a while, you will find that you need to perform a task that you have done in the past. In fact, over time, the number of completely novel things you have to do in creating a program decreases significantly. Half of the work may involve simple tasks that you have done before. Thus it is important for your code to be *re-usable*. One effective way to do this is to abstract commonly used sequences of steps into a **function**, as we briefly saw in Chapter 2.

For example, suppose we find that we often want to read text from an HTML file. This involves several steps: opening the file, reading it in, normalizing whitespace, and stripping HTML markup. We can collect these steps into a function, and give it a name such as `get_text()`:

Listing 17 Read text from a file

```
import re
def get_text(file):
    """Read text from a file, normalizing whitespace
    and stripping HTML markup."""
    text = open(file).read()
    text = re.sub('\s+', ' ', text)
    text = re.sub(r'<.*?>', ' ', text)
    return text
```

Now, any time we want to get cleaned-up text from an HTML file, we can just call `get_text()` with the name of the file as its only argument. It will return a string, and we can assign this to a variable, e.g.: `contents = get_text("test.html")`. Each time we want to use this series of steps we only have to call the function.

Notice that a function consists of the keyword `def` (short for "define"), followed by the function name, followed by a sequence of parameters enclosed in parentheses, then a colon. The following lines contain an indented block of code, the **function body**.

Using functions has the benefit of saving space in our program. More importantly, our choice of name for the function helps make the program *readable*. In the case of the above example, whenever our program needs to read cleaned-up text from a file we don't have to clutter the program with four lines of code, we simply need to call `get_text()`. This naming helps to provide some "semantic interpretation" — it helps a reader of our program to see what the program "means".

Notice that the above function definition contains a string. The first string inside a function definition is called a **docstring**. Not only does it document the purpose of the function to someone reading the code, it is accessible to a programmer who has loaded the code from a file:

```
>>> help(get_text)
get_text(file)
    Read text from a file, normalizing whitespace and stripping HTML markup
```

We have seen that functions help to make our work reusable and readable. They also help make it *reliable*. When we re-use code that has already been developed and tested, we can be more confident

that it handles a variety of cases correctly. We also remove the risk that we forget some important step, or introduce a bug. The program which calls our function also has increased reliability. The author of that program is dealing with a shorter program, and its components behave transparently.

- [More: overview of section]

6.4.1 Function arguments

- multiple arguments
- named arguments
- default values

Python is a **dynamically typed** language. It does not force us to declare the type of a variable when we write a program. This feature is often useful, as it permits us to define functions that are flexible about the type of their arguments. For example, a tagger might expect a sequence of words, but it wouldn't care whether this sequence is expressed as a list, a tuple, or an iterator.

However, often we want to write programs for later use by others, and want to program in a defensive style, providing useful warnings when functions have not been invoked correctly. Observe that the `tag()` function in Listing 6.9 behaves sensibly for string arguments, but that it does not complain when it is passed a dictionary.

Listing 18 A tagger which tags anything

```
def tag(word):
    if word in ['a', 'the', 'all']:
        return 'dt'
    else:
        return 'nn'

>>> tag('the')
'dt'
>>> tag('dog')
'nn'
>>> tag({'lexeme':'turned', 'pos':'VBD', 'pron':['t3:nd', 't3`nd']})
'nn'
```

It would be helpful if the author of this function took some extra steps to ensure that the `word` parameter of the `tag()` function is a string. A naive approach would be to check the type of the argument and return a diagnostic value, such as Python's special empty value, `None`, as shown in Listing 6.10.

However, this approach is dangerous because the calling program may not detect the error, and the diagnostic return value may be propagated to later parts of the program with unpredictable consequences. A better solution is shown in Listing 6.11.

This produces an error that cannot be ignored, since it halts program execution. Additionally, the error message is easy to interpret. (We will see an even better approach, known as "duck typing" in Chapter 10.)

Another aspect of defensive programming concerns the return statement of a function. In order to be confident that all execution paths through a function lead to a return statement, it is best to have

Listing 19 A tagger which only tags strings

```
def tag(word):
    if not type(word) is str:
        return None
    if word in ['a', 'the', 'all']:
        return 'dt'
    else:
        return 'nn'
```

Listing 20 A tagger which generates an error message when not passed a string

```
def tag(word):
    if not type(word) is str:
        raise ValueError, "argument to tag() must be a string"
    if word in ['a', 'the', 'all']:
        return 'dt'
    else:
        return 'nn'
```

a single return statement at the end of the function definition. This approach has a further benefit: it makes it more likely that the function will only return a single type. Thus, the following version of our `tag()` function is safer:

```
>>> def tag(word):
...     result = 'nn'                          # default value, a string
...     if word in ['a', 'the', 'all']:        # in certain cases...
...         result = 'dt'                      #   overwrite the value
...     return result                          # all paths end here
```

A return statement can be used to pass multiple values back to the calling program, by packing them into a tuple. Here we define a function that returns a tuple consisting of the average word length of a sentence, and the inventory of letters used in the sentence. It would have been clearer to write two separate functions.

Of course, functions do not need to have a return statement at all. Some functions do their work as a side effect, printing a result, modifying a file, or updating the contents of a parameter to the function. Consider the following three sort functions; the last approach is dangerous because a programmer could use it without realizing that it had modified its input.

```
>>> def my_sort1(l):       # good: modifies its argument, no return value
...     l.sort()
>>> def my_sort2(l):       # good: doesn't touch its argument, returns value
...     return sorted(l)
>>> def my_sort3(l):       # bad: modifies its argument and also returns it
...     l.sort()
...     return l
```

6.4.2 An Important Subtlety

Back in Section 6.2.1 you saw that in Python, assignment works on values, but that the value of a structured object is a reference to that object. The same is true for functions. Python interprets function

parameters as values (this is known as **call-by-value**). Consider Listing 6.12. Function `set_up()` has two parameters, both of which are modified inside the function. We begin by assigning an empty string to `w` and an empty dictionary to `p`. After calling the function, `w` is unchanged, while `p` is changed:

Listing 21

```
def set_up(word, properties):
    word = 'cat'
    properties['pos'] = 'noun'

>>> w = ''
>>> p = {}
>>> set_up(w, p)
>>> w
''
>>> p
{'pos': 'noun'}
```

To understand why `w` was not changed, it is necessary to understand call-by-value. When we called `set_up(w, p)`, the value of `w` (an empty string) was assigned to a new variable `word`. Inside the function, the value of `word` was modified. However, that had no affect on the external value of `w`. This parameter passing is identical to the following sequence of assignments:

```
>>> w = ''
>>> word = w
>>> word = 'cat'
>>> w
''
```

In the case of the structured object, matters are quite different. When we called `set_up(w, p)`, the value of `p` (an empty dictionary) was assigned to a new local variable `properties`. Since the value of `p` is an object reference, both variables now reference the same memory location. Modifying something inside `properties` will also change `p`, just as if we had done the following sequence of assignments:

```
>>> p = {}
>>> properties = p
>>> properties['pos'] = 'noun'
>>> p
{'pos': 'noun'}
```

Thus, to understand Python's call-by-value parameter passing, it is enough to understand Python's assignment operation. We will address some closely related issues in our later discussion of variable scope (Section 10.1.1).

6.4.3 Functional Decomposition

Well-structured programs usually make extensive use of functions. When a block of program code grows longer than 10-20 lines, it is a great help to readability if the code is broken up into one or more functions, each one having a clear purpose. This is analogous to the way a good essay is divided into paragraphs, each expressing one main idea.

Functions provide an important kind of *abstraction*. They allow us to group multiple actions into a single, complex action, and associate a name with it. (Compare this with the way we combine the actions of *go* and *bring back* into a single more complex action *fetch*.) When we use functions, the main program can be written at a higher level of abstraction, making its structure transparent, e.g.

```
>>> data = load_corpus()
>>> results = analyze(data)
>>> present(results)
```

Appropriate use of functions makes programs more readable and maintainable. Additionally, it becomes possible to reimplement a function — replacing the function's body with more efficient code — without having to be concerned with the rest of the program.

Consider the `freq_words` function in Listing 6.13. It updates the contents of a frequency distribution that is passed in as a parameter, and it also prints a list of the *n* most frequent words.

Listing 22

```
def freq_words(url, freqdist, n):
    from nltk_lite.corpora import web
    for word in web.raw(url):
        freqdist.inc(word.lower())
    print freqdist.sorted_samples()[:n]

>>> constitution = "http://www.archives.gov/national-archives-experience/charters/c
>>> from nltk_lite.probability import FreqDist
>>> fd = FreqDist()
>>> freq_words(constitution, fd, 20)
['the', ',', 'of', 'and', 'shall', '.', 'be', 'to', 'in', 'states', 'or',
';', 'united', 'a', 'state', 'by', 'for', 'any', 'president', 'which']
```

This function has a number of problems. The function has two side-effects: it modifies the contents of its second parameter, and it prints a selection of the results it has computed. The function would be easier to understand and to reuse elsewhere if we initialize the `FreqDist()` object inside the function (in the same place it is populated), and if we moved the selection and display of results to the calling program. In Listing 6.14 we **refactor** this function, and simplify its interface by providing a single `url` parameter.

6.4.4 Documentation (notes)

- some guidelines for literate programming (e.g. variable and function naming)
- documenting functions (user-level and developer-level documentation)

6.4.5 Functions as Arguments

So far the arguments we have passed into functions have been simple objects like strings, or structured objects like lists. These arguments allow us to parameterise the behavior of a function. As a result, functions are very flexible and powerful abstractions, permitting us to repeatedly apply the *same operation* on *different data*. Python also lets us pass a function as an argument to another function. Now

Listing 23

```
def freq_words(url):
    from nltk_lite.corpora import web
    from nltk_lite.probability import FreqDist
    freqdist = FreqDist()
    for word in web.raw(url):
        freqdist.inc(word.lower())
    return freqdist

>>> fd = freq_words(constitution)
>>> print fd.sorted_samples()[:20]
['the', ',', 'of', 'and', 'shall', '.', 'be', 'to', 'in', 'states', 'or',
';', 'united', 'a', 'state', 'by', 'for', 'any', 'president', 'which']
```

we can abstract out the operation, and apply a *different operation* on the *same data*. As the following examples show, we can pass the built-in function `len()` or a user-defined function `last_letter()` as parameters to another function:

```
>>> def extract_property(prop):
...     words = ['The', 'dog', 'gave', 'John', 'the', 'newspaper']
...     return [prop(word) for word in words]
>>> extract_property(len)
[3, 3, 4, 4, 3, 9]
>>> def last_letter(word):
...     return word[-1]
>>> extract_property(last_letter)
['e', 'g', 'e', 'n', 'e', 'r']
```

Surprisingly, `len` and `last_letter` are objects that can be passed around like lists and dictionaries. Notice that parentheses are only used after a function name if we are invoking the function; when we are simply passing the function around as an object these are not used.

Python provides us with one more way to define functions as arguments to other functions, so-called **lambda expressions**. Supposing there was no need to use the above `last_letter()` function in multiple places, we can equivalently write the following:

```
>>> extract_property(lambda w: w[-1])
['e', 'g', 'e', 'n', 'e', 'r']
```

Our next example illustrates passing a function to the `sorted()` function. When we call the latter with a single argument (the list to be sorted), it uses the built-in lexicographic comparison function `cmp()`. However, we can supply our own sort function, e.g. to sort by decreasing length.

```
>>> words = 'I turned off the spectroroute'.split()
>>> sorted(words)
['I', 'off', 'spectroroute', 'the', 'turned']
>>> sorted(words, cmp)
['I', 'off', 'spectroroute', 'the', 'turned']
>>> sorted(words, lambda x, y: cmp(len(y), len(x)))
['spectroroute', 'turned', 'off', 'the', 'I']
```

In 6.2.5 we saw an example of filtering out some items in a list comprehension, using an `if` test. Similarly, we can restrict a list to just the lexical words, using `[word for word in sent if is_lexical(word)]`. This is a little cumbersome as it mentions the `word` variable three times. A more compact way to express the same thing is as follows.

```
>>> def is_lexical(word):
...     return word.lower() not in ('a', 'an', 'the', 'that', 'to')
>>> sent = ['The', 'dog', 'gave', 'John', 'the', 'newspaper']
>>> filter(is_lexical, sent)
['dog', 'gave', 'John', 'newspaper']
```

The function `is_lexical(word)` returns `True` just in case `word`, when normalized to lowercase, is not in the given list. This function is itself used as an argument to `filter()`; in Python, functions are just another kind of object that can be passed around a program; we will return to this in Section 6.4.5. The `filter()` function applies its first argument (a function) to each item of its second (a sequence), only passing it through if the function returns true for that item. Thus `filter(f, seq)` is equivalent to `[item for item in seq if apply(f,item) == True]`.

Another helpful function, which like `filter()` applies a function to a sequence, is `map()`. Here is a simple way to find the average length of a sentence in a section of the Brown Corpus:

```
>>> average(map(len, brown.raw('a')))
21.7461072664
```

Instead of `len()`, we could have passed in any other function we liked:

```
>>> sent = ['The', 'dog', 'gave', 'John', 'the', 'newspaper']
>>> def is_vowel(letter):
...     return letter in "AEIOUaeiou"
>>> def vowelcount(word):
...     return len(filter(is_vowel, word))
>>> map(vowelcount, sent)
[1, 1, 2, 1, 1, 3]
```

Instead of using `filter()` to call a named function `is_vowel`, we can define a lambda expression as follows:

```
>>> map(lambda w: len(filter(lambda c: c in "AEIOUaeiou", w)), sent)
[1, 1, 2, 1, 1, 3]
```

6.4.6 Exercises

1. ☼ Review the answers that you gave for the exercises in 6.2, and rewrite the code as one or more functions.

2. ◑ In this section we saw examples of some special functions such as `filter()` and `map()`. Other functions in this family are `zip()` and `reduce()`. Find out what these do, and write some code to try them out. What uses might they have in language processing?

3. ◑ Write a function that takes a list of words (containing duplicates) and returns a list of words (with no duplicates) sorted by decreasing frequency. E.g. if the input list contained 10 instances of the word `table` and 9 instances of the word `chair`, then `table` would appear before `chair` in the output list.

4. ◑ As you saw, `zip()` combines two lists into a single list of pairs. What happens when the lists are of unequal lengths? Define a function `myzip()` which does something different with unequal lists.

5. ◑ Import the `itemgetter()` function from the `operator` module in Python's standard library (i.e. `from operator import itemgetter`). Create a list `words` containing several words. Now try calling: `sorted(words, key=itemgetter(1))`, and `sorted(words, key=itemgetter(-1))`. Explain what `itemgetter()` is doing.

6.5 Algorithm Design Strategies

A major part of algorithmic problem solving is selecting or adapting an appropriate algorithm for the problem at hand. Whole books are written on this topic (e.g. [Levitin, 2004]) and we only have space to introduce some key concepts and elaborate on the approaches that are most prevalent in natural language processing.

The best known strategy is known as **divide-and-conquer**. We attack a problem of size *n* by dividing it into two problems of size *n/2*, solve these problems, and combine their results into a solution of the original problem. Figure 6.4 illustrates this approach for sorting a list of words.

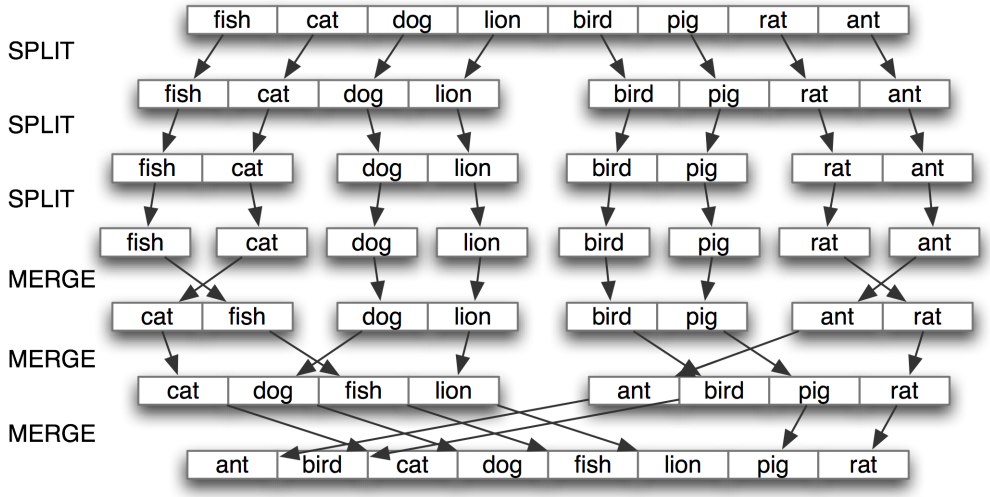


Figure 6.4: Sorting by Divide-and-Conquer (Mergesort)

Another strategy is **decrease-and-conquer**. In this approach, a small amount of work on a problem of size *n* permits us to reduce it to a problem of size *n/2*. Figure 6.5 illustrates this approach for the problem of finding the index of an item in a sorted list.

A third well-known strategy is **transform-and-conquer**. We attack a problem by transforming it into an instance of a problem we already know how to solve. For example, in order to detect duplicates entries in a list, we can **pre-sort** the list, then look for adjacent identical items, as shown in Listing 6.15. Our approach to n-gram chunking in Section 5.5 is another case of transform and conquer (why?).

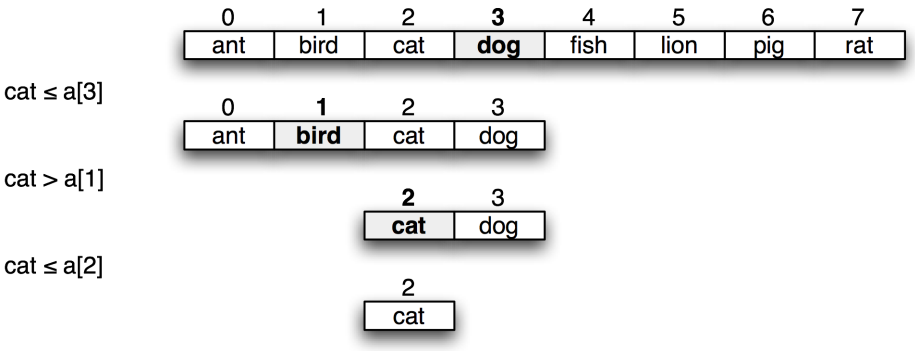


Figure 6.5: Searching by Decrease-and-Conquer (Binary Search)

Listing 24 Presorting a list for duplicate detection

```
def duplicates(words):
    prev = None
    dup = []
    for word in sorted(words):
        if word == prev:
            dup.append(word)
        else:
            prev = word
    return dup

>>> duplicates(['cat', 'dog', 'cat', 'pig', 'dog', 'cat', 'ant', 'cat'])
['cat', 'dog']
```

6.5.1 Recursion (notes)

We first saw recursion in Chapter 3, in a function that navigated the hypernym hierarchy of WordNet...

Iterative solution:

```
>>> def factorial(n):
...     result = 1
...     for i in range(n+1):
...         result *= i
...     return result
```

Recursive solution (base case, induction step)

```
>>> def factorial(n):
...     if n == 1:
...         return n
...     else:
...         return n * factorial(n-1)
```

[Simple example of recursion on strings.]

Generating all permutations of words, to check which ones are grammatical:

```
>>> def perms(seq):
...     if len(seq) <= 1:
...         yield seq
...     else:
...         for perm in perms(seq[1:]):
...             for i in range(len(perm)+1):
...                 yield perm[:i] + seq[0:1] + perm[i:]
>>> list(perms(['police', 'fish', 'cream']))
[['police', 'fish', 'cream'], ['fish', 'police', 'cream'],
 ['fish', 'cream', 'police'], ['police', 'cream', 'fish'],
 ['cream', 'police', 'fish'], ['cream', 'fish', 'police']]
```

6.5.2 Deeply Nested Objects (notes)

We can use recursive functions to build deeply-nested objects. Building a letter trie, Listing 6.16.

6.5.3 Dynamic Programming

Dynamic programming is a general technique for designing algorithms which is widely used in natural language processing. The term 'programming' is used in a different sense to what you might expect, to mean planning or scheduling. Dynamic programming is used when a problem contains overlapping sub-problems. Instead of computing solutions to these sub-problems repeatedly, we simply store them in a lookup table. In the remainder of this section we will introduce dynamic programming, but in a rather different context to syntactic parsing.

Pingala was an Indian author who lived around the 5th century B.C., and wrote a treatise on Sanscrit prosody called the *Chandas Shastra*. Virahanka extended this work around the 6th century A.D., studying the number of ways of combining short and long syllables to create a meter of length n. He found, for example, that there are five ways to construct a meter of length 4: V_4 = *{LL, SSL, SLS, LSS, SSSS}*. Observe that we can split V_4 into two subsets, those starting with L and those starting with S, as shown in (16).

Listing 25 Building a Letter Trie

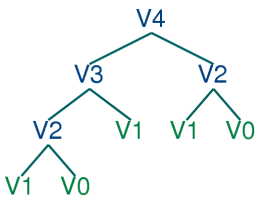
```
def insert(trie, key, value):
    if key:
        first, rest = key[0], key[1:]
        if first not in trie:
            trie[first] = {}
        insert(trie[first], rest, value)
    else:
        trie['value'] = value

>>> trie = {}
>>> insert(trie, 'chat', 'cat')
>>> insert(trie, 'chien', 'dog')
>>> trie['c']['h']
{'a': {'t': {'value': 'cat'}}, 'i': {'e': {'n': {'value': 'dog'}}}}
>>> trie['c']['h']['a']['t']['value']
'cat'
>>> pprint(trie)
{'c': {'h': {'a': {'t': {'value': 'cat'}},
             'i': {'e': {'n': {'value': 'dog'}}}}}}
```

(16)

V_4 =
LL, LSS
i.e. L prefixed to each item of V_2 = {L, SS}
SSL, SLS, SSSS
i.e. S prefixed to each item of V_3 = {SL, LS, SSS}

With this observation, we can write a little recursive function called `virahanka1()` to compute these meters, shown in Listing 6.17. Notice that, in order to compute V_4 we first compute V_3 and V_2. But to compute V_3, we need to first compute V_2 and V_1. This **call structure** is depicted in (17).

(17)



As you can see, V_2 is computed twice. This might not seem like a significant problem, but it turns out to be rather wasteful as n gets large: to compute V_{20} using this recursive technique, we would compute V_2 4,181 times; and for V_{40} we would compute V_2 63,245,986 times! A much better alternative is to store the value of V_2 in a table and look it up whenever we need it. The same goes for other values, such as V_3 and so on. Function `virahanka2()` implements a dynamic programming approach to the problem. It works by filling up a table (called `lookup`) with solutions to *all* smaller instances of the problem, stopping as soon as we reach the value we're interested in. At this point we read off the value and return it. Crucially, each sub-problem is only ever solved once.

Notice that the approach taken in `virahanka2()` is to solve smaller problems on the way to solving larger problems. Accordingly, this is known as the **bottom-up** approach to dynamic programming.

Listing 26 Three Ways to Compute Sansrit Meter

```
def virahanka1(n):
    if n == 0:
        return [""]
    elif n == 1:
        return ["S"]
    else:
        s = ["S" + prosody for prosody in virahanka1(n-1)]
        l = ["L" + prosody for prosody in virahanka1(n-2)]
        return s + l

def virahanka2(n):
    lookup = [[""], ["S"]]
    for i in range(n-1):
        s = ["S" + prosody for prosody in lookup[i+1]]
        l = ["L" + prosody for prosody in lookup[i]]
        lookup.append(s + l)
    return lookup[n]

def virahanka3(n, lookup={0:[""], 1:["S"]}):
    if n not in lookup:
        s = ["S" + prosody for prosody in virahanka3(n-1)]
        l = ["L" + prosody for prosody in virahanka3(n-2)]
        lookup[n] = s + l
    return lookup[n]

>>> virahanka1(4)
['SSSS', 'SSL', 'SLS', 'LSS', 'LL']
>>> virahanka2(4)
['SSSS', 'SSL', 'SLS', 'LSS', 'LL']
>>> virahanka3(4)
['SSSS', 'SSL', 'SLS', 'LSS', 'LL']
```

Unfortunately it turns out to be quite wasteful for some applications, since it may compute solutions to sub-problems that are never required for solving the main problem. This wasted computation can be avoided using the **top-down** approach to dynamic programming, which is illustrated in the function `virahanka3()` in Listing 6.17. Unlike the bottom-up approach, this approach is recursive. It avoids the huge wastage of `virahanka1()` by checking whether it has previously stored the result. If not, it computes the result recursively and stores it in the table. The last step is to return the stored result.

This concludes our brief introduction to dynamic programming. We will encounter it again in Chapter 8.

> **Note**
>
> Dynamic programming is a kind of **memoization**. A memoized function stores results of previous calls to the function along with the supplied parameters. If the function is subsequently called with those parameters, it returns the stored result instead of recalculating it.

6.5.4 Timing (notes)

We can easily test the efficiency gains made by the use of dynamic programming, or any other putative performance enhancement, using the `timeit` module:

```
>>> from timeit import Timer
>>> Timer("PYTHON CODE", "INITIALIZATION CODE").timeit()
```

[MORE]

6.5.5 Exercises

1. ◑ Write a recursive function `lookup(trie, key)` that looks up a key in a trie, and returns the value it finds. Extend the function to return a word when it is uniquely determined by its prefix (e.g. `vanguard` is the only word which starts with `vang-`, so `lookup(trie, 'vang')` should return the same thing as `lookup(trie, 'vanguard')`).

2. ◑ Read about string edit distance and the Levenshtein Algorithm. Try the implementation provided in `nltk_lite.utilities.edit_dist()`. How is this using dynamic programming? Does it use the bottom-up or top-down approach?

3. ◑ The Catalan numbers arise in many applications of combinatorial mathematics, including the counting of parse trees (Chapter 8). The series can be defined as follows: $C_0 = 1$, and $C_{n+1} = \Sigma_{0..n} (C_i C_{n-i})$.

   a) Write a recursive function to compute nth Catalan number C_n

   b) Now write another function that does this computation using dynamic programming

   c) Use the `timeit` module to compare the performance of these functions as n increases.

4. ★ Write a recursive function that pretty prints a trie in alphabetically sorted order, as follows

   chat: 'cat' --ien: 'dog' -???: ???

5. ★ Write a recursive function that processes text, locating the uniqueness point in each word, and discarding the remainder of each word. How much compression does this give? How readable is the resulting text?

6.6 Conclusion

[TO DO]

6.7 Further Reading

[Harel, 2004]
[Levitin, 2004]
http://docs.python.org/lib/typesseq-strings.html

Chapter 7

Grammars and Parsing

7.1 Introduction

Early experiences with the kind of grammar taught in school are sometimes perplexing. Your written work might have been graded by a teacher who red-lined all the grammar errors they wouldn't put up with. Like the plural pronoun or the dangling preposition in the last sentence, or sentences like this one which lack a main verb. If you learnt English as a second language, you might have found it difficult to discover which of these errors need to be fixed (or *needs* to be fixed?). Correct punctuation is an obsession for many writers and editors. It is easy to find cases where changing punctuation changes meaning. In the following example, the interpretation of a relative clause as restrictive or non-restrictive depends on the presence of commas alone:

(18a) The presidential candidate, who was extremely popular, smiled broadly.

(18b) The presidential candidate who was extremely popular smiled broadly.

In (18a), we assume there is just one presidential candidate, and say two things about her: that she was popular and that she smiled. In (18b), on the other hand, we use the description *who was extremely popular* as a means of identifying for the hearer which of several candidates we are referring to.

It is clear that some of these rules are important. However, others seem to be vestiges of antiquated style. Consider the injunction that *however* — when used to mean *nevertheless* — must not appear at the start of a sentence. Pullum argues that Strunk and White [Strunk and White, 1999] were merely insisting that English usage should conform to "an utterly unimportant minor statistical detail of style concerning adverb placement in the literature they knew" [Pullum, 2005]. This is a case where, a *descriptive* observation about language use became a *prescriptive* requirement. In NLP we usually discard such prescriptions, and use grammar to formalize observations about language as it is used, particularly as it is used in corpora.

In this chapter we present the fundamentals of syntax, focusing on constituency and tree representations, before describing the formal notation of context free grammar. Next we present parsers as an automatic way to associate syntactic structures with sentences. Finally, we give a detailed presentation of simple top-down and bottom-up parsing algorithms available in NLTK. Before launching into the theory we present some more naive observations about grammar, for the benefit of readers who do not have a background in linguistics.

7.2 More Observations about Grammar

Another function of a grammar is to explain our observations about ambiguous sentences. Even when the individual words are unambiguous, we can put them together to create ambiguous sentences, as in (19).

(19a) Fighting animals could be dangerous.

(19b) Visiting relatives can be tiresome.

A grammar will be able to assign two structures to each sentence, accounting for the two possible interpretations.

Perhaps another kind of syntactic variation, word order, is easier to understand. We know that the two sentences *Kim likes Sandy* and *Sandy likes Kim* have different meanings, and that *likes Sandy Kim* is simply ungrammatical. Similarly, we know that the following two sentences are equivalent:

(20a) The farmer *loaded* the cart with sand

(20b) The farmer *loaded* sand into the cart

However, consider the semantically similar verbs *filled* and *dumped*. Now the word order cannot be altered (ungrammatical sentences are prefixed with an asterisk.)

(21a) The farmer *filled* the cart with sand

(21b) *The farmer *filled* sand into the cart

(21c) *The farmer *dumped* the cart with sand

(21d) The farmer *dumped* sand into the cart

A further notable fact is that we have no difficulty accessing the meaning of sentences we have never encountered before. It is not difficult to concoct an entirely novel sentence, one that has probably never been used before in the history of the language, and yet all speakers of the language will agree about its meaning. In fact, the set of possible sentences is infinite, given that there is no upper bound on length. Consider the following passage from a children's story, containing a rather impressive sentence:

> You can imagine Piglet's joy when at last the ship came in sight of him. In after-years he liked to think that he had been in Very Great Danger during the Terrible Flood, but the only danger he had really been in was the last half-hour of his imprisonment, when Owl, who had just flown up, sat on a branch of his tree to comfort him, and told him a very long story about an aunt who had once laid a seagull's egg by mistake, and the story went on and on, rather like this sentence, until Piglet who was listening out of his window without much hope, went to sleep quietly and naturally, slipping slowly out of the window towards the water until he was only hanging on by his toes, at which moment, luckily, a sudden loud squawk from Owl, which was really part of the story, being what his aunt said, woke the Piglet up and just gave him time to jerk himself back into safety and say, "How interesting, and did she?" when -- well, you can imagine his joy when at last he saw the good ship, Brain of Pooh (Captain, C. Robin; 1st Mate, P. Bear) coming over the sea to rescue him... (from A.A. Milne *In which Piglet is Entirely Surrounded by Water*)

Our ability to produce and understand entirely new sentences, of arbitrary length, demonstrates that the set of well-formed sentences in English is infinite. The same case can be made for any human language.

This chapter presents grammars and parsing, as the formal and computational methods for investigating and modelling the linguistic phenomena we have been touching on (or tripping over). As we shall see, patterns of well-formedness and ill-formedness in a sequence of words can be understood with respect to the underlying **phrase structure** of the sentences. We can develop formal models of these structures using grammars and parsers. As before, the motivation is natural language *understanding*. How much more of the meaning of a text can we access when we can reliably recognize the linguistic structures it contains? Having read in a text, can a program 'understand' it enough to be able to answer simple questions about "what happened" or "who did what to whom" Also as before, we will develop simple programs to process annotated corpora and perform useful tasks.

7.3 What's the Use of Syntax?

Earlier chapters focussed on words: how to identify them, how to analyse their morphology, and how to assign them to classes via part-of-speech tags. We have also seen how to identify recurring sequences of words (i.e. n-grams). Nevertheless, there seem to be linguistic regularities which cannot be described simply in terms of n-grams.

In this section we will see why it is useful to have some kind of syntactic representation of sentences. In particular, we will see that there are systematic aspects of meaning which are much easier to capture once we have established a level of syntactic structure.

7.3.1 Syntactic Ambiguity

We have seen that sentences can be ambiguous. If we overheard someone say *I went to the bank*, we wouldn't know whether it was a river bank or a financial institution. This ambiguity concerns the meaning of the word *bank*, and is a kind of **lexical ambiguity**.

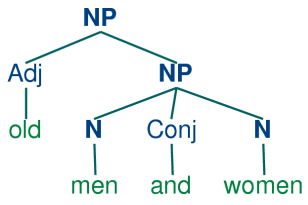
However, other kinds of ambiguity cannot be explained in terms of ambiguity of specific words. Consider a phrase involving an adjective with a conjunction: *old men and women*. Does *old* have wider scope than *and*, or is it the other way round? In fact, both interpretations are possible, and we can represent the different scopes using parentheses:

(22a) old (men and women)

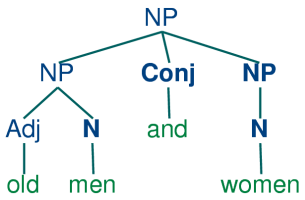
(22b) (old men) and women

One convenient way of representing this scope difference at a structural level is by means of a **tree diagram**, as shown in (23).

(23a)



(23b)



Note that linguistic trees grow upside down: the node labeled S is the **root** of the tree, while the **leaves** of the tree are labeled with the words.

In NLTK, you can easily produce trees like this yourself with the following commands:

```
>>> from nltk_lite.parse import bracket_parse
>>> tree = bracket_parse('(NP (Adj old) (NP (N men) (Conj and) (N women)))')
>>> tree.draw()                    # doctest: +SKIP
```

We can construct other examples of syntactic ambiguity involving the coordinating conjunctions *and* and *or*, e.g. *Kim left or Dana arrived and everyone cheered*. We can describe this ambiguity in terms of the relative semantic **scope** of *or* and *and*.

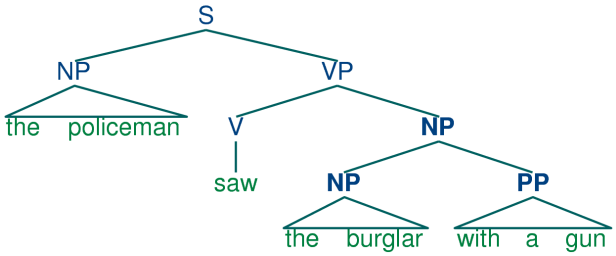
For our third illustration of ambiguity, we look at prepositional phrases. Consider a sentence like: *I saw the man with a telescope*. Who has the telescope? To clarify what is going on here, consider the following pair of sentences:

(24a) The policeman saw a burglar *with a gun*. (not some other burglar)

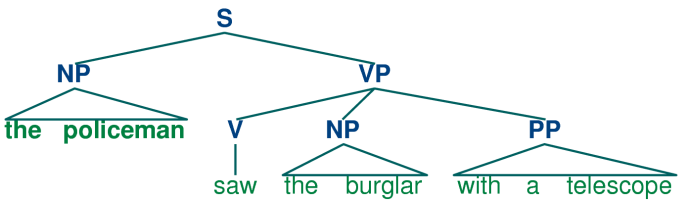
(24b) The policeman saw a burglar *with a telescope*. (not with his naked eye)

In both cases, there is a prepositional phrase introduced by *with*. In the first case this phrase modifies the noun *burglar*, and in the second case it modifies the verb *saw*. We could again think of this in terms of scope: does the prepositional phrase (PP) just have scope over the NP *a burglar*, or does it have scope over the whole verb phrase? As before, we can represent the difference in terms of tree structure:

(25a)



(25b)



In (25)a, the PP attaches to the NP, while in (25)b, the PP attaches to the VP.

We can generate these trees in Python as follows:

```
>>> s1 = '(S (NP the policeman) (VP (V saw) (NP (NP the burglar) (PP with a gun))))'
>>> s2 = '(S (NP the policeman) (VP (V saw) (NP the burglar) (PP with a telescope)))'
>>> tree1 = bracket_parse(s1)
>>> tree2 = bracket_parse(s2)
```

We can discard the structure to get the list of **leaves**, and we can confirm that both trees have the same leaves. We can also see that the trees have different **heights** (given by the number of nodes in the longest branch of the tree, starting at S and descending to the words):

```
>>> tree1.leaves()
['the', 'policeman', 'saw', 'the', 'burglar', 'with', 'a', 'gun']
>>> tree1.leaves() == tree2.leaves()
True
>>> tree1.height() == tree2.height()
False
```

In general, how can we determine whether a prepositional phrase modifies the preceding noun or verb? This problem is known as **prepositional phrase attachment ambiguity**. The **Prepositional Phrase Attachment Corpus** makes it possible for us to study this question systematically. The corpus is derived from the IBM-Lancaster Treebank of Computer Manuals and from the Penn Treebank, and distills out only the essential information about PP attachment. Consider the sentence from the WSJ in (26a). The corresponding line in the Prepositional Phrase Attachment Corpus is shown in (26b).

(26a) Four of the five surviving workers have asbestos-related diseases, including three with recently diagnosed cancer.

(26b)
```
16 including three with cancer N
```

That is, it includes an identifier for the original sentence, the head of the relevant verb phrase (i.e., *including*), the head of the verb's NP object (*three*), the preposition (*with*), and the head noun within the prepositional phrase (*cancer*). Finally, it contains an "attachment" feature (N or V) to indicate whether the prepositional phrase attaches to (modifies) the noun phrase or the verb phrase. Here are some further examples:

(27)
```
47830 allow visits between families N
47830 allow visits on peninsula V
42457 acquired interest in firm N
42457 acquired interest in 1986 V
```

The PP attachments in (27) can also be made explicit by using phrase groupings as in (28).

(28)
```
allow (NP visits (PP between families))
allow (NP visits) (PP on peninsula)
acquired (NP interest (PP in firm))
acquired (NP interest) (PP in 1986)
```

Observe in each case that the argument of the verb is either a single complex expression `(visits (between families))` or a pair of simpler expressions `(visits) (on peninsula)`.

We can access the Prepositional Phrase Attachment Corpus from NLTK as follows:

```
>>> from nltk_lite.corpora import ppattach, extract
>>> from pprint import pprint
>>> item = extract(9, ppattach.dictionary('training'))
```

```
>>> pprint(item)
{'attachment': 'N',
 'noun1': 'three',
 'noun2': 'cancer',
 'prep': 'with',
 'sent': '16',
 'verb': 'including'}
```

If we go back to our first examples of PP attachment ambiguity, it appears as though it is the PP itself (e.g., *with a gun* versus *with a telescope*) that determines the attachment. However, we can use this corpus to find examples where other factors come into play. For example, it appears that the verb is the key factor in (29).

(29)
```
8582 received offer from group V
19131 rejected offer from group N
```

7.3.2 Constituency

We claimed earlier that one of the motivations for building syntactic structure was to help make explicit how a sentence says "who did what to whom". Let's just focus for a while on the "who" part of this story: in other words, how can syntax tell us what the subject of a sentence is? At first, you might think this task is rather simple — so simple indeed that we don't need to bother with syntax. In a sentence such as *The fierce dog bit the man* we know that it is the dog that is doing the biting. So we could say that the noun phrase immediately preceding the verb is the subject of the sentence. And we might try to make this more explicit in terms of sequences part-of-speech tags. Let's try to come up with a simple definition of *noun phrase*; we might start off with something like this, based on our knowledge of noun phrase chunking (Chapter 5):

(30) DT JJ* NN

We're using regular expression notation here in the form of JJ* to indicate a sequence of zero or more JJs. So this is intended to say that a noun phrase can consist of a determiner, possibly followed by some adjectives, followed by a noun. Then we can go on to say that if we can find a sequence of tagged words like this that precedes a word tagged as a verb, then we've identified the subject. But now think about this sentence:

(31) The child with a fierce dog bit the man.

This time, it's the child that is doing the biting. But the tag sequence preceding the verb is:

(32) DT NN IN DT JJ NN

Our previous attempt at identifying the subject would have incorrectly come up with *the fierce dog* as the subject. So our next hypothesis would have to be a bit more complex. For example, we might say that the subject can be identified as any string matching the following pattern before the verb:

(33) DT JJ* NN (IN DT JJ* NN)*

In other words, we need to find a noun phrase followed by zero or more sequences consisting of a preposition followed by a noun phrase. Now there are two unpleasant aspects to this proposed solution. The first is aesthetic: we are forced into repeating the sequence of tags (DT JJ* NN) that constituted our initial notion of noun phrase, and our initial notion was in any case a drastic simplification. More worrying, this approach still doesn't work! For consider the following example:

(34) The seagull that attacked the child with the fierce dog bit the man.

This time the seagull is the culprit, but it won't be detected as subject by our attempt to match sequences of tags. So it seems that we need a richer account of how words are *grouped* together into patterns, and a way of referring to these groupings at different points in the sentence structure. This idea of grouping is often called syntactic **constituency**.

As we have just seen, a well-formed sentence of a language is more than an arbitrary sequence of words from the language. Certain kinds of words usually go together. For instance, determiners like *the* are typically followed by adjectives or nouns, but not by verbs. Groups of words form intermediate structures called phrases or **constituents**. These constituents can be identified using standard syntactic tests, such as substitution, movement and coordination. For example, if a sequence of words can be replaced with a pronoun, then that sequence is likely to be a constituent. According to this test, we can infer that the italicised string in the following example is a constituent, since it can be replaced by *they*:

(35a) *Ordinary daily multivitamin and mineral supplements* could help adults with diabetes fight off some minor infections.

(35b) *They* could help adults with diabetes fight off some minor infections.

In order to identify whether a phrase is the subject of a sentence, we can use the construction called **Subject-Auxiliary Inversion** in English. This construction allows us to form so-called Yes-No Questions. That is, corresponding to the statement in (36a), we have the question in (36b):

(36a) All the cakes have been eaten.

(36b) Have *all the cakes* been eaten?

Roughly speaking, if a sentence already contains an auxiliary verb, such as *has* in (36a), then we can turn it into a Yes-No Question by moving the auxiliary verb 'over' the subject noun phrase to the front of the sentence. If there is no auxiliary in the statement, then we insert the appropriate form of *do* as the fronted auxiliary and replace the tensed main verb by its base form:

(37a) The fierce dog bit the man.

(37b) Did *the fierce dog* bite the man?

As we would hope, this test also confirms our earlier claim about the subject constituent of (34):

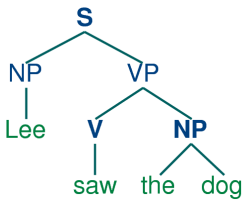
(38) Did *the seagull that attacked the child with the fierce dog* bite the man?

To sum up then, we have seen that the notion of constituent brings a number of benefits. By having a constituent labeled NOUN PHRASE, we can provide a unified statement of the classes of word that constitute that phrase, and reuse this statement in describing noun phrases wherever they occur in the sentence. Second, we can use the notion of a noun phrase in defining the subject of sentence, which in turn is a crucial ingredient in determining the "who does what to whom" aspect of meaning.

7.3.3 More on Trees

A tree is a set of connected nodes, each of which is labeled with a category. It common to use a 'family' metaphor to talk about the relationships of nodes in a tree: for example, S is the **parent** of VP; conversely VP is a **daughter** (or **child**) of S. Also, since NP and VP are both daughters of S, they are also **sisters**. Here is an example of a tree:

(39)



Although it is helpful to represent trees in a graphical format, for computational purposes we usually need a more text-oriented representation. One standard method (used in the Penn Treebank) is to use a combination of bracket and labels to indicate the structure, as shown here:

```
(S
   (NP  'Lee')
   (VP
      (V 'saw')
      (NP
         (Det 'the')
         (N  'dog'))))
```

The conventions for displaying trees in NLTK are similar:

```
(S: (NP: 'Lee') (VP: (V: 'saw') (NP: 'the' 'dog')))
```

In such trees, the node value is a string containing the tree's constituent type (e.g., NP or VP), while the children encode the hierarchical contents of the tree.

Although we will focus on syntactic trees, trees can be used to encode *any* homogeneous hierarchical structure that spans a sequence of linguistic forms (e.g. morphological structure, discourse structure). In the general case, leaves and node values do not have to be strings.

In NLTK, trees are created with the `Tree` constructor, which takes a node value and a list of zero or more children. Here's a couple of simple trees:

```
>>> from nltk_lite.parse import Tree
>>> tree1 = Tree('NP', ['John'])
>>> print tree1
(NP: 'John')
>>> tree2 = Tree('NP', ['the', 'man'])
>>> print tree2
(NP: 'the' 'man')
```

We can incorporate these into successively larger trees as follows:

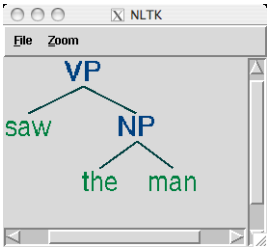
```
>>> tree3 = Tree('VP', ['saw', tree2])
>>> tree4 = Tree('S', [tree1, tree3])
>>> print tree4
(S: (NP: 'John') (VP: 'saw' (NP: 'the' 'man')))
```

Here are some of the methods available for tree objects:

```
>>> tree4[1]
(VP: 'saw' (NP: 'the' 'man'))
>>> tree4[1].node
'VP'
>>> tree4.leaves()
['John', 'saw', 'the', 'man']
>>> tree4[1,1,0]
'saw'
```

The printed representation for complex trees can be difficult to read. In these cases, the `draw` method can be very useful. It opens a new window, containing a graphical representation of the tree. The tree display window allows you to zoom in and out; to collapse and expand subtrees; and to print the graphical representation to a postscript file (for inclusion in a document).

```
>>> tree3.draw()
```



7.3.4 Treebanks (notes)

The `nltk_lite.corpora` module defines the `treebank` corpus reader, which contains a 10% sample of the Penn Treebank corpus.

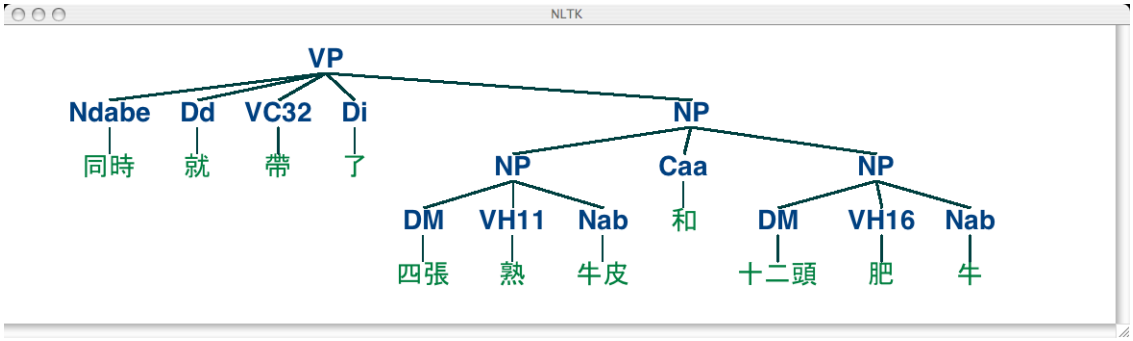
```
>>> from nltk_lite.corpora import treebank, extract
>>> print extract(0, treebank.parsed())
(S:
  (NP-SBJ:
    (NP: (NNP: 'Pierre') (NNP: 'Vinken'))
    (,: ',')
    (ADJP: (NP: (CD: '61') (NNS: 'years')) (JJ: 'old'))
    (,: ','))
  (VP:
    (MD: 'will')
    (VP:
      (VB: 'join')
      (NP: (DT: 'the') (NN: 'board'))
      (PP-CLR:
        (IN: 'as')
        (NP: (DT: 'a') (JJ: 'nonexecutive') (NN: 'director')))
      (NP-TMP: (NNP: 'Nov.') (CD: '29'))))
  (.: '.'))
```

NLTK also includes a sample from the *Sinica Treebank Corpus*, consisting of 10,000 parsed sentences drawn from the *Academia Sinica Balanced Corpus of Modern Chinese*. Here is a code fragment to read and display one of the trees in this corpus.

Listing 27

```
def indent_tree(t, level=0, first=False, width=8):
    if not first:
        print ' '*(width+1)*level,
    try:
        print "%-*s" % (width, t.node),
        indent_tree(t[0], level+1, first=True)
        for child in t[1:]:
            indent_tree(child, level+1, first=False)
    except AttributeError:
        print t

>>> t = extract(0, treebank.parsed())
>>> indent_tree(t)
 S        NP-SBJ   NP       NNP      Pierre
                            NNP      Vinken
                   ,        ,
                   ADJP     NP       CD       61
                                     NNS      years
                            JJ       old
                   ,        ,
          VP       MD       will
                   VP       VB       join
                            NP       DT       the
                                     NN       board
                            PP-CLR   IN       as
                                     NP       DT       a
                                              JJ       nonexecutive
                                              NN       director
                            NP-TMP   NNP      Nov.
                                     CD       29
          .        .
```

```
>>> from nltk_lite.corpora import sinica_treebank, extract
>>> extract(3450, sinica_treebank.parsed()).draw()
```



(40)

Note that we can read tagged text from a Treebank corpus, using the `tagged()` method:

```
>>> print extract(0, treebank.tagged())
[('Pierre', 'NNP'), ('Vinken', 'NNP'), (',', ','), ('61', 'CD'), ('years', 'NNS'),
('old', 'JJ'), (',', ','), ('will', 'MD'), ('join', 'VB'), ('the', 'DT'),
('board', 'NN'), ('as', 'IN'), ('a', 'DT'), ('nonexecutive', 'JJ'),
('director', 'NN'), ('Nov.', 'NNP'), ('29', 'CD'), ('.', '.')]
```

7.3.5 Exercises

1. ☼ Can you come up with grammatical sentences which have probably never been uttered before? (Take turns with a partner.) What does this tell you about human language?

2. ☼ Recall Strunk and White's prohibition against sentence-initial *however* used to mean "although". Do a web search for *however* used at the start of the sentence. How widely used is this construction?

3. ☼ Consider the sentence *Kim arrived or Dana left and everyone cheered.* Write down the parenthesized forms to show the relative scope of *and* and *or*. Generate tree structures corresponding to both of these interpretations.

4. ☼ The `Tree` class implements a variety of other useful methods. See the `Tree` help documentation for more details, i.e. import the Tree class and then type `help(Tree)`.

5. ☼ **Building trees:**

   a) Write code to produce two trees, one for each reading of the phrase *old men and women*

   b) Encode any of the trees presented in this chapter as a labeled bracketing and use the `nltk_lite.parse` module's `bracket_parse()` method to check that it is well-formed. Now use the `draw()` to display the tree.

   c) As in (a) above, draw a tree for *The woman saw a man last Thursday*.

6. ☼ Write a recursive function to traverse a tree and return the depth of the tree, such that a tree with a single node would have depth zero. (Hint: the depth of a subtree is the maximum depth of its children, plus one.)

7. ☼ Analyze the A.A. Milne sentence about Piglet, by underlining all of the sentences it contains then replacing these with S (e.g. the first sentence becomes S *when*:lx‘ S). Draw a tree structure for this “compressed” sentence. What are the main syntactic constructions used for building such a long sentence?

8. ◑ To compare multiple trees in a single window, we can use the `draw_trees()` method. Define some trees and try it out:

```
>>> from nltk_lite.draw.tree import draw_trees
>>> draw_trees(tree1, tree2, tree3)
```

9. ◑ Using tree positions, list the subjects of the first 100 sentences in the Penn treebank; to make the results easier to view, limit the extracted subjects to subtrees whose height is 2.

10. ◑ Inspect the Prepositional Phrase Attachment Corpus and try to suggest some factors that influence PP attachment.

11. ◑ In this section we claimed that there are linguistic regularities which cannot be described simply in terms of n-grams. Consider the following sentence, particularly the position of the phrase *in his turn*. Does this illustrate a problem for an approach based on n-grams?

> *What was more, the in his turn somewhat youngish Nikolay Parfenovich also turned out to be the only person in the entire world to acquire a sincere liking to our “discriminated-against” public procurator.* (Dostoevsky: The Brothers Karamazov)

12. ◑ Write a recursive function that produces a nested bracketing for a tree, leaving out the leaf nodes, and displaying the non-terminal labels after their subtrees. So the above example about Pierre Vinken would produce: `[[[NNP NNP]NP , [ADJP [CD NNS]NP JJ]ADJP ,]NP-SBJ MD [VB [DT NN]NP [IN [DT JJ NN]NP]PP-CLR [NNP CD]NP-TMP]VP .]S` Consecutive categories should be separated by space.

1. ◑ Download several electronic books from Project Gutenberg. Write a program to scan these texts for any extremely long sentences. What is the longest sentence you can find? What syntactic construction(s) are responsible for such long sentences?

2. ★ One common way of defining the subject of a sentence S in English is as *the noun phrase that is the daughter of* S *and the sister of* VP. Write a function that takes the tree for a sentence and returns the subtree corresponding to the subject of the sentence. What should it do if the root node of the tree passed to this function is not S, or it lacks a subject?

7.4 Context Free Grammar

As we have seen, languages are infinite — there is no principled upper-bound on the length of a sentence. Nevertheless, we would like to write (finite) programs that can process well-formed sentences. It turns out that we can characterize what we mean by well-formedness using a grammar. The way that finite grammars are able to describe an infinite set uses **recursion**. (We already came across this idea when we looked at regular expressions: the finite expression `a+` is able to describe the infinite set `{a , aa, aaa, aaaa, ...}`). Apart from their compactness, grammars usually capture important

structural and distributional properties of the language, and can be used to map between sequences of words and abstract representations of meaning. Even if we were to impose an upper bound on sentence length to ensure the language was finite, we would probably still want to come up with a compact representation in the form of a grammar.

A **grammar** is a formal system which specifies which sequences of words are well-formed in the language, and which provides one or more phrase structures for well-formed sequences. We will be looking at **context-free grammar** (CFG), which is a collection of **productions** of the form S → NP VP. This says that a constituent S can consist of sub-constituents NP and VP. Similarly, the production V → `'saw' | ''walked'` means that the constituent V can consist of the string *saw* or *walked*. For a phrase structure tree to be well-formed relative to a grammar, each non-terminal node and its children must correspond to a production in the grammar.

7.4.1 A Simple Grammar

Let's start off by looking at a simple context-free grammar. By convention, the left-hand-side of the first production is the **start-symbol** of the grammar, and all well-formed trees must have this symbol as their root label.

(41) S → NP VP
NP → Det N | Det N PP
VP → V | V NP | V NP PP
PP → P NP

Det → 'the' | 'a'
N → 'man' | 'park' | 'dog' | 'telescope'
V → 'saw' | 'walked'
P → 'in' | 'with'

This grammar contains productions involving various syntactic categories, as laid out in Table 7.1.

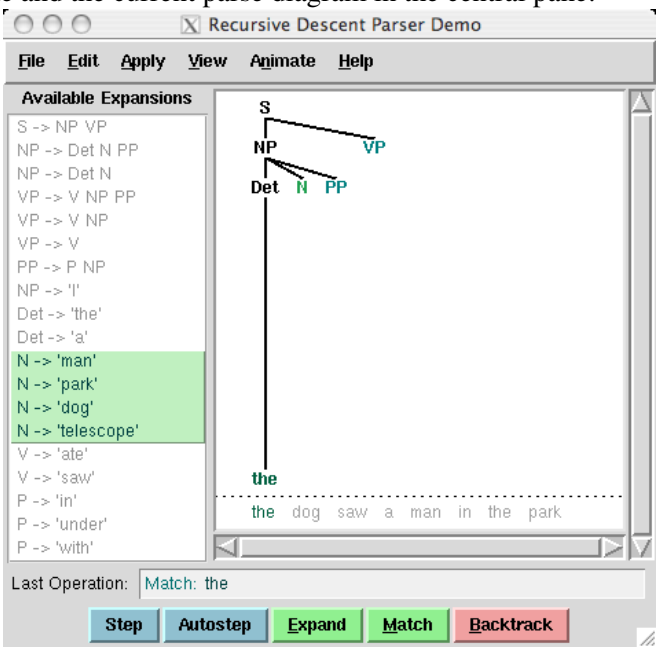
Symbol	Meaning	Example
S	sentence	*the man walked*
NP	noun phrase	*a dog*
VP	verb phrase	*saw a park*
PP	prepositional phrase	*with a telescope*
...	...	...
Det	determiner	*the*
N	noun	*dog*
V	verb	*walked*
P	preposition	*in*

Table 7.1: Syntactic Categories

In our following discussion of grammar, we will use the following terminology. The grammar consists of productions, where each production involves a single **non-terminal** (e.g. S, NP), an arrow, and one or more non-terminals and **terminals** (e.g. *walked*). The productions are often divided into two

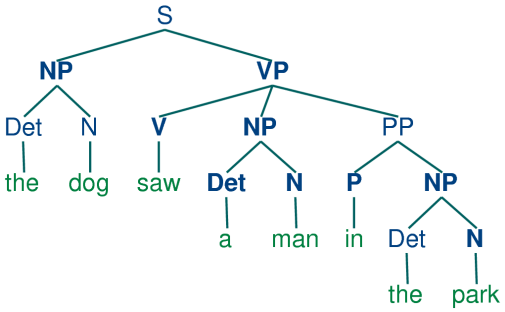
main groups. The **grammatical productions** are those without a terminal on the right-hand side. The **lexical productions** are those having a terminal on the right-hand side. A special case of non-terminals are the **pre-terminals**, which appear on the left-hand side of lexical productions. We will say that a grammar **licenses** a tree if each non-terminal X with children Y_1 ... Y_n corresponds to a production in the grammar of the form: $X \rightarrow Y_1 \ldots Y_n$.

In order to get started with developing simple grammars of your own, you will probably find it convenient to play with the recursive descent parser demo, `from nltk_lite.draw.rdparser .demo()`. The demo opens a window which displays a list of grammar productions in the lefthand pane and the current parse diagram in the central pane:

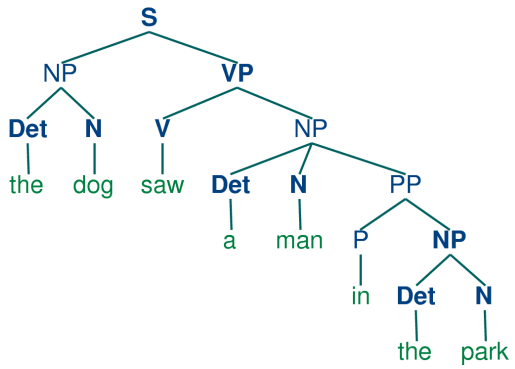


The demo comes with the grammar in (41) already loaded. We will discuss the parsing algorithm in greater detail below, but for the time being you can get an idea of how it works by using the *autostep* button. If we parse the string *The dog saw a man in the park* using the grammar in (41), we end up with two trees:

(42a)



(42b)



Since our grammar licenses two trees for this sentence, the sentence is said to be **structurally ambiguous**. The ambiguity in question is called a *prepositional phrase attachment ambiguity*, as we saw earlier in this chapter. As you may recall, it is an ambiguity about attachment since the PP *in the park* needs to be attached to one of two places in the tree: either as a daughter of VP or else as a daughter of NP. When the PP is attached to VP, the seeing event happened in the park. However, if the PP is attached to NP, then the man was in the park, and the agent of the seeing (the dog) might have been sitting on the balcony of an apartment overlooking the park. As we will see, dealing with ambiguity is a key challenge in parsing.

7.4.2 Recursion in syntactic structure

Observe that sentences can be nested within sentences, with no limit to the depth:

(43a) Jodie won the 100m freestyle

(43b) "The Age" reported that Jodie won the 100m freestyle

(43c) Sandy said "The Age" reported that Jodie won the 100m freestyle

(43d) I think Sandy said "The Age" reported that Jodie won the 100m freestyle

This nesting is explained in terms of **recursion**. A grammar is said to be **recursive** if a category occurring on the lefthand side of a production (such as S in this case) also appears on the righthand side of a production. If this dual occurrence takes place in *one and the same production*, then we have **direct recursion**; otherwise we have **indirect recursion**. There is no recursion in (41). However, the grammar in (44) illustrates both kinds of recursive production:

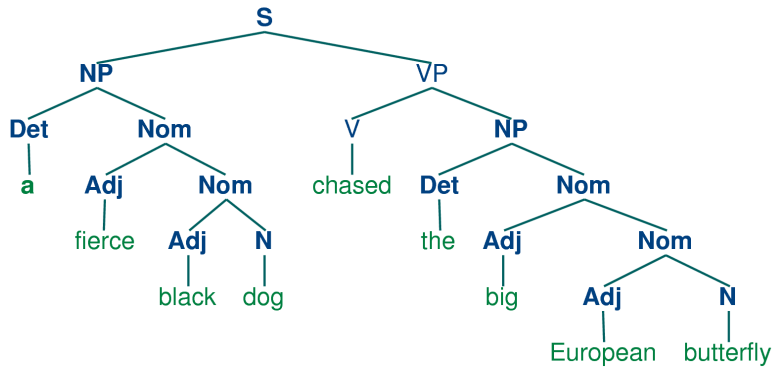
(44)

```
S  → NP VP
NP → Det Nom | Det Nom PP | PropN
Nom → Adj Nom | N
VP → V | V NP | V NP PP | V S
PP → P NP

PropN → 'John' | 'Mary'
Det → 'the' | 'a'
N → 'man' | 'woman' | 'park' | 'dog' | 'lead' | 'telescope' | 'butterfly'
Adj  → 'fierce' | 'black' |  'big' | 'European'
V → 'saw' | 'chased' | 'barked'  | 'disappeared' | 'said' | 'reported'
P → 'in' | 'with'
```

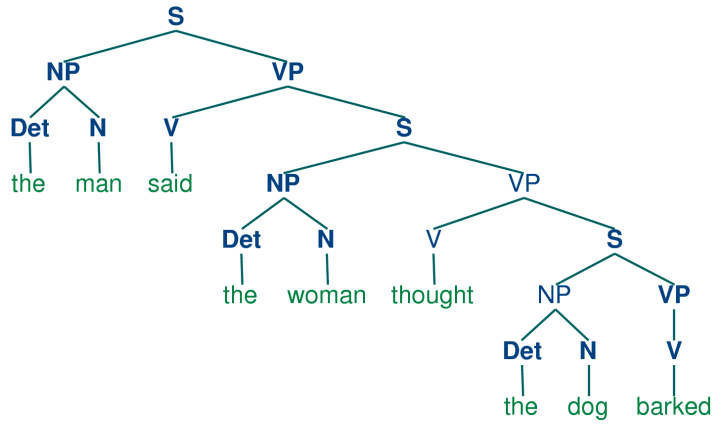
Notice that the production NOM → ADJ NOM (where NOM is the category of nominals) involves direct recursion on the category NOM, whereas indirect recursion on S arises from the combination of two productions, namely S → NP VP and VP → V S.

To see how recursion is handled in this grammar, consider the following trees. Example nested-nominals involves nested nominal phrases, while nested-sentences contains nested sentences.

(45a)

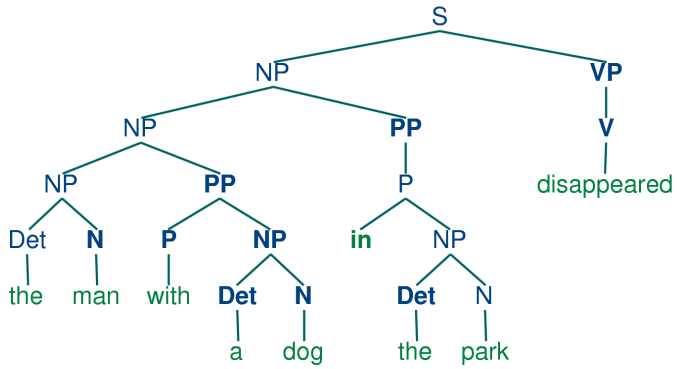


(45b)



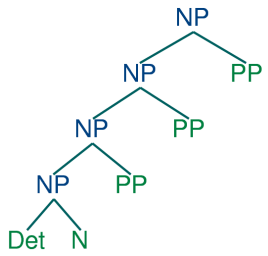
If you did the exercises for the last section, you will have noticed that the recursive descent parser fails to deal properly with the following production: NP → NP PP. From a linguistic point of view, this production is perfectly respectable, and will allow us to derive trees like this:

(46)



More schematically, the trees for these compound noun phrases will be of the following shape:

(47)



The structure in (47) is called a **left recursive** structure. These occur frequently in analyses of English, and the failure of recursive descent parsers to deal adequately with left recursion means that we will need to find alternative approaches.

7.4.3 Heads, Complements and Modifiers

Let us take a closer look at verbs. The grammar (44) correctly generates examples like (48), corresponding to the four productions with VP on the lefthand side:

(48a) The woman gave the telescope to the dog

(48b) The woman saw a man

(48c) A man said that the woman disappeared

(48d) The dog barked

That is, *gave* can occur with a following NP and PP; *saw* can occur with a following NP; *said* can occur with a following S; and *barked* can occur with no following phrase. In these cases, NP, PP and S are called **complements** of the respective verbs, and the verbs themselves are called **heads** of the verb phrase.

However, there are fairly strong constraints on what verbs can occur with what complements. Thus, we would like our grammars to mark the following examples as ungrammatical[1]:

(49a) *The woman disappeared the telescope to the dog

(49b) *The dog barked a man

(49c) *A man gave that the woman disappeared

(49d) *A man said

How can we ensure that our grammar correctly excludes the ungrammatical examples in (49)? We need some way of constraining grammar productions which expand VP so that verbs *only* cooccur with their correct complements. We do this by dividing the class of verbs into **subcategories**, each of which is associated with a different set of complements. For example, **transitive verbs** such as *saw*, *kissed* and *hit* require a following NP object complement. Borrowing from the terminology of chemistry, we sometimes refer to the **valency** of a verb, that is, its capacity to combine with a sequence of arguments and thereby compose a verb phrase.

Let's introduce a new category label for such verbs, namely TV (for Transitive Verb), and use it in the following productions:

[1] It should be borne in mind that it is possible to create examples which involve 'non-standard' but interpretable combinations of verbs and complements. Thus, we can, at a stretch, interpret *the man disappeared the dog* as meaning that the man made the dog disappear. We will ignore such examples here.

(50)
```
VP → TV NP
TV → 'saw' | 'kissed' | 'hit'
```

Now **the dog barked the man* is excluded since we haven't listed *barked* as a V_TR, but *the woman saw a man* is still allowed. Table 7.2 provides more examples of labels for verb subcategories.

Symbol	Meaning	Example
IV	intransitive verb	*barked*
TV	transitive verb	*saw a man*
DatV	dative verb	*gave a dog to a man*
SV	sentential verb	*said that a dog barked*

Table 7.2: Verb Subcategories

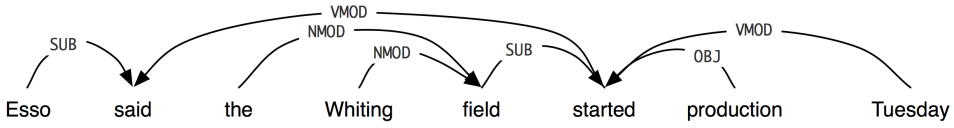
The revised grammar for VP will now look like this:

(51)
```
VP → DATV NP PP
VP → TV NP
VP → SV S
VP → IV

DATV → 'gave' | 'donated' | 'presented'
TV → 'saw' | 'kissed' | 'hit' | 'sang'
SV → 'said' | 'knew' | 'alleged'
IV → 'barked' | 'disappeared' | 'elapsed' | 'sang'
```

Notice that according to (51), a given lexical item can belong to more than one subcategory. For example, *sang* can occur both with and without a following NP complement.

7.4.4 Dependency Grammar

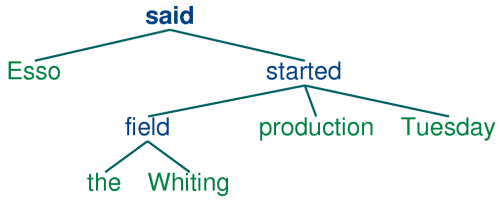
Although we concentrate on phrase structure grammars in this chapter, we should mention an alternative approach, namely **dependency grammar**. Rather than taking starting from the grouping of words into constituents, dependency grammar takes as basic the notion that one word can be dependent on another (namely, its head). The head of a sentence is usually taken to be the main verb, and every other word is either dependent on this head, or connects to it through a path of dependencies. Figure (52) illustrates a dependency graph, where the head of the arrow points to the head of a dependency.

(52)


As you will see, the arcs in Figure (52) are labeled with the particular dependency relation that holds between a dependent and its head. For example, *Esso* bears the subject relation to *said* (which is the head of the whole sentence), and *Tuesday* bears a verbal modifier (VMOD) relation to *started*.

An alternative way of representing the dependency relationships is illustrated in the tree (53), where dependents are shown as daughters of their heads.

(53)



One format for encoding dependency information places each word on a line, followed by its part-of-speech tag, the index of its head, and the label of the dependency relation (cf. [Nivre et al., 2006]). The index of a word is implicitly given by the ordering of the lines (with 1 as the first index). This is illustrated in the following code snippet:

```
>>> from nltk_lite.contrib.dependency import DepGraph
>>> dg = DepGraph().read("""Esso     NNP     2       SUB
... said     VBD     0       ROOT
... the      DT      5       NMOD
... Whiting  NNP     5       NMOD
... field    NN      6       SUB
... started  VBD     2       VMOD
... production       NN      6       OBJ
... Tuesday  NNP     6       VMOD""")
```

As you will see, this format also adopts the convention that the head of the sentence is dependent on an empty node, indexed as 0. We can use the `deptree()` method of a `DepGraph()` object to build an NLTK tree like that illustrated earlier in (53).

```
>>> tree = dg.deptree()
>>> tree.draw()
```

7.4.5 Formalizing Context Free Grammars

We have seen that a CFG contains terminal and nonterminal symbols, and productions which dictate how constituents are expanded into other constituents and words. In this section, we provide some formal definitions.

A CFG is a 4-tuple N, Σ, P, S, where:

- Σ is a set of *terminal* symbols (e.g., lexical items);
- N is a set of *non-terminal* symbols (the category labels);
- P is a set of *productions* of the form $A \rightarrow \alpha$, where
  - A is a non-terminal, and
  - α is a string of symbols from $(N \ \Sigma)^*$ (i.e., strings of either terminals or non-terminals);
- S is the *start symbol*.

A **derivation** of a string from a non-terminal A in grammar G is the result of successively applying productions from G to A. For example, (54) is a derivation of *the dog with a telescope* for the grammar in (41).

(54)
```
NP
Det N PP
the N PP
the dog PP
the dog P NP
the dog with NP
the dog with Det N
the dog with a N
the dog with a telescope
```

Although we have chosen here to expand the leftmost non-terminal symbol at each stage, this is not obligatory; productions can be applied in any order. Thus, derivation (54) could equally have started off in the following manner:

(55)
```
NP
Det N PP
Det N P NP
Det N with NP
...
```

We can also write derivation (54) as:

(56) NP ⇒ DET N PP ⇒ *the* N PP ⇒ *the dog* PP ⇒ *the dog* P NP ⇒ *the dog with* NP ⇒ *the dog with a* N ⇒ *the dog with a telescope*

where ⇒ means "derives in one step". We use ⇒* to mean "derives in zero or more steps":

- α ⇒* α for any string α, and
- if α ⇒* β and β ⇒ γ, then α ⇒* γ.

We write A ⇒* α to indicate that α can be derived from A.

In NLTK, context free grammars are defined in the `parse.cfg` module. The easiest way to construct a grammar object is from the standard string representation of grammars. In Listing 7.2 we define a grammar and use it to parse a simple sentence. You will learn more about parsing in the next section.

7.4.6 Exercises

1. ☼ In the recursive descent parser demo, experiment with changing the sentence to be parsed by selecting *Edit Text* in the *Edit* menu.
2. ☼ Can the grammar in (41) be used to describe sentences which are more than 20 words in length?
3. ◑ You can modify the grammar in the recursive descent parser demo by selecting *Edit Grammar* in the *Edit* menu. Change the first expansion production, namely `NP -> Det N PP`, to `NP -> NP PP`. Using the *Step* button, try to build a parse tree. What happens?
4. ◑ Extend the grammar in (44) with productions which expand prepositions as intransitive, transitive and requiring a PP complement. Based on these productions, use the method of the preceding exercise to draw a tree for the sentence *Lee ran away home*.

Listing 28 Context Free Grammars in NLTK

```
from nltk_lite import parse
grammar = parse.cfg.parse_cfg("""
  S -> NP VP
  VP -> V NP | V NP PP
  V -> "saw" | "ate"
  NP -> "John" | "Mary" | "Bob" | Det N | Det N PP
  Det -> "a" | "an" | "the" | "my"
  N -> "dog" | "cat" | "cookie" | "park"
  PP -> P NP
  P -> "in" | "on" | "by" | "with"
  """)

>>> from nltk_lite import tokenize
>>> sent = list(tokenize.whitespace("Mary saw Bob"))
>>> rd_parser = parse.RecursiveDescent(grammar)
>>> for p in rd_parser.get_parse_list(sent):
...     print p
(S: (NP: 'Mary') (VP: (V: 'saw') (NP: 'Bob')))
```

5. ◑ Pick some common verbs and complete the following tasks:

   a) Write a program to find those verbs in the PP Attachment Corpus included with NLTK. Find any cases where the same verb exhibits two different attachments, but where the first noun, or second noun, or preposition, stay unchanged (as we saw in the PP Attachment Corpus example data above).

   b) Devise CFG grammar productions to cover some of these cases.

6. ★ **Lexical Acquisition:** As we saw in Chapter 5, it is possible to collapse chunks down to their chunk label. When we do this for sentences involving the word *gave*, we find patterns such as the following:

```
gave NP
gave up NP in NP
gave NP up
gave NP NP
gave NP to NP
```

   a) Use this method to study the complementation patterns of a verb of interest, and write suitable grammar productions.

   b) Identify some English verbs that are near-synonyms, such as the *dumped/filled/loaded* example from earlier in this chapter. Use the chunking method to study the complementation patterns of these verbs. Create a grammar to cover these cases. Can the verbs be freely substituted for each other, or are their constraints? Discuss your findings.

7.5 Parsing

A **parser** processes input sentences according to the productions of a grammar, and builds one or more constituent structures which conform to the grammar. A grammar is a declarative specification of well-formedness. In NLTK, it is just a multi-line string; it is not itself a program that can be used for anything. A parser is a procedural interpretation of the grammar. It searches through the space of trees licensed by a grammar to find one that has the required sentence along its fringe.

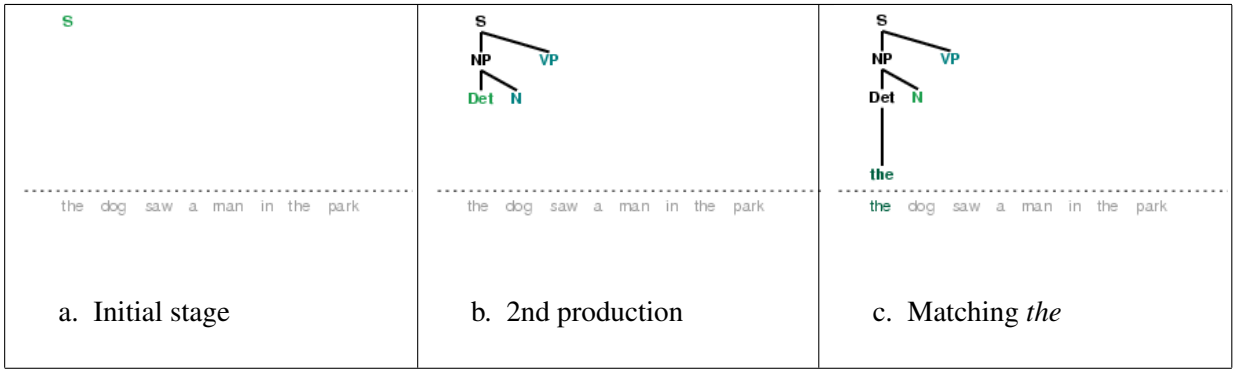
Parsing is important in both linguistics and natural language processing. A parser permits a grammar to be evaluated against a potentially large collection of test sentences, helping linguists to find any problems in their grammatical analysis. A parser can serve as a model of psycholinguistic processing, helping to explain the difficulties that humans have with processing certain syntactic constructions. Many natural language applications involve parsing at some point; for example, we would expect the natural language questions submitted to a question-answering system to undergo parsing as an initial step.

In this section we see two simple parsing algorithms, a top-down method called recursive descent parsing, and a bottom-up method called shift-reduce parsing.

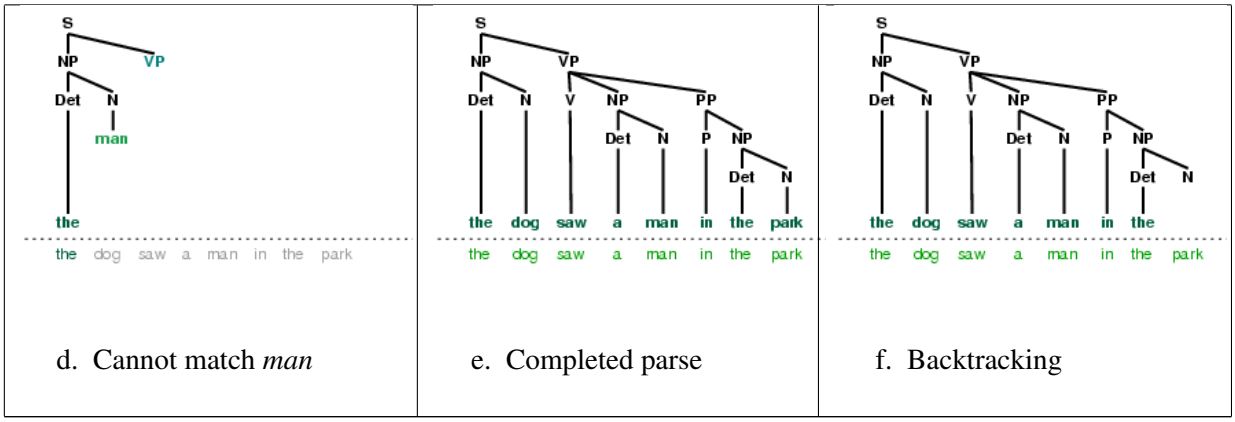
7.5.1 Recursive Descent Parsing

The simplest kind of parser interprets a grammar as a specification of how to break a high-level goal into several lower-level subgoals. The top-level goal is to find an S. The S → NP VP production permits the parser to replace this goal with two subgoals: find an NP, then find a VP. Each of these subgoals can be replaced in turn by sub-sub-goals, using productions that have NP and VP on their left-hand side. Eventually, this expansion process leads to subgoals such as: find the word *telescope*. Such subgoals can be directly compared against the input string, and succeed if the next word is matched. If there is no match the parser must back up and try a different alternative.

The recursive descent parser builds a parse tree during the above process. With the initial goal (find an S), the S root node is created. As the above process recursively expands its goals using the productions of the grammar, the parse tree is extended downwards (hence the name *recursive descent*). We can see this in action using the parser demonstration `nltk_lite.draw.rdparser.demo()`. Six stages of the execution of this parser are shown in Table 7.3.



a. Initial stage b. 2nd production c. Matching *the*



d. Cannot match *man* e. Completed parse f. Backtracking

Table 7.3: Six Stages of a Recursive Descent Parser

During this process, the parser is often forced to choose between several possible productions. For example, in going from step c to step d, it tries to find productions with N on the left-hand side. The first of these is N → *man*. When this does not work it *backtracks*, and tries other N productions in order, under it gets to N → *dog*, which matches the next word in the input sentence. Much later, as shown in step e, it finds a complete parse. This is a tree which covers the entire sentence, without any dangling edges. Once a parse has been found, we can get the parser to look for additional parses. Again it will backtrack and explore other choices of production in case any of them result in a parse.

NLTK provides a recursive descent parser:

```
>>> from nltk_lite import parse
>>> rd_parser = parse.RecursiveDescent(grammar)
>>> sent = list(tokenize.whitespace('Mary saw a dog'))
>>> rd_parser.get_parse_list(sent)
[('S': ('NP': 'Mary') ('VP': ('V': 'saw') ('NP': ('Det': 'a') ('N': 'dog'))))]
```

Note

`parse.RecursiveDescent()` takes an optional parameter `trace`. If `trace` is greater than zero, then the parser will report the steps that it takes as it parses a text.

Recursive descent parsing has three key shortcomings. First, left-recursive productions like NP → NP PP send it into an infinite loop. Second, the parser wastes a lot of time considering words and structures that do not correspond to the input sentence. Third, the backtracking process may discard parsed constituents that will need to be rebuilt again later. For example, backtracking over VP → V NP will discard the subtree created for the NP. If the parser then proceeds with VP → V NP PP, then the NP subtree must be created all over again.

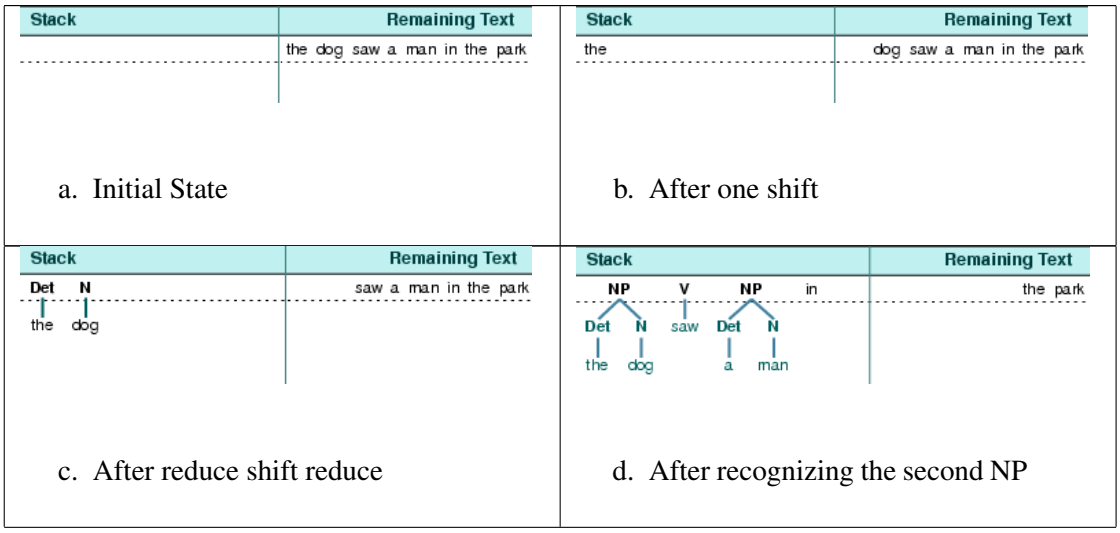
Recursive descent parsing is a kind of **top-down parsing**. Top-down parsers use a grammar to *predict* what the input will be, before inspecting the input! However, since the input is available to the parser all along, it would be more sensible to consider the input sentence from the very beginning. This approach is called **bottom-up parsing**, and we will see an example in the next section.

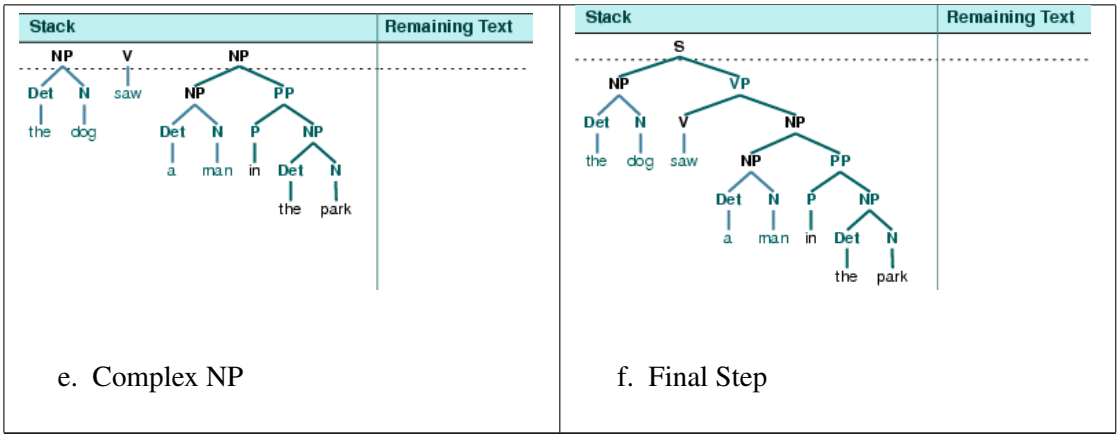
7.5.2 Shift-Reduce Parsing

A simple kind of bottom-up parser is the **shift-reduce parser**. In common with all bottom-up parsers, a shift-reduce parser tries to find sequences of words and phrases that correspond to the *right-hand* side of a grammar production, and replace them with the left-hand side, until the whole sentence is reduced to an S.

The shift-reduce parser repeatedly pushes the next input word onto a stack (Section 6.2.4); this is the **shift** operation. If the top *n* items on the stack match the *n* items on the right-hand side of some production, then they are all popped off the stack, and the item on the left-hand side of the production is pushed on the stack. This replacement of the top *n* items with a single item is the **reduce** operation. (This reduce operation may only be applied to the top of the stack; reducing items lower in the stack must be done before later items are pushed onto the stack.) The parser finishes when all the input is consumed and there is only one item remaining on the stack, a parse tree with an S node as its root.

The shift-reduce parser builds a parse tree during the above process. If the top of stack holds the word *dog*, and if the grammar has a production N → *dog*, then the reduce operation causes the word to be replaced with the parse tree for this production. For convenience we will represent this tree as `N(dog)`. At a later stage, if the top of the stack holds two items `Det(the) N(dog)` and if the grammar has a production NP → DET N then the reduce operation causes these two items to be replaced with `NP(Det(the), N(dog))`. This process continues until a parse tree for the entire sentence has been constructed. We can see this in action using the parser demonstration `nltk_lite.draw.srparser.demo()`. Six stages of the execution of this parser are shown in Figure 7.4.





e. Complex NP f. Final Step

Table 7.4: Six Stages of a Shift-Reduce Parser

NLTK provides `parse.ShiftReduce()`, a simple implementation of a shift-reduce parser. This parser does not implement any backtracking, so it is not guaranteed to find a parse for a text, even if one exists. Furthermore, it will only find at most one parse, even if more parses exist. We can provide an optional `trace` parameter, which controls how verbosely the parser reports the steps that it takes as it parses a text:

```
>>> sr_parse = parse.ShiftReduce(grammar, trace=2)
>>> sent = list(tokenize.whitespace('Mary saw a dog'))
>>> sr_parse.parse(sent)
Parsing 'Mary saw a dog'
    [ * Mary saw a dog]
  S [ 'Mary' * saw a dog]
  R [ <NP> * saw a dog]
  S [ <NP> 'saw' * a dog]
  R [ <NP> <V> * a dog]
  S [ <NP> <V> 'a' * dog]
  R [ <NP> <V> <Det> * dog]
  S [ <NP> <V> <Det> 'dog' * ]
  R [ <NP> <V> <Det> <N> * ]
  R [ <NP> <V> <NP> * ]
  R [ <NP> <VP> * ]
  R [ <S> * ]
  ('S': ('NP': 'Mary') ('VP': ('V': 'saw') ('NP': ('Det': 'a') ('N': 'dog'))))
```

Shift-reduce parsers have a number of problems. A shift-reduce parser may fail to parse the sentence, even though the sentence is well-formed according to the grammar. In such cases, there are no remaining input words to shift, and there is no way to reduce the remaining items on the stack, as exemplified in Table 7.5a. The parser entered this blind alley at an earlier stage shown in Table 7.5b, when it reduced instead of shifted. This situation is called a **shift-reduce conflict**. At another possible stage of processing shown in Table 7.5c, the parser must choose between two possible reductions, both matching the top items on the stack: VP → VP NP PP or NP → NP PP. This situation is called a **reduce-reduce conflict**.

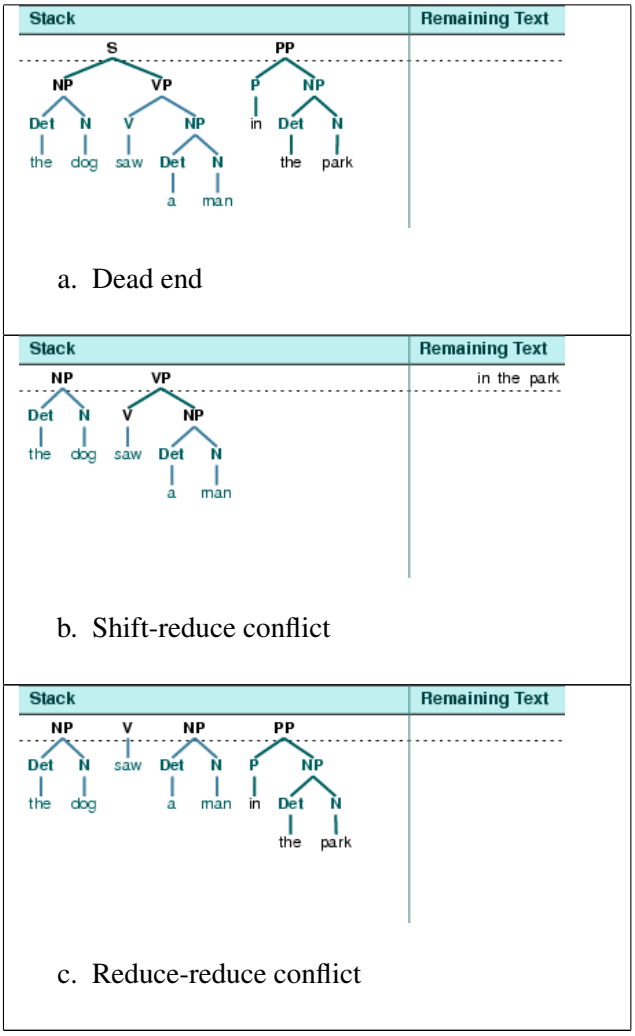


Table 7.5: Conflict in Shift-Reduce Parsing

Shift-reduce parsers may implement policies for resolving such conflicts. For example, they may address shift-reduce conflicts by shifting only when no reductions are possible, and they may address reduce-reduce conflicts by favouring the reduction operation that removes the most items from the stack. No such policies are failsafe however.

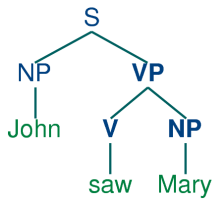
The advantages of shift-reduce parsers over recursive descent parsers is that they only build structure that corresponds to the words in the input. Furthermore, they only build each sub-structure once, e.g. `NP(Det(the), N(man))` is only built and pushed onto the stack a single time, regardless of whether it will later be used by the VP → V NP PP reduction or the NP → NP PP reduction.

7.5.3 The Left-Corner Parser

One of the problems with the recursive descent parser is that it can get into an infinite loop. This is because it applies the grammar productions blindly, without considering the actual input sentence. A left-corner parser is a hybrid between the bottom-up and top-down approaches we have seen.

Grammar (44) allows us to produce the following parse of *John saw Mary*:

(57)



Recall that the grammar in (44) has the following productions for expanding NP:

(58a) NP → DT NOM

(58b) NP → DT NOM PP

(58c) NP → PROPN

Suppose we ask you to first look at tree (57), and then decide which of the NP productions you'd want a recursive descent parser to apply first — obviously, (58c) is the right choice! How do you know that it would be pointless to apply (58a) or (58b) instead? Because neither of these productions will derive a string whose first word is *John*. That is, we can easily tell that in a successful parse of *John saw Mary*, the parser has to expand NP in such a way that NP derives the string *John* α. More generally, we say that a category *B* is a **left-corner** of a tree rooted in *A* if $A \Rightarrow^* B\ \alpha$.

(59)

A **left-corner parser** is a top-down parser with bottom-up filtering. Unlike an ordinary recursive descent parser, it does not get trapped in left recursive productions. Before starting its work, a left-corner parser preprocesses the context-free grammar to build a table where each row contains two cells, the first holding a non-terminal, and the second holding the collection of possible left corners of that non-terminal. Table 7.6 illustrates this for the grammar from (44).

Category	Left-Corners (pre-terminals)
S	NP
NP	Det, PropN
VP	V
PP	P

Table 7.6: Left-Corners in (44)

Each time a production is considered by the parser, it checks that the next input word is compatible with at least one of the pre-terminal categories in the left-corner table.

[TODO: *explain how this effects the action of the parser, and why this solves the problem.*]

7.5.4 Exercises

1. ☼ With pen and paper, manually trace the execution of a recursive descent parser and a shift-reduce parser, for a CFG you have already seen, or one of your own devising.

2. ◑ Compare the performance of the top-down, bottom-up, and left-corner parsers using the same grammar and three grammatical test sentences. Use `timeit` to log the amount of time each parser takes on the same sentence (Section 6.5.4). Write a function which runs all three parsers on all three sentences, and prints a 3-by-3 grid of times, as well as row and column totals. Discuss your findings.

3. ◑ Read up on "garden path" sentences. How might the computational work of a parser relate to the difficulty humans have with processing these sentences? `http://en.wikipedia.org/wiki/Garden_path_sentence`

4. ★ **Left-corner parser:** Develop a left-corner parser based on the recursive descent parser, and inheriting from `ParseI`. (Note, this exercise requires knowledge of Python classes, covered in Chapter 10.)

5. ★ Extend NLTK's shift-reduce parser to incorporate backtracking, so that it is guaranteed to find all parses that exist (i.e. it is **complete**).

7.6 Conclusion

We began this chapter talking about confusing encounters with grammar at school. We just wrote what we wanted to say, and our work was handed back with red marks showing all our grammar mistakes. If this kind of "grammar" seems like secret knowledge, the linguistic approach we have taken in this chapter is quite the opposite: grammatical structures are made explicit as we build trees on top of sentences. We can write down the grammar productions, and parsers can build the trees automatically. This thoroughly objective approach is widely referred to as **generative grammar**.

Note that we have only considered "toy grammars," small grammars that illustrate the key aspects of parsing. But there is an obvious question as to whether the general approach can be scaled up to cover large corpora of natural languages. How hard would it be to construct such a set of productions by hand? In general, the answer is: *very hard*. Even if we allow ourselves to use various formal devices that give much more succinct representations of grammar productions (some of which will be discussed in Chapter 8), it is still extremely difficult to keep control of the complex interactions between the many productions required to cover the major constructions of a language. In other words, it is hard to modularize grammars so that one portion can be developed independently of the other parts. This in turn means that it is difficult to distribute the task of grammar writing across a team of linguists. Another difficulty is that as the grammar expands to cover a wider and wider range of constructions, there is a corresponding increase in the number of analyses which are admitted for any one sentence. In other words, ambiguity increases with coverage.

Despite these problems, there are a number of large collaborative projects which have achieved interesting and impressive results in developing rule-based grammars for several languages. Examples are the Lexical Functional Grammar (LFG) Pargram project (http://www2.parc.com/istl/groups/nltt/pargram/), the Head-Driven Phrase Structure Grammar (HPSG) LinGO Matrix framework (http://www.delph-in.net/matrix/), and the Lexicalized Tree Adjoining Grammar XTAG Project (http://www.cis.upenn.edu/~xtag/).

7.7 Summary (notes)

- Sentences have internal organization, or constituent structure, which can be represented using a tree; notable features of constituent structure are: recursion, heads, complements, modifiers

- A grammar is a compact characterization of a potentially infinite set of sentences; we say that a tree is well-formed according to a grammar, or that a grammar licenses a tree.
- Syntactic ambiguity arises when one sentence has more than one syntactic structure (e.g. prepositional phrase attachment ambiguity).
- A parser is a procedure for finding one or more trees corresponding to a grammatically well-formed sentence.
- A simple top-down parser is the recursive descent parser (summary, problems)
- A simple bottom-up parser is the shift-reduce parser (summary, problems)
- It is difficult to develop a broad-coverage grammar...

7.8 Further Reading

There are many introductory books on syntax. [O'Grady1989LI]_ is a general introduction to linguistics, while [Radford, 1988] provides a gentle introduction to transformational grammar, and can be recommended for its coverage of transformational approaches to unbounded dependency constructions.

[Burton-Roberts, 1997] is very practically oriented textbook on how to analyse constituency in English, with extensive exemplification and exercises. [Huddleston and Pullum, 2002] provides an up-to-date and comprehensive analysis of syntactic phenomena in English.

- LALR(1)
- Marcus parser
- Lexical Functional Grammar (LFG)
  - Pargram project
  - LFG Portal
- Head-Driven Phrase Structure Grammar (HPSG) LinGO Matrix framework
- Lexicalized Tree Adjoining Grammar XTAG Project

Chapter 8

Advanced Parsing

8.1 Introduction

Chapter 7 started with an introduction to constituent structure in English, showing how words in a sentence group together in predictable ways. We showed how to describe this structure using syntactic tree diagrams, and observed that it is sometimes desirable to assign more than one such tree to a given string. In this case, we said that the string was structurally ambiguous; and example was *old men and women*.

Treebanks are language resources in which the syntactic structure of a corpus of sentences has been annotated, usually by hand. However, we would also like to be able to produce trees algorithmically. A context-free phrase structure grammar (CFG) is a formal model for describing whether a given string can be assigned a particular constituent structure. Given a set of syntactic categories, the CFG uses a set of productions to say how a phrase of some category A can be analysed into a sequence of smaller parts α_1 ... α_n. But a grammar is a static description of a set of strings; it does not tell us what sequence of steps we need to take to build a constituent structure for a string. For this, we need to use a parsing algorithm. We presented two such algorithms: Top-Down Recursive Descent (7.5.1) and Bottom-Up Shift-Reduce (7.5.2). As we pointed out, both parsing approaches suffer from important shortcomings. The Recursive Descent parser cannot handle left-recursive productions (e.g., productions such as NP → NP PP), and blindly expands categories top-down without checking whether they are compatible with the input string. The Shift-Reduce parser is not guaranteed to find a valid parse for the input even if one exists, and builds substructure without checking whether it is globally consistent with the grammar. As we will describe further below, the Recursive Descent parser is also inefficient in its search for parses.

So, parsing builds trees over sentences, according to a phrase structure grammar. Now, all the examples we gave in Chapter 7 only involved toy grammars containing a handful of productions. What happens if we try to scale up this approach to deal with realistic corpora of language? Unfortunately, as the coverage of the grammar increases and the length of the input sentences grows, the number of parse trees grows rapidly. In fact, it grows at an astronomical rate.

Let's explore this issue with the help of a simple example. The word *fish* is both a noun and a verb. We can make up the sentence *fish fish fish*, meaning *fish like to fish for other fish*. (Try this with *police* if you prefer something more sensible.) Here is a toy grammar for the "fish" sentences.

```
>>> from nltk_lite.parse import cfg, chart
>>> grammar = cfg.parse_cfg("""
... S -> NP V NP
... NP -> NP Sbar
```

```
... Sbar -> NP V
... NP -> 'fish'
... V -> 'fish'
... """)
```

Now we can try parsing a longer sentence, *fish fish fish fish fish*, which amongst other things, means 'fish that other fish fish are in the habit of fishing fish themselves'. We use the NLTK chart parser, which is presented later on in this chapter. This sentence has two readings.

```
>>> tokens = ["fish"] * 5
>>> cp = chart.ChartParse(grammar, chart.TD_STRATEGY)
>>> for tree in cp.get_parse_list(tokens):
...     print tree
(S:
  (NP: 'fish')
  (V: 'fish')
  (NP: (NP: 'fish') (Sbar: (NP: 'fish') (V: 'fish'))))
(S:
  (NP: (NP: 'fish') (Sbar: (NP: 'fish') (V: 'fish')))
  (V: 'fish')
  (NP: 'fish'))
```

As the length of this sentence goes up (3, 5, 7, ...) we get the following numbers of parse trees: 1; 2; 5; 14; 42; 132; 429; 1,430; 4,862; 16,796; 58,786; 208,012; ... (These are the *Catalan numbers*, which we saw in an exercise in Section 6.5). The last of these is for a sentence of length 23, the average length of sentences in the WSJ section of Penn Treebank. For a sentence of length 50 there would be over 10^{12} parses, and this is only half the length of the Piglet sentence (Section 7.2), which young children process effortlessly. No practical NLP system could construct all millions of trees for a sentence and choose the appropriate one in the context. It's clear that humans don't do this either!

Note that the problem is not with our choice of example. [Church and Patil, 1982] point out that the syntactic ambiguity of PP attachment in sentences like (60) also grows in proportion to the Catalan numbers.

(60) Put the block in the box on the table.

So much for structural ambiguity; what about lexical ambiguity? As soon as we try to construct a broad-coverage grammar, we are forced to make lexical entries highly ambiguous for their part of speech. In a toy grammar, *a* is only a determiner, *dog* is only a noun, and *runs* is only a verb. However, in a broad-coverage grammar, *a* is also a noun (e.g. *part a*), *dog* is also a verb (meaning to follow closely), and *runs* is also a noun (e.g. *ski runs*). In fact, all words can be referred to by name: e.g. *the verb 'ate' is spelled with three letters*; in speech we do not need to supply quotation marks. Furthermore, it is possible to *verb* most nouns. Thus a parser for a broad-coverage grammar will be overwhelmed with ambiguity. Even complete gibberish will often have a reading, e.g. *the a are of I*. As [Abney, 1996b] has pointed out, this is not word salad but a grammatical noun phrase, in which *are* is a noun meaning a hundredth of a hectare (or 100 sq m), and *a* and *I* are nouns designating coordinates, as shown in Figure 8.1.

Even though this phrase is unlikely, it is still grammatical and a a broad-coverage parser should be able to construct a parse tree for it. Similarly, sentences which seem to be unambiguous, such as *John saw Mary*, turn out to have other readings we would not have anticipated (as Abney explains). This ambiguity is unavoidable, and leads to horrendous inefficiency in parsing seemingly inoccuous sentences.

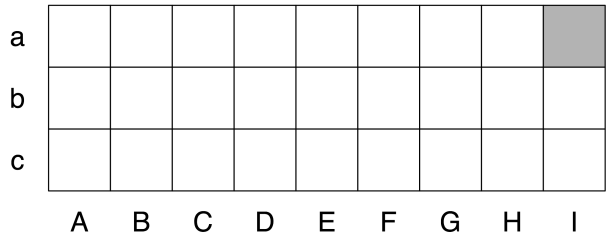


Figure 8.1: The a are of I

Let's look more closely at this issue of efficiency. The top-down recursive-descent parser presented in Chapter 7 can be very inefficient, since it often builds and discards the same sub-structure many times over. We see this in Figure 8.1, where a phrase *the block* is identified as a noun phrase several times, and where this information is discarded each time we backtrack.

Note

You should try the recursive-descent parser demo if you haven't already: `nltk_lite.draw.srparser.demo()`

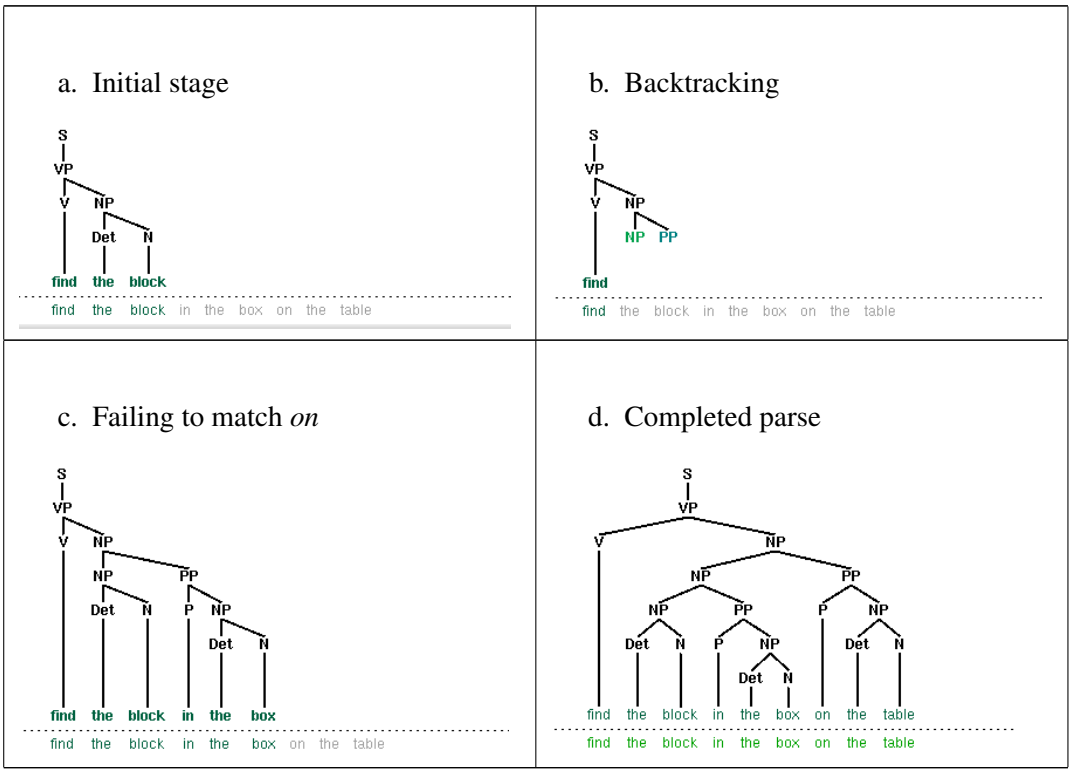


Table 8.1: Backtracking and Repeated Parsing of Subtrees

In this chapter, we will present two independent methods for dealing with ambiguity. The first is *chart parsing*, which uses the algorithmic technique of dynamic programming to derive the parses of an ambiguous sentence more *efficiently*. The second is *probabilistic parsing*, which allows us to *rank* the parses of an ambiguous sentence on the basis of evidence from corpora.

8.2 Chart Parsing

In the introduction to this chapter, we pointed out that the simple parsers discussed in Chapter 7 suffered from limitations in both completeness and efficiency. In order to remedy these, we will apply the algorithm design technique of *dynamic programming* to the parsing problem. As we saw in Section 6.5.3, dynamic programming stores intermediate results and re-uses them when appropriate, achieving significant efficiency gains. This technique can be applied to syntactic parsing, allowing us to store

partial solutions to the parsing task and then look them up as necessary in order to efficiently arrive at a complete solution. This approach to parsing is known as **chart parsing**, and is the focus of this section.

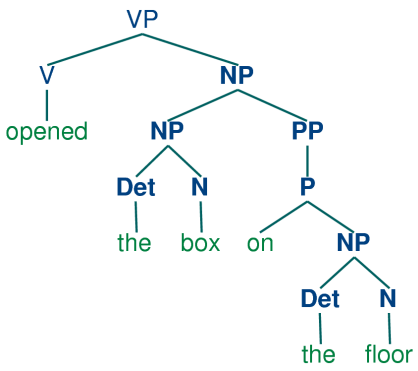
8.2.1 Well-formed Substring Tables

Let's start off by defining a simple grammar.

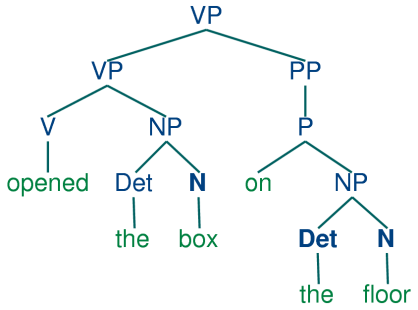
```
>>> from nltk_lite.parse import cfg
>>> grammar = cfg.parse_cfg("""
... S -> NP VP
... PP -> P NP
... NP -> Det N | NP PP
... VP -> V NP | VP PP
... Det -> 'the'
... N -> 'kids' | 'box' | 'floor'
... V -> 'opened'
... P -> 'on'
... """)
```

As you can see, this grammar allows the VP *opened the box on the floor* to be analysed in two ways, depending on where the PP is attached.

(61a)



(61b)



Dynamic programming allows us to build the PP *on the floor* just once. The first time we build it we save it in a table, then we look it up when we need to use it as a subconstituent of either the object NP or the higher VP. This table is known as a **well-formed substring table** (or WFST for short). We will show how to construct the WFST bottom-up so as to systematically record what syntactic constituents have been found.

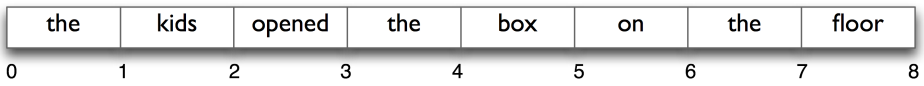


Figure 8.2: Slice Points in the Input String

Let's set our input to be the sentence *the kids opened the box on the floor*. It is helpful to think of the input as being indexed like a Python list. We have illustrated this in Figure 8.2.
This allows us to say that, for instance, the word *opened* spans (2, 3) in the input. This is reminiscent of the slice notation:

```
>>> tokens = ["the", "kids", "opened", "the", "box", "on", "the", "floor"]
>>> tokens[2:3]
['opened']
```

In a WFST, we record the position of the words by filling in cells in a triangular matrix: the vertical axis will denote the start position of a substring, while the horizontal axis will denote the end position (thus *opened* will appear in the cell with coordinates (2, 3)). To simplify this presentation, we will assume each word has a unique lexical category, and we will store this (not the word) in the matrix. So cell (2, 3) will contain the entry V. More generally, if our input string is $a_1 a_2 \ldots a_n$, and our grammar contains a production of the form $A \rightarrow a_i$, then we add A to the cell (i-1, i).

So, for every word in `tokens`, we can look up in our grammar what category it belongs to.

```
>>> grammar.productions(rhs=tokens[2])
[V -> 'opened']
```

For our WFST, we create an $(n-1) \times (n-1)$ matrix as a list of lists in Python, and initialize it with the lexical categories of each token, in the `init_wfst()` function in Listing 8.1. We also define a utility function `display()` to pretty-print the WFST for us. As expected, there is a V in cell (2, 3).

Returning to our tabular representation, given that we have DET in cell (0, 1), and N in cell (1, 2), what should we put into cell (0, 2)? In other words, what syntactic category derives *the kids*? We have already established that DET derives *the* and N derives *kids*, so we need to find a production of the form $A \rightarrow$ DET N, that is, a production whose right hand side matches the categories in the cells we have already found. From the grammar, we know that we can enter NP in cell (0,2).

More generally, we can enter A in (i, j) if there is a production $A \rightarrow B\ C$, and we find nonterminal B in (i, k) and C in (k, j). Listing 8.1 uses this inference step to complete the WFST.

> **Note**
>
> To help us easily retrieve productions by their right hand sides, we create an index for the grammar. This is an example of a space-time trade-off: we do a reverse lookup on the grammar, instead of having to check through entire list of productions each time we want to look up via the right hand side.

We conclude that there is a parse for the whole input string once we have constructed an S node that covers the whole input, from position *0* to position *8*; i.e., we can conclude that S $\Rightarrow^*$ $a_1 a_2 \ldots a_n$.

Notice that we have not used any built-in parsing functions here. We've implemented a complete, primitive chart parser from the ground up!

Listing 29 Acceptor Using Well-Formed Substring Table (based on CYK algorithm)

```
def init_wfst(tokens, grammar):
    numtokens = len(tokens)
    wfst = [['.' for i in range(numtokens+1)] for j in range(numtokens+1)]
    for i in range(numtokens):
        productions = grammar.productions(rhs=tokens[i])
        wfst[i][i+1] = productions[0].lhs()
    return wfst
def complete_wfst(wfst, tokens, trace=False):
    index = {}
    for prod in grammar.productions():
        index[prod.rhs()] = prod.lhs()
    numtokens = len(tokens)
    for span in range(2, numtokens+1):
        for start in range(numtokens+1-span):
            end = start + span
            for mid in range(start+1, end):
                nt1, nt2 = wfst[start][mid], wfst[mid][end]
                if (nt1,nt2) in index:
                    if trace:
                        print "[%s] %3s [%s] %3s [%s] ==> [%s] %3s [%s]" % \
                        (start, nt1, mid, nt2, end, start, index[(nt1,nt2)], end)
                    wfst[start][end] = index[(nt1,nt2)]
    return wfst
def display(wfst, tokens):
    import string
    print '\nWFST ' + string.join([("%-4d" % i) for i in range(1, len(wfst))])
    for i in range(len(wfst)-1):
        print "%d   " % i,
        for j in range(1, len(wfst)):
            print "%-4s" % wfst[i][j],
        print

>>> wfst0 = init_wfst(tokens, grammar)
>>> display(wfst0, tokens)
WFST 1    2    3    4    5    6    7    8
0    Det  .    .    .    .    .    .    .
1    .    N    .    .    .    .    .    .
2    .    .    V    .    .    .    .    .
3    .    .    .    Det  .    .    .    .
4    .    .    .    .    N    .    .    .
5    .    .    .    .    .    P    .    .
6    .    .    .    .    .    .    Det  .
7    .    .    .    .    .    .    .    N
>>> wfst1 = complete_wfst(wfst0, tokens)
>>> display(wfst1, tokens)
WFST 1    2    3    4    5    6    7    8
0    Det  NP   .    .    S    .    .    S
1    .    N    .    .    .    .    .    .
2    .    .    V    .    VP   .    .    VP
3    .    .    .    Det  NP   .    .    NP
4    .    .    .    .    N    .    .    .
5    .    .    .    .    .    P    .    PP
6    .    .    .    .    .    .    Det  NP
7    .    .    .    .    .    .    .    N
```

8.2.2 Charts

By setting `trace` to `True` when calling the function `complete_wfst()`, we get additional output.

```
>>> wfst1 = complete_wfst(wfst0, tokens, trace=True)
[0] Det [1]   N [2] ==> [0]  NP [2]
[3] Det [4]   N [5] ==> [3]  NP [5]
[6] Det [7]   N [8] ==> [6]  NP [8]
[2]   V [3]  NP [5] ==> [2]  VP [5]
[5]   P [6]  NP [8] ==> [5]  PP [8]
[0]  NP [2]  VP [5] ==> [0]   S [5]
[3]  NP [5]  PP [8] ==> [3]  NP [8]
[2]   V [3]  NP [8] ==> [2]  VP [8]
[2]  VP [5]  PP [8] ==> [2]  VP [8]
[0]  NP [2]  VP [8] ==> [0]   S [8]
```

For example, this says that since we found `Det` at `wfst[0][1]` and `N` at `wfst[1][2]`, we can add `NP` to `wfst[0][2]`. The same information can be represented in a directed acyclic graph, as shown in Figure 8.2(a). This graph is usually called a **chart**. Figure 8.2(b) is the corresponding graph representation, where we add a new edge labeled NP to cover the input from *0* to *2*.

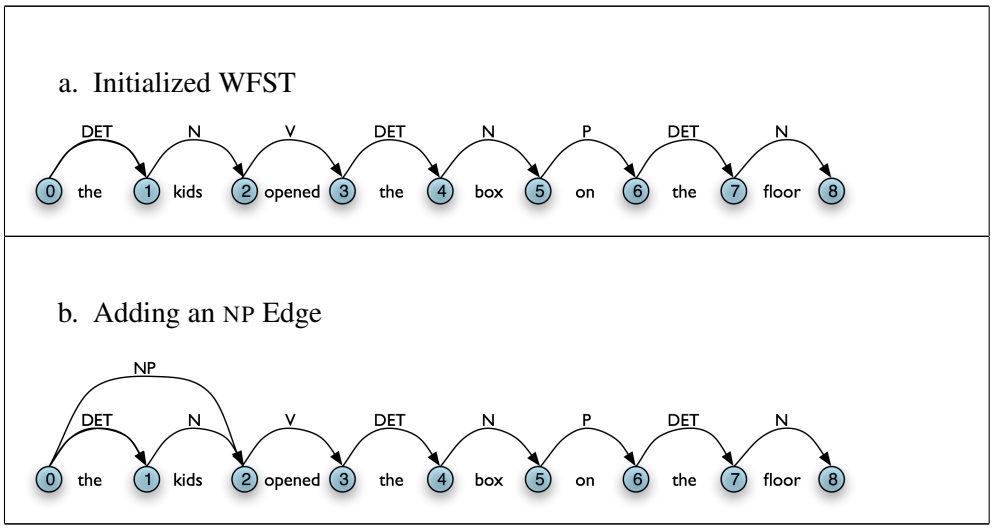


Table 8.2: A Graph Representation for the WFST

(Charts are more general than the WFSTs we have seen, since they can hold multiple hypotheses for a given span.)

A WFST is a data structure that can be used by a variety of parsing algorithms. The particular method for constructing a WFST that we have just seen and has some shortcomings. First, as you can see, the WFST is not itself a parse tree, so the technique is strictly speaking **recognizing** that a sentence is admitted by a grammar, rather than parsing it. Second, it requires every non-lexical grammar production to be *binary* (see Section 8.5.1). Although it is possible to convert an arbitrary CFG into this form, we would prefer to use an approach without such a requirement. Third, as a bottom-up approach it is potentially wasteful, being able to propose constituents in locations that would not be licensed by the grammar. Finally, the WFST did not represent the structural ambiguity in the sentence (i.e. the two

verb phrase readings). The VP in cell (*2,8*) was actually entered twice, once for a V NP reading, and oce for a VP PP reading. In the next section we will address these issues.

8.2.3 Exercises

1. ☼ Consider the sequence of words: *Buffalo buffalo Buffalo buffalo buffalo buffalo Buffalo buffalo*. This is a grammatically correct sentence, as explained at `http://en.wikipedia.org/wiki/Buffalo_buffalo_Buffalo_buffalo_buffalo_buffalo_Buffa`. Consider the tree diagram presented on this Wikipedia page, and write down a suitable grammar. Normalise case to lowercase, to simulate the problem that a listener has when hearing this sentence. Can you find other parses for this sentence? How does the number of parse trees grow as the sentence gets longer? (More examples of these sentences can be found at `http://en.wikipedia.org/wiki/List_of_homophonous_phrases`).

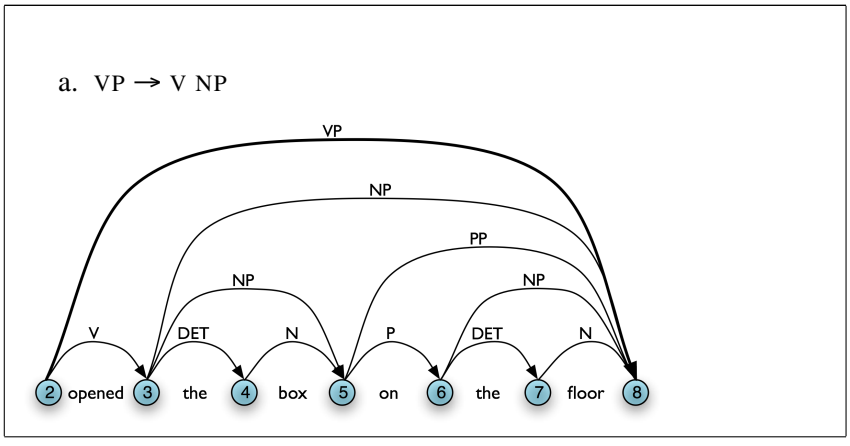
2. ◑ Consider the algorithm in Listing 8.1. Can you explain why parsing context-free grammar is proportional to n^3?

3. ◑ Modify the functions `init_wfst()` and `complete_wfst()` so that the contents of each cell in the WFST is a set of non-terminal symbols rather than a single non-terminal.

4. ★ Modify the functions `init_wfst()` and `complete_wfst()` so that when a non-terminal symbol is added to a cell in the WFST, it includes a record of the cells from which it was derived. Implement a function which will convert a WFST in this form to a parse tree.

8.3 Active Charts

One important aspect of the tabular approach to parsing can be seen more clearly if we look at the graph representation: given our grammar, there are two different ways to derive a top-level VP for the input, as shown in Table 8.3(a,b). In our graph representation, we simply combine the two sets of edges to yield Table 8.3(c).



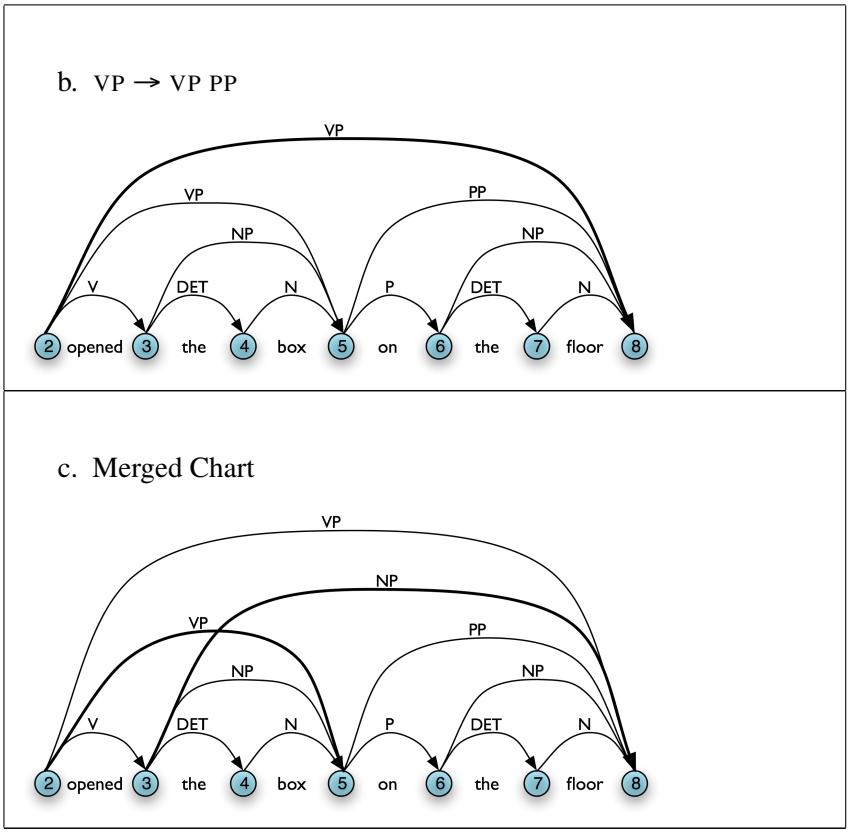


Table 8.3: Combining Multiple Parses in a Single Chart

However, given a WFST we cannot necessarily read off the justification for adding a particular edge. For example, in 8.3(b), `[Edge: VP, 2:8]` might owe its existence to a production VP → V NP PP. Unlike phrase structure trees, a WFST does not encode a relation of immediate dominance. In order to make such information available, we can label edges not just with a non-terminal category, but with the whole production which justified the addition of the edge. This is illustrated in Figure 8.3.

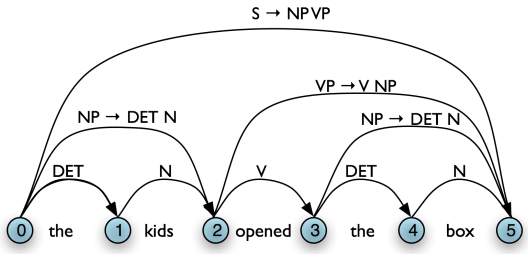


Figure 8.3: Chart Annotated with Productions

In general, a chart parser hypothesizes constituents (i.e. adds edges) based on the grammar, the tokens, and the constituents already found. Any constituent that is compatible with the current knowledge can be hypothesized; even though many of these hypothetical constituents will never be used in the final result. A WFST just records these hypotheses.

All of the edges that we've seen so far represent complete constituents. However, as we will see, it is helpful to hypothesize *incomplete* constituents. For example, the work done by a parser in processing the production *VP → V NP PP* can be reused when processing *VP → V NP*. Thus, we will record the hypothesis that "the V constituent *likes* is the beginning of a VP."

We can record such hypotheses by adding a **dot** to the edge's right hand side. Material to the left of the dot specifies what the constituent starts with; and material to the right of the dot specifies what still needs to be found in order to complete the constituent. For example, the edge in the Figure 8.4 records the hypothesis that "a VP starts with the V *likes*, but still needs an NP to become complete":

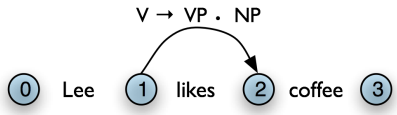


Figure 8.4: Chart Containing Incomplete VP Edge

These **dotted edges** are used to record all of the hypotheses that a chart parser makes about constituents in a sentence. Formally a dotted edge $[A \rightarrow c_1 \ldots c_d \bullet c_{d+1} \ldots c_n, (i, j)]$ records the hypothesis that a constituent of type A with span (i, j) starts with children $c_1 \ldots c_d$, but still needs children $c_{d+1} \ldots c_n$ to be complete ($c_1 \ldots c_d$ and $c_{d+1} \ldots c_n$ may be empty). If $d = n$, then $c_{d+1} \ldots c_n$ is empty and the edge represents a complete constituent and is called a **complete edge**. Otherwise, the edge represents an incomplete constituent, and is called an **incomplete edge**. In Figure 8.4(a), [VP → V NP •, (1, 3)] is a complete edge, and [VP → V • NP, (1, 2)] is an incomplete edge.

If $d = 0$, then $c_1 \ldots c_n$ is empty and the edge is called a **self-loop edge**. This is illustrated in Table 8.4(b). If a complete edge spans the entire sentence, and has the grammar's start symbol as its left-hand side, then the edge is called a **parse edge**, and it encodes one or more parse trees for the sentence. In Table 8.4(c), [S → NP VP •, (0, 3)] is a parse edge.

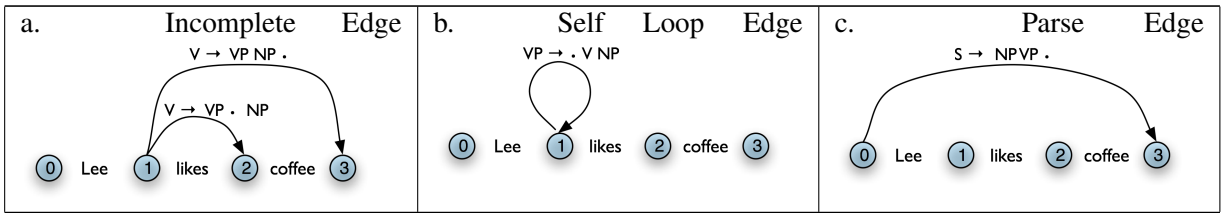


Table 8.4: Chart Terminology

8.3.1 The Chart Parser

To parse a sentence, a chart parser first creates an empty chart spanning the sentence. It then finds edges that are licensed by its knowledge about the sentence, and adds them to the chart one at a time until one or more parse edges are found. The edges that it adds can be licensed in one of three ways:

1. The *input* can license an edge. In particular, each word w_i in the input licenses the complete edge $[w_i \rightarrow \bullet, (i, i{+}1)]$.
2. The *grammar* can license an edge. In particular, each grammar production A → α licenses the self-loop edge $[A \rightarrow \bullet\ \alpha, (i, i)]$ for every i, $0 \leq i < n$.
3. The *current chart contents* can license an edge.

However, it is not wise to add *all* licensed edges to the chart, since many of them will not be used in any complete parse. For example, even though the edge in the following chart is licensed (by the grammar), it will never be used in a complete parse:

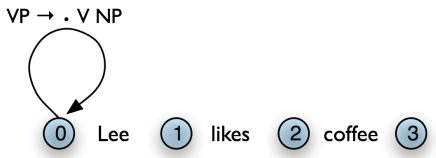


Figure 8.5: Chart Containing Redundant Edge

Chart parsers therefore use a set of **rules** to heuristically decide when an edge should be added to a chart. This set of rules, along with a specification of when they should be applied, forms a **strategy**.

8.3.2 The Fundamental Rule

One rule is particularly important, since it is used by every chart parser: the **Fundamental Rule**. This rule is used to combine an incomplete edge that's expecting a nonterminal *B* with a following, complete edge whose left hand side is *B*.

(62) **Fundamental Rule**

```
If the chart contains the edges
  [A → α • B β , (i, j)]
  [B → γ • , (j, k)]
then add the new edge
  [A → α B • β , (i, k)]
where α, β, and γ are (possibly empty) sequences
of terminals or non-terminals
```

Note that the dot has moved one place to the right, and the span of this new edige is the combined span of the other two. Note also that in adding this new edge we do not remove the other two, because they might be used again.

A somewhat more intuitive version of the operation of the Fundamental Rule can be given using chart diagrams. Thus, if we have a chart of the form shown in Table 8.5(a), then we can add a new complete edge as shown in Table 8.5(b).

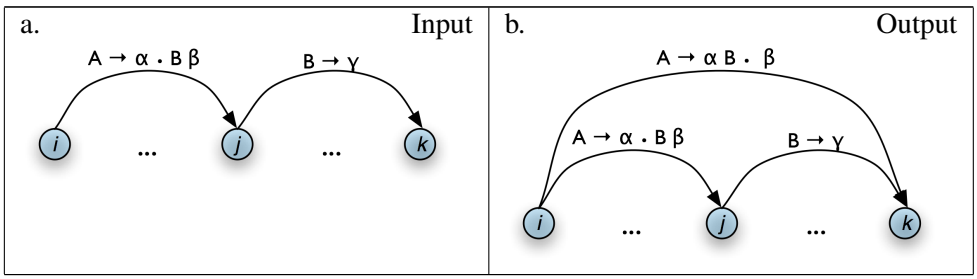


Table 8.5: Fundamental Rule

[2]The Fundamental Rule corresponds to the Completer function in the Earley algorithm; cf. [Jurafsky and Martin, 2000].

8.3.3 Bottom-Up Parsing

As we saw in Chapter 7, bottom-up parsing starts from the input string, and tries to find sequences of words and phrases that correspond to the *right hand* side of a grammar production. The parser then replaces these with the left-hand side of the production, until the whole sentence is reduced to an S. Bottom-up chart parsing is an extension of this approach in which hypotheses about structure are recorded as edges on a chart. In terms of our earlier terminology, bottom-up chart parsing can be seen as a parsing strategy; in other words, bottom-up is a particular choice of heuristics for adding new edges to a chart.

The general procedure for chart parsing is inductive: we start with a base case, and then show how we can move from a given state of the chart to a new state. Since we are working bottom-up, the base case for our induction will be determined by the words in the input string, so we add new edges for each word. Now, for the induction step, suppose the chart contains an edge labeled with constituent *A*. Since we are working bottom-up, we want to build constituents which can have an *A* as a daughter. In other words, we are going to look for productions of the form $B \rightarrow A\ \beta$ and use these to label new edges.

Let's look at the procedure a bit more formally. To create a bottom-up chart parser, we add to the Fundamental Rule two new rules: the **Bottom-Up Initialization Rule**; and the **Bottom-Up Predict Rule**. The Bottom-Up Initialization Rule says to add all edges licensed by the input.

(63) **Bottom-Up Initialization Rule**

For every word w_i add the edge
$[w_i \rightarrow \bullet\ ,\ (i,\ i+1)]$

Table 8.6(a) illustrates this rule using the chart notation, while Table 8.6(b) shows the bottom-up initialization for the input *Lee likes coffee*.

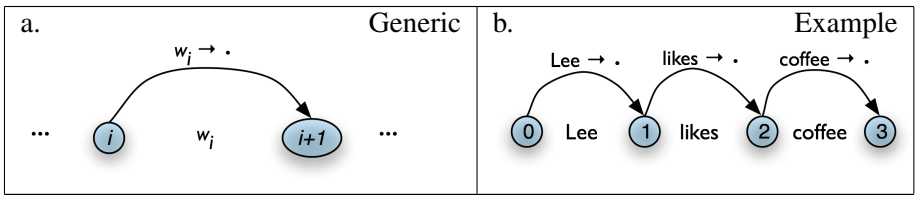


Table 8.6: Bottom-Up Initialization Rule

Notice that the dot on the right hand side of these productions is telling us that we have complete edges for the lexical items. By including this information, we can give a uniform statement of how the Fundamental Rule operates in Bottom-Up parsing, as we will shortly see.

Next, suppose the chart contains a complete edge *e* whose left hand category is *A*. Then the Bottom-Up Predict Rule requires the parser to add a self-loop edge at the left boundary of *e* for each grammar production whose right hand side begins with category *A*.

(64) **Bottom-Up Predict Rule**

If the chart contains the complete edge
$[A \rightarrow \alpha\ \bullet\ ,\ (i,\ j)]$
and the grammar contains the production
$B \rightarrow A\ \beta$
then add the self-loop edge

[B → • A β , (i, i)]

Graphically, if the chart looks as in Figure 8.7(a), then the Bottom-Up Predict Rule tells the parser to augment the chart as shown in Figure 8.7(b).

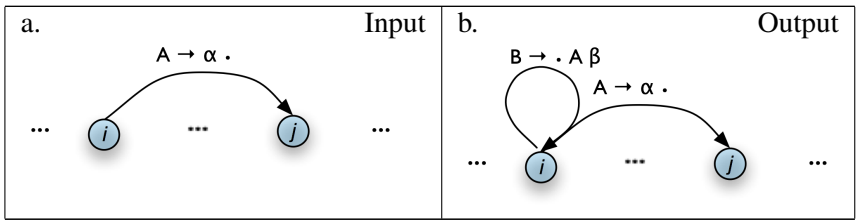


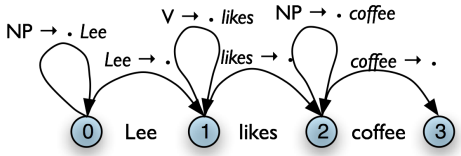
Table 8.7: Bottom-Up Prediction Rule

To continue our earlier example, let's suppose that our grammar contains the lexical productions shown in (65a). This allows us to add three self-loop edges to the chart, as shown in (65b).

(65a) NP → *Lee* | *coffee*

V → *likes*

(65b)



Once our chart contains an instance of the pattern shown in Figure 8.7(b), we can use the Fundamental Rule to add an edge where we have "moved the dot" one position to the right, as shown in Figure 8.8 (we have omitted the self-loop edges for simplicity.)

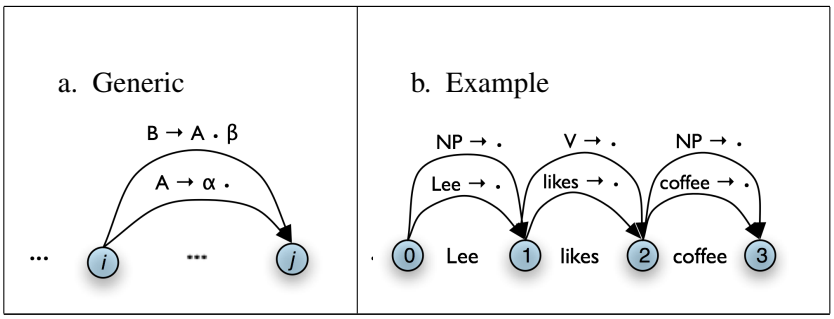


Table 8.8: Fundamental Rule used in Bottom-Up Parsing

We will now be able to add new self-loop edges such as [S → • NP VP, (0, 0)] and [VP → • VP NP, (1, 1)], and use these to build more complete edges.

Using these three productions, we can parse a sentence as shown in (66).

(66) **Bottom-Up Strategy**

Create an empty chart spanning the sentence.

```
Apply the Bottom-Up Initialization Rule to each word.
Until no more edges are added:
  Apply the Bottom-Up Predict Rule everywhere it applies.
  Apply the Fundamental Rule everywhere it applies.
Return all of the parse trees corresponding to the parse edges in the chart
```

NLTK provides a useful interactive tool for visualizing the way in which charts are built, `nltk_lite.draw.chart.demo()`. The tool comes with a pre-defined input string and grammar, but both of these can be readily modified with options inside the *Edit* menu. Figure 8.6 illustrates a window after the grammar has been updated:

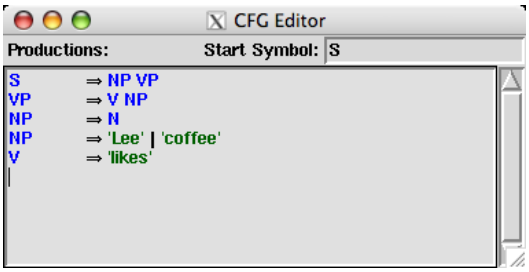


Figure 8.6: Modifing the *demo()* grammar

> **Note**
>
> To get the symbol ⇒ illustrated in Figure 8.6. you just have to type the keyboard characters *'->'*.

Figure 8.7 illustrates the tool interface. In order to invoke a rule, you simply click one of the green buttons at the bottom of the window. We show the state of the chart on the input *Lee likes coffee* after three applications of the Bottom-Up Initialiation Rule, followed by successive applications of the Bottom-Up Predict Rule and the Fundamental Rule.

Notice that in the topmost pane of the window, there is a partial tree showing that we have constructed an S with an NP subject in the expectation that we will be able to find a VP.

8.3.4 Top-Down Parsing

Top-down chart parsing works in a similar way to the recursive descent parser discussed in Chapter 7, in that it starts off with the top-level goal of finding an S. This goal is then broken into the subgoals of trying to find constituents such as NP and VP which can be immediatedly dominated by S. To create a top-down chart parser, we use the Fundamental Rule as before plus three other rules: the **Top-Down Initialization Rule**, the **Top-Down Expand Rule**, and the **Top-Down Match Rule**. The Top-Down Initialization Rule in (67) captures the fact that the root of any parse must be the start symbol S. It is illustrated graphically in Table 8.9.

(67) **Top-Down Initialization Rule**

```
For every grammar production of the form:
  s → α
add the self-loop edge:
  [s →  • α, (0, 0)]
```

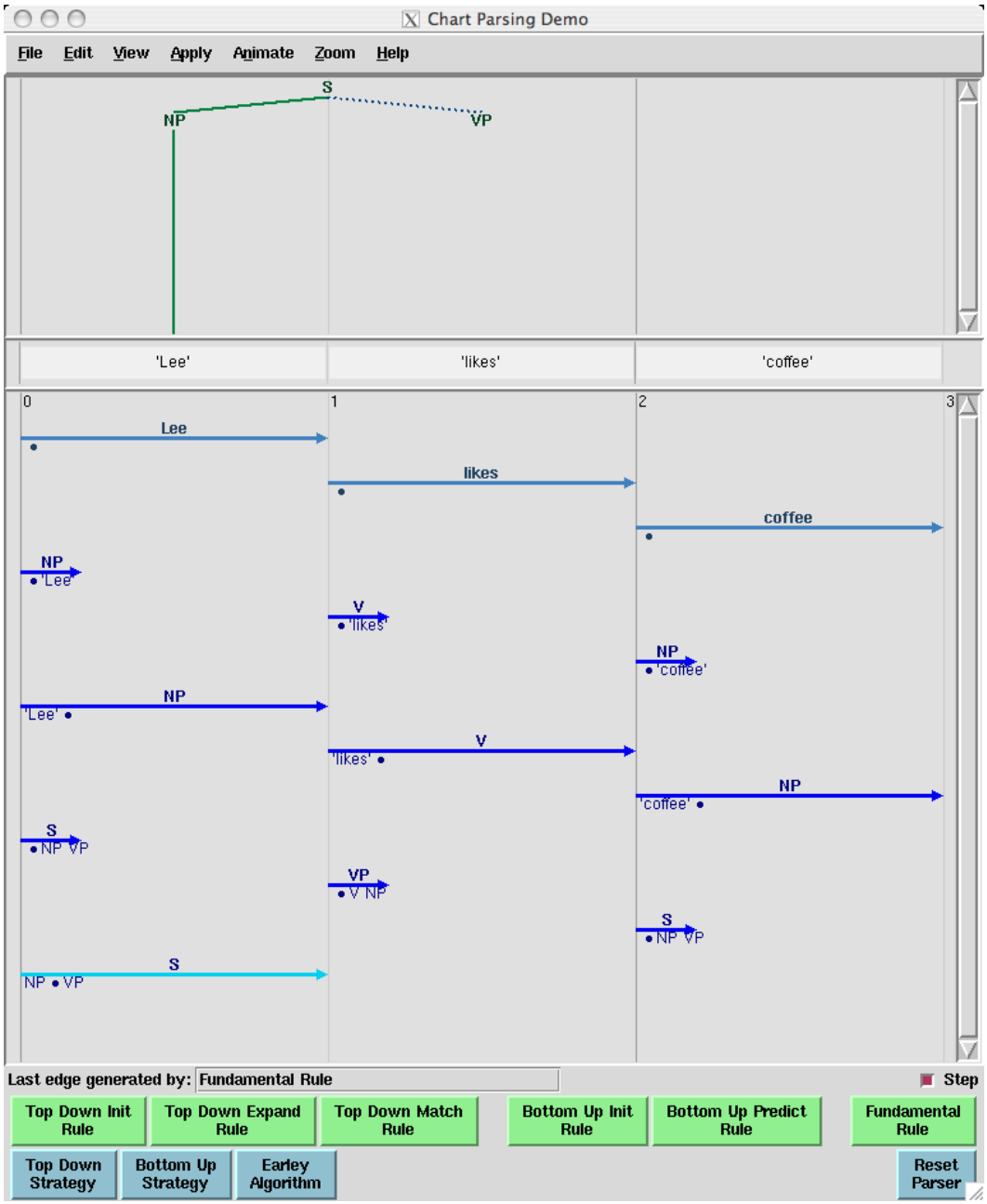


Figure 8.7: Incomplete chart for *Lee likes coffee*

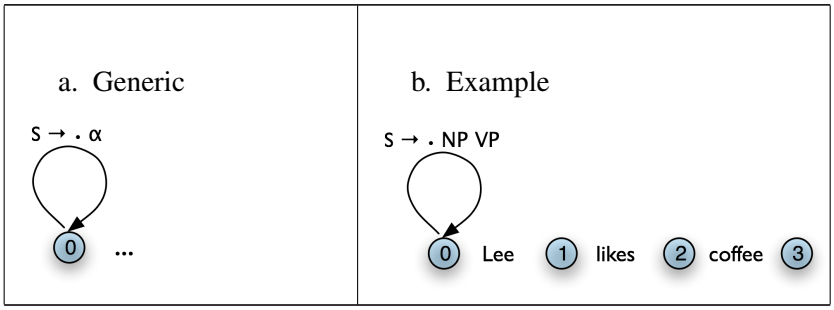


Table 8.9: Top-Down Initialization Rule

As we mentioned before, the dot on the right hand side of a production records how far our goals have been satisfied. So in Figure 8.9(b), we are predicting that we will be able to find an NP and a VP, but have not yet satisfied these subgoals. So how do we pursue them? In order to find an NP, for instance, we need to invoke a production which has NP on its left hand side. The step of adding the required edge to the chart is accomplished with the Top-Down Expand Rule (68). This tells us that if our chart contains an incomplete edge whose dot is followed by a nonterminal *B*, then the parser should add any self-loop edges licensed by the grammar whose left-hand side is *B*.

(68) **Top-Down Expand Rule**

```
If the chart contains the incomplete edge
  [A → α • B β , (i, j)]
then for each grammar production
  B → γ
add the edge
  [B → • γ , (j, j)]
```

Thus, given a chart that looks like the one in Table 8.10(a), the Top-Down Expand Rule augments it with the edge shown in Table 8.10(b). In terms of our running example, we now have the chart shown in Table 8.10(c).

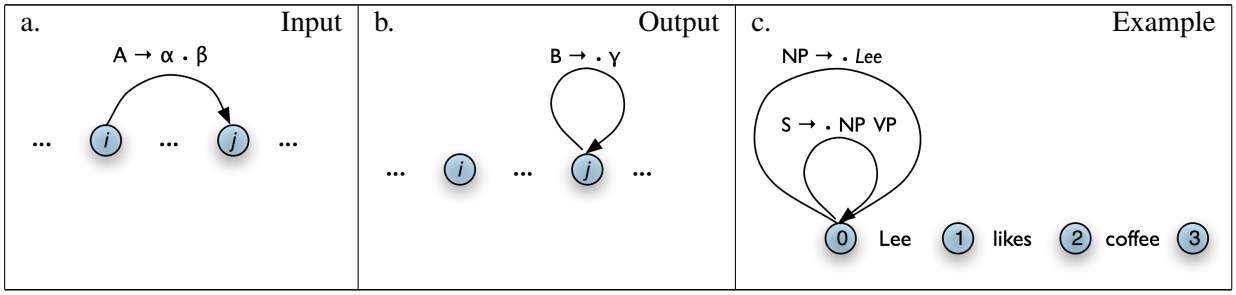


Table 8.10: Top-Down Expand Rule

The Top-Down Match rule allows the predictions of the grammar to be matched against the input string. Thus, if the chart contains an incomplete edge whose dot is followed by a terminal *w*, then the parser should add an edge if the terminal corresponds to the current input symbol.

(69) **Top-Down Match Rule**

```
If the chart contains the incomplete edge
  [A → α • wj β, (i, j)],
where wj is the j th word of the input,
then add a new complete edge
  [wj → • , (j, j+1)]
```

Graphically, the Top-Down Match rule takes us from Table 8.11(a), to Table 8.11(b).

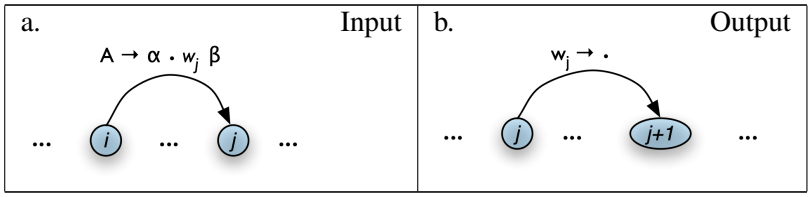


Table 8.11: Top-Down Match Rule

Figure 8.12(a) illustrates how our example chart after applying the Top-Down Match rule. What rule is relevant now? The Fundamental Rule. If we remove the self-loop edges from Figure 8.12(a) for simplicity, the Fundamental Rule gives us Figure 8.12(b).

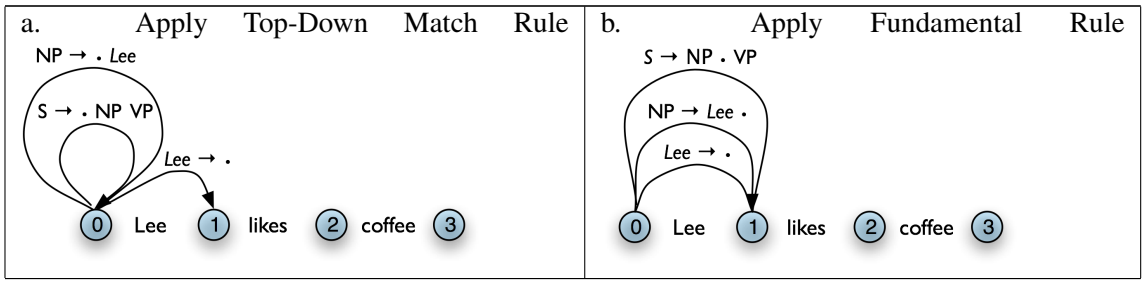


Table 8.12: Top-Down Example (cont)

Using these four rules, we can parse a sentence top-down as shown in (70).

(70) **Top-Down Strategy**

```
Create an empty chart spanning the sentence.
Apply the Top-Down Initialization Rule.
Until no more edges are added:
  Apply the Top-Down Expand Rule everywhere it applies.
  Apply the Top-Down Match Rule everywhere it applies.
  Apply the Fundamental Rule everywhere it applies.
Return all of the parse trees corresponding to the parse edges in
the chart.
```

We encourage you to experiment with the NLTK chart parser demo, as before, in order to test out the top-down strategy yourself.

8.3.5 The Earley Algorithm

The Earley algorithm [Earley, 1970] is a parsing strategy that resembles the Top-Down Strategy, but deals more efficiently with matching against the input string. Table 8.13 shows the correspondence between the parsing rules introduced above and the rules used by the Earley algorithm.

Top-Down/Bottom-Up	**Earley**
Top-Down Initialization Rule Top-Down Expand Rule	Predictor Rule
Top-Down/Bottom-Up Match Rule	Scanner Rule
Fundamental Rule	Completer Rule

Table 8.13: Terminology for rules in the Earley algorithm

Let's look in more detail at the Scanner Rule. Suppose the chart contains an incomplete edge with a lexical category *P* immediately after the dot, the next word in the input is *w*, *P* is a part-of-speech label for *w*. Then the Scanner Rule admits a new complete edge in which *P* dominates *w*. More precisely:

(71) **Scanner Rule**

If the chart contains the incomplete edge
  [$A \rightarrow \alpha \bullet P \beta$, ($i$, j)]
and w_j is the j^{th} word of the input,
and P is a valid part of speech for w_j,
then add the new complete edges
  [$P \rightarrow w_j \bullet$, ($j$, j+1)]
  [$w_j \rightarrow \bullet$, (j, j+1)]

To illustrate, suppose the input is of the form *I saw* ..., and the chart already contains the edge [VP → • V ..., (1, 1)]. Then the Scanner Rule will add to the chart the edges [V -> 'saw', (1, 2)] and ['saw'→ •, (1, 2)]. So in effect the Scanner Rule packages up a sequence of three rule applications: the Bottom-Up Initialization Rule for [$w \rightarrow \bullet$, (j, j+1)], the Top-Down Expand Rule for [$P \rightarrow \bullet\ w_j$, ($j$, j)], and the Fundamental Rule for [$P \rightarrow w_j \bullet$, ($j$, j+1))]. This is considerably more efficient than the Top-Down Strategy, which adds a new edge of the form [$P \rightarrow \bullet\ w$, (j, j)] for *every* lexical rule $P \rightarrow w$, regardless of whether *w* can be found in the input. By contrast with Bottom-Up Initialization, however, the Earley algorithm proceeds strictly left-to-right through the input, applying all applicable rules at that point in the chart, and never backtracking. The NLTK chart parser demo, described above, allows the option of parsing according to the Earley algorithm.

8.3.6 Chart Parsing in NLTK

`nltk_lite.parse.chart` defines a simple yet flexible chart parser, `ChartParse`. A new chart parser is constructed from a grammar and a list of chart rules (also known as a *strategy*). These rules will be applied, in order, until no new edges are added to the chart. In particular, `ChartParse` uses the algorithm shown in (72).

(72)
```
Until no new edges are added:
  For each chart rule R:
    Apply R to any applicable edges in the chart.
    Return any complete parses in the chart.
```

`nltk_lite.parse.chart` defines two ready-made strategies: `TD_STRATEGY`, a basic top-down strategy; and `BU_STRATEGY`, a basic bottom-up strategy. When constructing a chart parser, you can use either of these strategies, or create your own.

The following example illustrates the use of the chart parser. We start by defining a simple grammar, and tokenizing a sentence. We make sure it is a list (not an iterator), since we wish to use the same tokenized sentence several times.

Listing 30

```
from nltk_lite.parse import cfg, ChartParse, BU_STRATEGY
from nltk_lite import tokenize

grammar = cfg.parse_cfg('''
  NP  -> NNS | JJ NNS | NP CC NP
  NNS -> "men" | "women" | "children" | NNS CC NNS
  JJ  -> "old" | "young"
  CC  -> "and" | "or"
  ''')
parser = ChartParse(grammar, BU_STRATEGY)

>>> sent = list(tokenize.whitespace('old men and women'))
>>> for tree in parser.get_parse_list(sent):
...     print tree
(NP: (JJ: 'old') (NNS: (NNS: 'men') (CC: 'and') (NNS: 'women')))
(NP: (NP: (JJ: 'old') (NNS: 'men')) (CC: 'and') (NP: (NNS: 'women')))
```

The `trace` parameter can be specified when creating a parser, to turn on tracing (higher trace levels produce more verbose output). Example 8.3 shows the trace output for parsing a sentence with the bottom-up strategy. Notice that in this output, `'[-----]'` indicates a complete edge, `'>'` indicates a self-loop edge, and `'[----->'` indicates an incomplete edge.

8.3.7 Exercises

1. ☼ Use the graphical chart-parser interface to experiment with different rule invocation strategies. Come up with your own strategy which you can execute manually using the graphical interface. Describe the steps, and report any efficiency improvements it has (e.g. in terms of the size of the resulting chart). Do these improvements depend on the structure of the grammar? What do you think of the prospects for significant performance boosts from cleverer rule invocation strategies?

2. ☼ We have seen that a chart parser adds but never removes edges from a chart. Why?

3. ◑ Write a program to compare the efficiency of a top-down chart parser compared with a recursive descent parser (Section 7.5.1). Use the same grammar and input sentences for both. Compare their performance using the `timeit` module (Section 6.5.4).

Listing 31 Trace of Bottom-Up Parser

```
>>> parser = ChartParse(grammar, BU_STRATEGY, trace=2)
>>> trees = parser.get_parse_list(sent)
|.   old   .   men   .   and   .  women  .|
Bottom Up Init Rule:
|[---------]         .         .         .| [0:1] 'old'
|.         [---------]         .         .| [1:2] 'men'
|.         .         [---------]         .| [2:3] 'and'
|.         .         .         [---------]| [3:4] 'women'
Bottom Up Predict Rule:
|>         .         .         .         .| [0:0] JJ -> * 'old'
|.         >         .         .         .| [1:1] NNS -> * 'men'
|.         .         >         .         .| [2:2] CC -> * 'and'
|.         .         .         >         .| [3:3] NNS -> * 'women'
Fundamental Rule:
|[---------]         .         .         .| [0:1] JJ -> 'old' *
|.         [---------]         .         .| [1:2] NNS -> 'men' *
|.         .         [---------]         .| [2:3] CC -> 'and' *
|.         .         .         [---------]| [3:4] NNS -> 'women' *
Bottom Up Predict Rule:
|>         .         .         .         .| [0:0] NP -> * JJ NNS
|.         >         .         .         .| [1:1] NP -> * NNS
|.         >         .         .         .| [1:1] NNS -> * NNS CC NNS
|.         .         .         >         .| [3:3] NP -> * NNS
|.         .         .         >         .| [3:3] NNS -> * NNS CC NNS
Fundamental Rule:
|[---------]         .         .         .| [0:1] NP -> JJ * NNS
|.         [---------]         .         .| [1:2] NP -> NNS *
|.         [--------->         .         .| [1:2] NNS -> NNS * CC NNS
|[-------------------]         .         .| [0:2] NP -> JJ NNS *
|.         [------------------->         .| [1:3] NNS -> NNS CC * NNS
|.         .         .         [---------]| [3:4] NP -> NNS *
|.         .         .         [--------->| [3:4] NNS -> NNS * CC NNS
|.         [---------------------------]| [1:4] NNS -> NNS CC NNS *
|.         [---------------------------]| [1:4] NP -> NNS *
|.         [--------------------------->| [1:4] NNS -> NNS * CC NNS
|[=========================================]| [0:4] NP -> JJ NNS *
Bottom Up Predict Rule:
|.         >         .         .         .| [1:1] NP -> * NP CC NP
|>         .         .         .         .| [0:0] NP -> * NP CC NP
|.         .         .         >         .| [3:3] NP -> * NP CC NP
Fundamental Rule:
|.         [--------->         .         .| [1:2] NP -> NP * CC NP
|[------------------->         .         .| [0:2] NP -> NP * CC NP
|.         .         .         [--------->| [3:4] NP -> NP * CC NP
|.         [---------------------------->| [1:4] NP -> NP * CC NP
|[========================================>| [0:4] NP -> NP * CC NP
|.         [------------------->         .| [1:3] NP -> NP CC * NP
|[---------------------------->         .| [0:3] NP -> NP CC * NP
|.         [---------------------------]| [1:4] NP -> NP CC NP *
|[=========================================]| [0:4] NP -> NP CC NP *
|.         [---------------------------->| [1:4] NP -> NP * CC NP
|[========================================>| [0:4] NP -> NP * CC NP
```

8.4 Probabilistic Parsing

As we pointed out in the introduction to this chapter, dealing with ambiguity is a key challenge to broad coverage parsers. We have shown how chart parsing can help improve the efficiency of computing multiple parses of the same sentences. But the sheer number of parses can be just overwhelming. We will show how probabilistic parsing helps to manage a large space of parses. However, before we deal with these parsing issues, we must first back up and introduce weighted grammars.

8.4.1 Weighted Grammars

We begin by considering the verb *give*. This verb requires both a direct object (the thing being given) and an indirect object (the recipient). These complements can be given in either order, as illustrated in example (73). In the "prepositional dative" form, the indirect object appears last, and inside a prepositional phrase, while in the "double object" form, the indirect object comes first:

(73a) Kim gave a bone to the dog

(73b) Kim gave the dog a bone

Using the Penn Treebank sample, we can examine all instances of prepositional dative and double object constructions involving *give*, as shown in Listing 8.4.

We can observe a strong tendency for the shortest complement to appear first. However, this does not account for a form like `give NP: federal judges / NP: a raise`, where animacy may be playing a role. In fact there turn out to be a large number of contributing factors, as surveyed by [Bresnan and Hay, 2006].

How can such tendencies be expressed in a conventional context free grammar? It turns out that they cannot. However, we can address the problem by adding weights, or probabilities, to the productions of a grammar.

A **probabilistic context free grammar** (or *PCFG*) is a context free grammar that associates a probability with each of its productions. It generates the same set of parses for a text that the corresponding context free grammar does, and assigns a probability to each parse. The probability of a parse generated by a PCFG is simply the product of the probabilities of the productions used to generate it.

The simplest way to define a PCFG is to load it from a specially formatted string consisting of a sequence of weighted productions, where weights appear in brackets, as shown in Listing 8.5.

It is sometimes convenient to combine multiple productions into a single line, e.g. `VP -> 'saw' NP [0.4] | 'ate' [0.3] | 'gave' NP NP [0.3]`. In order to ensure that the trees generated by the grammar form a probability distribution, PCFG grammars impose the constraint that all productions with a given left-hand side must have probabilities that sum to one. The grammar in Listing 8.5 obeys this constraint: for `S`, there is only one production, with a probability of 1.0; for `VP`, 0.4+0.3+0.3=1.0; and for `NP`, 0.8+0.2=1.0. The parse tree returned by `get_parse()` includes probabilities:

```
>>> from nltk_lite.parse import ViterbiParse
>>> viterbi_parser = ViterbiParse(grammar)
>>> print viterbi_parser.get_parse(['Jack', 'saw', 'the', 'telescope'])
(S: (NP: 'Jack') (VP: 'saw' (NP: 'the' 'telescope'))) (p=0.064)
```

Listing 32

```
from nltk_lite.corpora import treebank
from string import join
def give(t):
    return t.node == 'VP' and len(t) > 2 and t[1].node == 'NP'\
           and (t[2].node == 'PP-DTV' or t[2].node == 'NP')\
           and ('give' in t[0].leaves() or 'gave' in t[0].leaves())
def sent(t):
    return join(token for token in t.leaves() if token[0] not in '*-0')
def print_node(t, width):
        output = "%s %s: %s / %s: %s" %\
            (sent(t[0]), t[1].node, sent(t[1]), t[2].node, sent(t[2]))
        if len(output) > width:
            output = output[:width] + "..."
        print output

>>> for tree in treebank.parsed():
...     for t in tree.subtrees(give):
...         print_node(t, 72)
gave NP: the chefs / NP: a standing ovation
give NP: advertisers / NP: discounts for maintaining or increasing ad sp...
give NP: it / PP-DTV: to the politicians
gave NP: them / NP: similar help
give NP: them / NP:
give NP: only French history questions / PP-DTV: to students in a Europe...
give NP: federal judges / NP: a raise
give NP: consumers / NP: the straight scoop on the U.S. waste crisis
gave NP: Mitsui / NP: access to a high-tech medical product
give NP: Mitsubishi / NP: a window on the U.S. glass industry
give NP: much thought / PP-DTV: to the rates she was receiving , nor to ...
give NP: your Foster Savings Institution / NP: the gift of hope and free...
give NP: market operators / NP: the authority to suspend trading in futu...
gave NP: quick approval / PP-DTV: to $ 3.18 billion in supplemental appr...
give NP: the Transportation Department / NP: up to 50 days to review any...
give NP: the president / NP: such power
give NP: me / NP: the heebie-jeebies
give NP: holders / NP: the right , but not the obligation , to buy a cal...
gave NP: Mr. Thomas / NP: only a `` qualified '' rating , rather than ``...
give NP: the president / NP: line-item veto power
```

Listing 33 Defining a Probabilistic Context Free Grammar (PCFG)

```
from nltk_lite.parse import pcfg
grammar = pcfg.parse_cfg("""
    S  -> NP VP              [1.0]
    VP -> 'saw' NP           [0.4]
    VP -> 'ate'              [0.3]
    VP -> 'gave' NP NP       [0.3]
    NP -> 'the' 'telescope'  [0.8]
    NP -> 'Jack'             [0.2]
    """)

>>> print grammar
Grammar with 6 productions (start state = S)
    S -> NP VP [1.0]
    VP -> 'saw' NP [0.4]
    VP -> 'ate' [0.3]
    VP -> 'gave' NP NP [0.3]
    NP -> 'the' 'telescope' [0.8]
    NP -> 'Jack' [0.2]
```

The next two sections introduce two probabilistic parsing algorithms for PCFGs. The first is an A* parser that uses Viterbi-style dynamic programming to find the single most likely parse for a given text. Whenever it finds multiple possible parses for a subtree, it discards all but the most likely parse. The second is a bottom-up chart parser that maintains a queue of edges, and adds them to the chart one at a time. The ordering of this queue is based on the probabilities associated with the edges, allowing the parser to expand more likely edges before less likely ones. Different queue orderings are used to implement a variety of different search strategies. These algorithms are implemented in the `nltk_lite.parse.viterbi` and `nltk_lite.parse.pchart` modules.

8.4.2 A* Parser

An **A* Parser** is a bottom-up PCFG parser that uses dynamic programming to find the single most likely parse for a text [Klein and Manning, 2003]. It parses texts by iteratively filling in a **most likely constituents table**. This table records the most likely tree for each span and node value. For example, after parsing the sentence "I saw the man with the telescope" with the grammar `pcfg.toy1`, the most likely constituents table contains the following entries (amongst others):

Span	Node	Tree	Prob
[0:1]	NP	(NP: I)	0.15
[6:7]	NP	(NN: telescope)	0.5
[5:7]	NP	(NP: the telescope)	0.2
[4:7]	PP	(PP: with (NP: the telescope))	0.122
[0:4]	S	(S: (NP: I) (VP: saw (NP: the man)))	0.01365
[0:7]	S	(S: (NP: I) (VP: saw (NP: (NP: the man) (PP: with (NP: the telescope)))))	0.0004163250

Span	Node	Tree	Prob

Table 8.14: Fragment of Most Likely Constituents Table

Once the table has been completed, the parser returns the entry for the most likely constituent that spans the entire text, and whose node value is the start symbol. For this example, it would return the entry with a span of [0:6] and a node value of "S".

Note that we only record the *most likely* constituent for any given span and node value. For example, in the table above, there are actually two possible constituents that cover the span [1:6] and have "VP" node values.

1. "saw the man, who has the telescope":

   (VP: saw (NP: (NP: John) (PP: with (NP: the telescope))))

2. "used the telescope to see the man":

   (VP: saw (NP: John) (PP: with (NP: the telescope)))

Since the grammar we are using to parse the text indicates that the first of these tree structures has a higher probability, the parser discards the second one.

Filling in the Most Likely Constituents Table: Because the grammar used by `ViterbiParse` is a PCFG, the probability of each constituent can be calculated from the probabilities of its children. Since a constituent's children can never cover a larger span than the constituent itself, each entry of the most likely constituents table depends only on entries for constituents with *shorter* spans (or equal spans, in the case of unary and epsilon productions).

`ViterbiParse` takes advantage of this fact, and fills in the most likely constituent table incrementally. It starts by filling in the entries for all constituents that span a single element of text. After it has filled in all the table entries for constituents that span one element of text, it fills in the entries for constituents that span two elements of text. It continues filling in the entries for constituents spanning larger and larger portions of the text, until the entire table has been filled.

To find the most likely constituent with a given span and node value, `ViterbiParse` considers all productions that could produce that node value. For each production, it checks the most likely constituents table for sequences of children that collectively cover the span and that have the node values specified by the production's right hand side. If the tree formed by applying the production to the children has a higher probability than the current table entry, then it updates the most likely constituents table with the new tree.

Handling Unary Productions and Epsilon Productions: A minor difficulty is introduced by unary productions and epsilon productions: an entry of the most likely constituents table might depend on another entry with the same span. For example, if the grammar contains the production `V` → `VP`, then the table entries for `VP` depend on the entries for `V` with the same span. This can be a problem if the constituents are checked in the wrong order. For example, if the parser tries to find the most likely constituent for a `VP` spanning [1:3] before it finds the most likely constituents for `V` spanning [1:3], then it can't apply the `V` → `VP` production.

To solve this problem, `ViterbiParse` repeatedly checks each span until it finds no new table entries. Note that cyclic grammar productions (e.g. `V` → `V`) will *not* cause this procedure to enter an

infinite loop. Since all production probabilities are less than or equal to 1, any constituent generated by a cycle in the grammar will have a probability that is less than or equal to the original constituent; so `ViterbiParse` will discard it.

In NLTK, we create Viterbi parsers using `ViterbiParse()`. Note that since `ViterbiParse` only finds the single most likely parse, that `get_parse_list` will never return more than one parse.

Listing 34

```
from nltk_lite.parse import pcfg, ViterbiParse
from nltk_lite import tokenize
grammar = pcfg.parse_cfg('''
  NP  -> NNS [0.5] | JJ NNS [0.3] | NP CC NP [0.2]
  NNS -> "men" [0.1] | "women" [0.2] | "children" [0.3] | NNS CC NNS [0.4]
  JJ  -> "old" [0.4] | "young" [0.6]
  CC  -> "and" [0.9] | "or" [0.1]
  ''')
viterbi_parser = ViterbiParse(grammar)

>>> sent = list(tokenize.whitespace('old men and women'))
>>> print viterbi_parser.parse(sent)
(NP:
  (JJ: 'old')
  (NNS: (NNS: 'men') (CC: 'and') (NNS: 'women'))) (p=0.000864)
```

The `trace` method can be used to set the level of tracing output that is generated when parsing a text. Trace output displays the constituents that are considered, and indicates which ones are added to the most likely constituent table. It also indicates the likelihood for each constituent.

```
>>> viterbi_parser.trace(3)
>>> tree = viterbi_parser.parse(sent)
Inserting tokens into the most likely constituents table...
   Insert: |=...| old
   Insert: |.=..| men
   Insert: |..=.| and
   Insert: |...=| women
Finding the most likely constituents spanning 1 text elements...
   Insert: |=...| JJ -> 'old' [0.4]                0.4000000000
   Insert: |.=..| NNS -> 'men' [0.1]               0.1000000000
   Insert: |.=..| NP -> NNS [0.5]                  0.0500000000
   Insert: |..=.| CC -> 'and' [0.9]                0.9000000000
   Insert: |...=| NNS -> 'women' [0.2]             0.2000000000
   Insert: |...=| NP -> NNS [0.5]                  0.1000000000
Finding the most likely constituents spanning 2 text elements...
   Insert: |==..| NP -> JJ NNS [0.3]               0.0120000000
Finding the most likely constituents spanning 3 text elements...
   Insert: |.===| NP -> NP CC NP [0.2]             0.0009000000
   Insert: |.===| NNS -> NNS CC NNS [0.4]          0.0072000000
   Insert: |.===| NP -> NNS [0.5]                  0.0036000000
  Discard: |.===| NP -> NP CC NP [0.2]             0.0009000000
  Discard: |.===| NP -> NP CC NP [0.2]             0.0009000000
Finding the most likely constituents spanning 4 text elements...
```

```
   Insert: |====| NP -> JJ NNS [0.3]                 0.0008640000
  Discard: |====| NP -> NP CC NP [0.2]               0.0002160000
  Discard: |====| NP -> NP CC NP [0.2]               0.0002160000
(NP:
  (JJ: 'old')
  (NNS: (NNS: 'men') (CC: 'and') (NNS: 'women'))) (p=0.000864)
```

8.4.3 A Bottom-Up PCFG Chart Parser

The *A* parser* described in the previous section finds the single most likely parse for a given text. However, when parsers are used in the context of a larger NLP system, it is often necessary to produce several alternative parses. In the context of an overall system, a parse that is assigned low probability by the parser might still have the best overall probability.

For example, a probabilistic parser might decide that the most likely parse for "I saw John with the cookie" is is the structure with the interpretation "I used my cookie to see John"; but that parse would be assigned a low probability by a semantic system. Combining the probability estimates from the parser and the semantic system, the parse with the interpretation "I saw John, who had my cookie" would be given a higher overall probability.

This section describes a probabilistic bottom-up chart parser. It maintains an **edge queue**, and adds these edges to the chart one at a time. The ordering of this queue is based on the probabilities associated with the edges, and this allows the parser to insert the most probable edges first. Each time an edge is added to the chart, it may become possible to insert new edges, so these are added to the queue. The bottom-up chart parser continues adding the edges in the queue to the chart until enough complete parses have been found, or until the edge queue is empty.

Like an edge in a regular chart, a probabilistic edge consists of a dotted production, a span, and a (partial) parse tree. However, unlike ordinary charts, this time the tree is weighted with a probability. Its probability is the product of the probability of the production that generated it and the probabilities of its children. For example, the probability of the edge `[Edge: S → NP • VP, 0:2]` is the probability of the PCFG production `S → NP VP` multiplied by the probability of its `NP` child. (Note that an edge's tree only includes children for elements to the left of the edge's dot. Thus, the edge's probability does *not* include probabilities for the constituents to the right of the edge's dot.)

8.4.4 Bottom-Up PCFG Strategies

The *edge queue* is a sorted list of edges that can be added to the chart. It is initialized with a single edge for each token in the text, with the form `[Edge: token |rarr| |dot|]`. As each edge from the queue is added to the chart, it may become possible to add further edges, according to two rules: (i) the Bottom-Up Initialization Rule can be used to add a self-loop edge whenever an edge whose dot is in position 0 is added to the chart; or (ii) the Fundamental Rule can be used to combine a new edge with edges already present in the chart. These additional edges are queued for addition to the chart.

By changing the sort order used by the queue, we can control the strategy that the parser uses to explore the search space. Since there are a wide variety of reasonable search strategies, `BottomUpChartParse()` does not define any sort order. Instead, different strategies are implemented in subclasses of `BottomUpChartParse()`.

Lowest Cost First: The simplest way to order the edge queue is to sort edges by the probabilities of their associated trees (`nltk_lite.parse.InsideParse()`). This ordering concentrates the efforts of the parser on those edges that are more likely to be correct analyses of their underlying tokens.

The probability of an edge's tree provides an upper bound on the probability of any parse produced using that edge. The probabilistic "cost" of using an edge to form a parse is one minus its tree's probability. Thus, inserting the edges with the most likely trees first results in a **lowest-cost-first search strategy**. Lowest-cost-first search is optimal: the first solution it finds is guaranteed to be the best solution.

However, lowest-cost-first search can be rather inefficient. Recall that a tree's probability is the product of the probabilities of all the productions used to generate it. Consequently, smaller trees tend to have higher probabilities than larger ones. Thus, lowest-cost-first search tends to work with edges having small trees before considering edges with larger trees. Yet any complete parse of the text will necessarily have a large tree, and so this strategy will tend to produce complete parses only once most other edges are processed.

Let's consider this problem from another angle. The basic shortcoming with lowest-cost-first search is that it ignores the probability that an edge's tree will be part of a complete parse. The parser will try parses that are locally coherent even if they are unlikely to form part of a complete parse. Unfortunately, it can be quite difficult to calculate the probability that a tree is part of a complete parse. However, we can use a variety of techniques to approximate that probability.

Best-First Search: This method sorts the edge queue in descending order of the edges' span, no the assumption that edges having a larger span are more likely to form part of a complete parse. Thus, `LongestParse` employs a **best-first search strategy**, where it inserts the edges that are closest to producing complete parses before trying any other edges. Best-first search is *not* an optimal search strategy: the first solution it finds is not guaranteed to be the best solution. However, it will usually find a complete parse much more quickly than lowest-cost-first search.

Beam Search: When large grammars are used to parse a text, the edge queue can grow quite long. The edges at the end of a large well-sorted queue are unlikely to be used. Therefore, it is reasonable to remove (or *prune*) these edges from the queue. This strategy is known as **beam search**; it only keeps the best partial results. The bottom-up chart parsers take an optional parameter `beam_size`; whenever the edge queue grows longer than this, it is pruned. This parameter is best used in conjunction with `InsideParse()`. Beam search reduces the space requirements for lowest-cost-first search, by discarding edges that are not likely to be used. But beam search also loses many of lowest-cost-first search's more useful properties. Beam search is not optimal: it is not guaranteed to find the best parse first. In fact, since it might prune a necessary edge, beam search is not even *complete*: it is not guaranteed to return a parse if one exists.

In NLTK we can construct these parsers using `InsideParse`, `LongestParse`, `RandomParse`.

The `trace` method can be used to set the level of tracing output that is generated when parsing a text. Trace output displays edges as they are added to the chart, and shows the probability for each edges' tree.

```
>>> inside_parser.trace(3)
>>> trees = inside_parser.get_parse_list(sent)
  |. . . [-]| [3:4] 'women'                    [1.0]
  |. . [-] .| [2:3] 'and'                      [1.0]
  |. [-] . .| [1:2] 'men'                      [1.0]
  |[-] . . .| [0:1] 'old'                      [1.0]
  |. . [-] .| [2:3] CC -> 'and' *              [0.9]
  |. . > . .| [2:2] CC -> * 'and'              [0.9]
  |[-] . . .| [0:1] JJ -> 'old' *              [0.4]
  |> . . . .| [0:0] JJ -> * 'old'              [0.4]
```

Listing 35

```
from nltk_lite.parse import pchart
inside_parser = pchart.InsideParse(grammar)
longest_parser = pchart.LongestParse(grammar)
beam_parser = pchart.InsideParse(grammar, beam_size=20)

>>> print inside_parser.parse(sent)
(NP:
  (JJ: 'old')
  (NNS: (NNS: 'men') (CC: 'and') (NNS: 'women'))) (p=0.000864)
>>> for tree in inside_parser.get_parse_list(sent):
...     print tree
(NP:
  (JJ: 'old')
  (NNS: (NNS: 'men') (CC: 'and') (NNS: 'women'))) (p=0.000864)
(NP:
  (NP: (JJ: 'old') (NNS: 'men'))
  (CC: 'and')
  (NP: (NNS: 'women'))) (p=0.000216)
```

```
  |> . . . .| [0:0] NP -> * JJ NNS                  [0.3]
  |. . . [-]| [3:4] NNS -> 'women' *                 [0.2]
  |. . . > .| [3:3] NP -> * NNS                      [0.5]
  |. . . > .| [3:3] NNS -> * NNS CC NNS              [0.4]
  |. . . > .| [3:3] NNS -> * 'women'                 [0.2]
  |[-> . . .| [0:1] NP -> JJ * NNS                   [0.12]
  |. . . [-]| [3:4] NP -> NNS *                      [0.1]
  |. . . > .| [3:3] NP -> * NP CC NP                 [0.2]
  |. [-] . .| [1:2] NNS -> 'men' *                   [0.1]
  |. > . . .| [1:1] NP -> * NNS                      [0.5]
  |. > . . .| [1:1] NNS -> * NNS CC NNS              [0.4]
  |. > . . .| [1:1] NNS -> * 'men'                   [0.1]
  |. . . [->| [3:4] NNS -> NNS * CC NNS              [0.08]
  |. [-] . .| [1:2] NP -> NNS *                      [0.05]
  |. > . . .| [1:1] NP -> * NP CC NP                 [0.2]
  |. [-> . .| [1:2] NNS -> NNS * CC NNS              [0.04]
  |. [---> .| [1:3] NNS -> NNS CC * NNS              [0.036]
  |. . . [->| [3:4] NP -> NP * CC NP                 [0.02]
  |[---] . .| [0:2] NP -> JJ NNS *                   [0.012]
  |> . . . .| [0:0] NP -> * NP CC NP                 [0.2]
  |. [-> . .| [1:2] NP -> NP * CC NP                 [0.01]
  |. [---> .| [1:3] NP -> NP CC * NP                 [0.009]
  |. [-----]| [1:4] NNS -> NNS CC NNS *              [0.0072]
  |. [-----]| [1:4] NP -> NNS *                      [0.0036]
  |. [-----># [1:4] NNS -> NNS * CC NNS             [0.00288]
  |[---> . .| [0:2] NP -> NP * CC NP                 [0.0024]
  |[-----> .| [0:3] NP -> NP CC * NP                 [0.00216]
  |. [-----]| [1:4] NP -> NP CC NP *                 [0.0009]
  |[=======]| [0:4] NP -> JJ NNS *                   [0.000864]
```

```
|. [----->| [1:4] NP -> NP * CC NP                    [0.00072]
|[=======]| [0:4] NP -> NP CC NP *                    [0.000216]
|. [----->| [1:4] NP -> NP * CC NP                    [0.00018]
|[------->| [0:4] NP -> NP * CC NP                    [0.0001728]
|[------->| [0:4] NP -> NP * CC NP                    [4.32e-05]
```

8.5 Grammar Induction

As we have seen, PCFG productions are just like CFG productions, adorned with probabilities. So far, we have simply specified these probabilities in the grammar. However, it is more usual to *estimate* these probabilities from training data, namely a collection of parse trees or *treebank*.

The simplest method uses *Maximum Likelihood Estimation*, so called because probabilities are chosen in order to maximize the likelihood of the training data. The probability of a production `VP` → `V NP PP` is *p(V,NP,PP | VP)*. We calculate this as follows:

$$P(V,NP,PP \mid VP) = \frac{count(VP \rightarrow V\ NP\ PP)}{count(VP \rightarrow \ldots)}$$

Here is a simple program that induces a grammar from the first three parse trees in the Penn Treebank corpus:

```
>>> from nltk_lite.corpora import treebank
>>> from itertools import islice
>>> productions = []
>>> S = cfg.Nonterminal('S')
>>> for tree in islice(treebank.parsed(),3):
...     productions += tree.productions()
>>> grammar = pcfg.induce(S, productions)
>>> for production in grammar.productions()[:10]:
...     print production
PP -> IN NP [1.0]
NNP -> 'Nov.' [0.0714285714286]
NNP -> 'Agnew' [0.0714285714286]
JJ -> 'industrial' [0.142857142857]
NP -> CD NNS [0.133333333333]
, -> ',' [1.0]
CC -> 'and' [1.0]
NNP -> 'Pierre' [0.0714285714286]
NP -> NNP NNP NNP NNP [0.0666666666667]
NNP -> 'Rudolph' [0.0714285714286]
```

8.5.1 Normal Forms

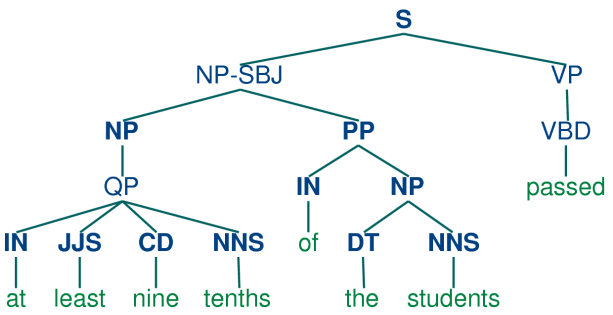
Grammar induction usually involves normalizing the grammar in various ways. The `nltk_lite.parse.treetransforms` module supports binarization (Chomsky Normal Form), parent annotation, Markov order-N smoothing, and unary collapsing. This information can be accessed by importing `treetransforms` from `nltk_lite.parse`, then calling `help(treetransforms)`.

```
>>> from nltk_lite.parse import bracket_parse
>>> from nltk_lite.parse import treetransforms
```

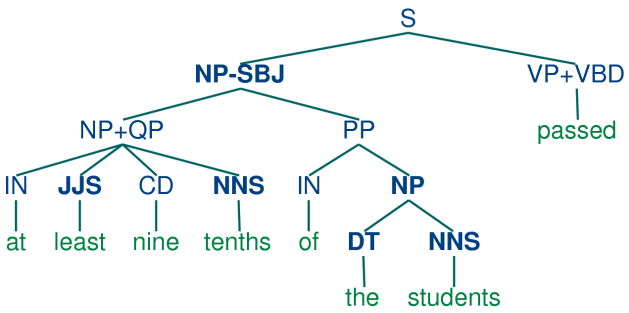
```
>>> treebank_string = """(S (NP-SBJ (NP (QP (IN at) (JJS least) (CD nine) (NNS tent
...     (PP (IN of) (NP (DT the) (NNS students) ))) (VP (VBD passed)))"""
>>> t = bracket_parse(treebank_string)
>>> print t
(S:
  (NP-SBJ:
    (NP: (QP: (IN: 'at') (JJS: 'least') (CD: 'nine') (NNS: 'tenths')))
    (PP: (IN: 'of') (NP: (DT: 'the') (NNS: 'students'))))
  (VP: (VBD: 'passed')))
>>> treetransforms.collapseUnary(t, collapsePOS=True)
>>> print t
(S:
  (NP-SBJ:
    (NP+QP: (IN: 'at') (JJS: 'least') (CD: 'nine') (NNS: 'tenths'))
    (PP: (IN: 'of') (NP: (DT: 'the') (NNS: 'students'))))
  (VP+VBD: 'passed'))
>>> treetransforms.chomskyNormalForm(t)
>>> print t
(S:
  (NP-SBJ:
    (NP+QP:
      (IN: 'at')
      (NP+QP|<JJS-CD-NNS>:
        (JJS: 'least')
        (NP+QP|<CD-NNS>: (CD: 'nine') (NNS: 'tenths'))))
    (PP: (IN: 'of') (NP: (DT: 'the') (NNS: 'students'))))
  (VP+VBD: 'passed'))
```

These trees are shown in (74).

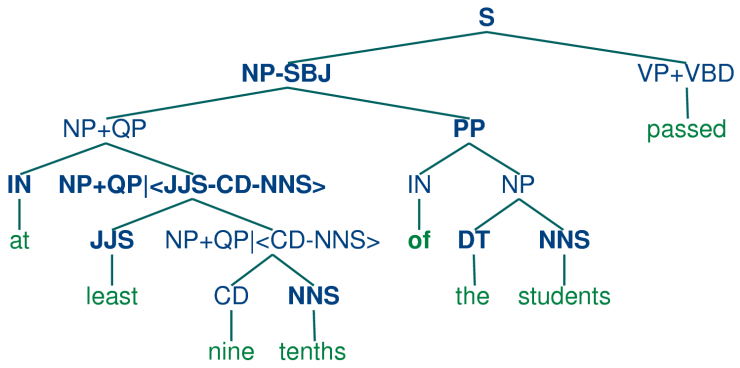
(74a)



(74b)



(74c)



8.6 Conclusion

8.7 Further Reading

- [Manning and Schutze, 1999] (esp chapter 12).
- [Klein and Manning, 2003]

Chapter 9

Feature Based Grammar

9.1 Introduction

The framework of context-free grammars that we presented in Chapter 7 describes syntactic constituents with the help of a limited set of category labels. These atomic labels are adequate for talking about the gross structure of sentences. But when we start to make finer grammatical distinctions it becomes awkward to enrich the set of categories in a systematic manner. In this chapter we will address this challenge by decomposing categories using features (somewhat similar to the key-value pairs of Python dictionaries).

We will start off by looking at the phenomenon of syntactic agreement; we will show how agreement constraints can be expressed elegantly using features, and illustrate how their use in a simple grammar. Feature structures are a general data structure for representing information of any kind; we will briefly look at them from a more formal point of view, and explain how they are made available in NLTK. In the final part of the chapter, we demonstrate that the additional expressiveness of features opens out a wide spectrum of possibilities for describing sophisticated aspects of linguistic structure.

9.2 Decomposing Lingustic Categories

9.2.1 Syntactic Agreement

Consider the following contrasts:

(75a) this dog

(75b) *these dog

(76a) these dogs

(76b) *this dog

In English, nouns are usually morphologically marked as being singular or plural. The form of the demonstrative also varies in a similar way; there is a singular form *this* and a plural form *these*. Examples (75) and (76) show that there are constraints on the realization of demonstratives and nouns within a noun phrase: either both are singular or both are plural. A similar kind of constraint is observed with subjects and predicates:

(77a) the dog runs

(77b) *the dog run

(78a) the dogs run

(78b) *the dogs runs

Here again, we can see that morphological properties of the verb co-vary with morphological properties of the subject noun phrase; this co-variance is usually termed **agreement**. The element which determines the agreement, here the subject noun phrase, is called the agreement **controller**, while the element whose form is determined by agreement, here the verb, is called the **target**. If we look further at verb agreement in English, we will see that present tense verbs typically have two inflected forms: one for third person singular, and another for every other combination of person and number:

	singular	**plural**
1st per	*I run*	*we run*
2nd per	*you run*	*you run*
3rd per	*he/she/it runs*	*they run*

Table 9.1: Agreement Paradigm for English Regular Verbs

We can make the role of morphological properties a bit more explicit as illustrated in (79) and (80). These representations indicate that the verb agrees with its subject in person and number. (We use "3" as an abbreviation for 3rd person, "SG" for singular and "PL" for plural.)

	the	dog	run-s
(79a)		dog.3.SG	run-3.SG

	the	dog-s	run
(80a)		dog.3.PL	run-3.PL

Despite the undoubted interest of agreement as a topic in its own right, we have introduced it here for another reason: we want to look at what happens when we try encode agreement constraints in a context-free grammar. Suppose we take as our starting point the very simple CFG in (81).

(81)
```
S → NP VP
NP → DET N
VP → V

DET → 'this'
N → 'dog'
V → 'runs'
```

(81) allows us to generate the sentence *this dog runs*; however, what we really want to do is also generate *these dogs run* while blocking unwanted strings such as **this dogs run* and **these dog runs*. The most straightforward approach is to add new non-terminals and productions to the grammar which reflect our number distinctions and agreement constraints (we ignore person for the time being):

(82)
```
S_SG → NP_SG VP_SG
S_PL → NP_PL VP_PL
NP_SG → DET_SG N_SG
NP_PL → DET_PL N_PL
VP_SG → V_SG
VP_PL → V_PL

DET_SG → 'this'
DET_PL → 'these'
N_SG → 'dog'
N_PL → 'dogs'
V_SG → 'runs'
V_PL → 'run'
```

It should be clear that this grammar will do the required task, but only at the cost of duplicating our previous set of rules. Rule multiplication is of course more severe if we add in person agreement constraints.

9.2.2 Using Attributes and Constraints

We spoke informally of linguistic categories having *properties*; for example, that a verb has the property of being plural. Let's try to make this more explicit:

(83)
```
N[NUM pl]
```

In (83), we have introduced some new notation which says that the category N has a **feature** called NUM (short for 'number') and that the value of this feature is *pl* (short for 'plural'). We can add similar annotations to other categories, and use them in lexical entries:

(84)
```
DET[NUM sg] → 'this'
DET[NUM pl]  → 'these'
N[NUM sg] → 'dog'
N[NUM pl] → 'dogs'
V[NUM sg] → 'runs'
V[NUM pl] → 'run'
```

Does this help at all? So far, it looks just like a slightly more verbose alternative to what was specified in (82). Things become more interesting when we allow *variables* over feature values, and use these to state constraints. This is illustrated in (85).

(85a)
```
S → NP[NUM ?n] VP[NUM ?n]
```

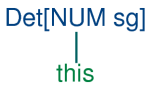
(85b)
```
NP[NUM ?n] → DET[NUM ?n] N[NUM ?n]
```

(85c)
```
VP[NUM ?n] → V[NUM ?n]
```

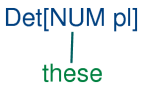
We are using '?*n*' as a variable over values of NUM; it can be instantiated either to *sg* or *pl*. Its scope is limited to individual rules. That is, within (85a), for example, ?*n* must be instantiated to the same constant value; we can read the rule as saying that whatever value NP takes for the feature NUM, VP must take the same value.

In order to understand how these feature constraints work, it's helpful to think about how one would go about building a tree. Lexical rules will admit the following local trees (trees of depth one):

(86a)



(86b)

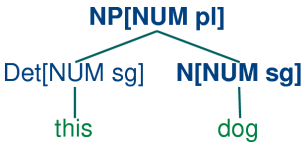


(87a)

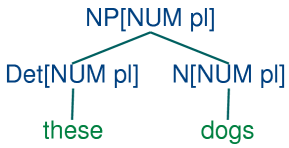
(87b)

Now (85b) says that whatever the NUM values of N and DET are, they have to be the same. Consequently, (85b) will permit (86a) and (87a) to be combined into an NP as shown in (88a) and it will also allow (86b) and (87b) to be combined, as in (88b). By contrast, (89a) and (89b) are prohibited because the roots of their constituent local trees differ in their values for the NUM feature.

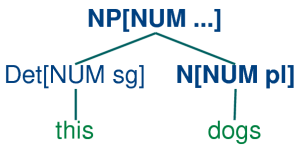
(88a)



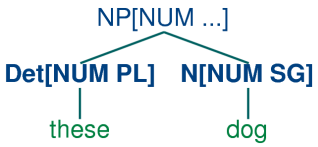
(88b)



(89a)

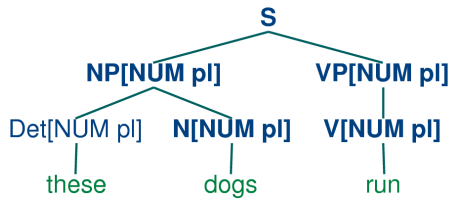


(89b)



Rule (85c) can be thought of as saying that the NUM value of the head verb has to be the same as the NUM value of the VP mother. Combined with (85a), we derive the consequence that if the NUM value of the subject head noun is *pl*, then so is the NUM value of the VP's head verb.

(90)



Grammar (91) illustrates most of the ideas we have introduced so far in this chapter, plus a couple of new ones. As you will see, the format of feature specifications in these productions inserts a `'=' ` between the feature and its value. In our exposition, we will stick to our earlier convention of just leaving space between the feature and value, except when we are directly referring to the NLTK grammar formalism.

(91)

```
% start S
############################
# Grammar Rules
############################

# S expansion rules
S -> NP[NUM=?n] VP[NUM=?n]

# NP expansion rules
NP[NUM=?n] -> N[NUM=?n]
NP[NUM=?n] -> PropN[NUM=?n]
NP[NUM=?n] -> Det[NUM=?n] N[NUM=?n]
NP[NUM=pl] -> N[NUM=pl]

# VP expansion rules
VP[TENSE=?t, NUM=?n] -> IV[TENSE=?t, NUM=?n]
VP[TENSE=?t, NUM=?n] -> TV[TENSE=?t, NUM=?n] NP

############################
# Lexical Rules
############################

Det[NUM=sg] -> 'this' | 'every'
Det[NUM=pl] -> 'these' | 'all'
Det -> 'the' | 'some'

PropN[NUM=sg]-> 'Kim' | 'Jody'

N[NUM=sg] -> 'dog' | 'girl' | 'car' | 'child'
N[NUM=pl] -> 'dogs' | 'girls' | 'cars' | 'children'

IV[TENSE=pres,  NUM=sg] -> 'disappears' | 'walks'
TV[TENSE=pres, NUM=sg] -> 'sees' | 'likes'

IV[TENSE=pres,  NUM=pl] -> 'disappear' | 'walk'
TV[TENSE=pres, NUM=pl] -> 'see' | 'like'
```

```
IV[TENSE=past, NUM=?n] -> 'disappeared' | 'walked'
TV[TENSE=past, NUM=?n] -> 'saw' | 'liked'
```

You will notice that a feature annotation on a syntactic category can contain more than one specification; for example, V[TENSE *pres*, NUM *pl*]. In general, there is no upper bound on the number of features we specify as part of our syntactic categories.

A second point is that we have used feature variables in lexical entries as well as grammatical rules. For example, *the* has been assigned the category DET[NUM ?*n*]. Why is this? Well, you know that the definite article *the* can combine with both singular and plural nouns. One way of describing this would be to add two lexical entries to the grammar, one each for the singular and plural versions of *the*. However, a more elegant solution is to leave the NUM value **underspecified** and letting it agree in number with whatever noun it combines with.

A final detail about (91) is the statement `%start S`. This a "directive" which tells the parser to take S as the start symbol for the grammar.

In general, when we are trying to develop even a very small grammar, it is convenient to put the rules in a file where they can be edited, tested and revised. Assuming we have saved (91) as a file named `'feat0.cfg'`, the function `GrammarFile.read_file()` allows us to read the grammar into NLTK, ready for use in parsing.

```
>>> from nltk_lite.parse import GrammarFile
>>> from pprint import pprint
>>> g = GrammarFile.read_file('feat0.cfg')
```

We can inspect the rules and the lexicon using the commands `print g.earley_grammar()` and `pprint(g.earley_lexicon())`.

Next, we can tokenize a sentence and use the `get_parse_list()` function to invoke the Earley chart parser.

It is important to observe that the parser works directly with the underspecified productions given by the grammar. That is, the Predictor rule does not attempt to compile out all admissible feature combinations before trying to expand the non-terminals on the lefthand side of a production. However, when the Scanner matches an input word against a lexical rule that has been predicted, the new edge will typically contain fully specified features; e.g., the edge [PropN[NUM = *sg*] → 'Kim', (0, 1)]. Recall from Chapter 7 that the Fundamental (or Completer) Rule in standard CFGs is used to combine an incomplete edge that's expecting a nonterminal *B* with a following, complete edge whose left hand side matches *B*. In our current setting, rather than checking for a complete match, we test whether the expected category *B* will **unify** with the lefthand side *B'* of a following complete edge. We will explain in more detail in Section 9.3 how unification works; for the moment, it is enough to know that as a result of unification, any variable values of features in *B* will be instantiated by constant values in the corresponding feature structure in *B'*, and these instantiated values will be used in the new edge added by the Completer. This instantiation can be seen, for example, in the edge [NP[NUM *sg*] → PropN[NUM *sg*] •, (0, 1)] in 9.1, where the feature NUM has been assigned the value *sg*.

Finally, we can inspect the resulting parse trees (in this case, a single one).

```
>>> for tree in trees: print tree
...
([INIT]:
  (Start:
    (S:
      (NP[NUM=sg]: (PropN[NUM=sg]: 'Kim'))
```

Listing 36 Trace of Feature-Based Chart Parser

```
>>> from nltk_lite import tokenize
>>> sent = 'Kim likes children'
>>> tokens = list(tokenize.whitespace(sent))
>>> tokens
['Kim', 'likes', 'children']
>>> cp = g.earley_parser(trace=10)
>>> trees = cp.get_parse_list(tokens)
          |.K.l.c.|

Predictor |> . . .| S -> * NP[NUM=?n] VP[NUM=?n]
Predictor |> . . .| NP[NUM=?n] -> * N[NUM=?n]
Predictor |> . . .| NP[NUM=?n] -> * PropN[NUM=?n]
Predictor |> . . .| NP[NUM=?n] -> * Det[NUM=?n] N[NUM=?n]
Predictor |> . . .| NP[NUM=pl] -> * N[NUM=pl]
Scanner   |[-] . .| [0:1] 'Kim'
Completer |[-] . .| NP[NUM=sg] -> PropN[NUM=sg] *
Completer |[-> . .| S -> NP[NUM=sg] * VP[NUM=sg]
Predictor |. > . .| VP[NUM=?n, TENSE=?t] -> * IV[NUM=?n, TENSE=?t]
Predictor |. > . .| VP[NUM=?n, TENSE=?t] -> * TV[NUM=?n, TENSE=?t] NP
Scanner   |. [-] .| [1:2] 'likes'
Completer |. [-> .| VP[NUM=sg, TENSE=pres] -> TV[NUM=sg, TENSE=pres] * NP
Predictor |. . > .| NP[NUM=?n] -> * N[NUM=?n]
Predictor |. . > .| NP[NUM=?n] -> * PropN[NUM=?n]
Predictor |. . > .| NP[NUM=?n] -> * Det[NUM=?n] N[NUM=?n]
Predictor |. . > .| NP[NUM=pl] -> * N[NUM=pl]
Scanner   |. . [-]| [2:3] 'children'
Completer |. . [-]| NP[NUM=pl] -> N[NUM=pl] *
Completer |. [---]| VP[NUM=sg, TENSE=pres] -> TV[NUM=sg, TENSE=pres] NP *
Completer |[=====]| S -> NP[NUM=sg] VP[NUM=sg] *
Completer |[=====]| [INIT] -> S *
```

```
(VP[NUM=sg, TENSE=pres]:
  (TV[NUM=sg, TENSE=pres]: 'likes')
  (NP[NUM=pl]: (N[NUM=pl]: 'children'))))))
```

9.2.3 Terminology

So far, we have only seen feature values like *sg* and *pl*. These simple values are usually called **atomic** — that is, they can't be decomposed into subparts. A special case of atomic values are **boolean** values, that is, values which just specify whether a property is true or false of a category. For example, we might want to distinguish **auxiliary** verbs such as *can*, *may*, *will* and *do* with the boolean feature AUX. Then our lexicon for verbs could include entries such as the following:

(92)
```
V[TENSE pres, AUX +] → 'can'
V[TENSE pres, AUX +] → 'may'

V[TENSE pres, AUX –] → 'walks'
V[TENSE pres, AUX –] → 'likes'
```

A frequently used abbreviation for boolean features allows the value to be prepended to the feature:

(93)
```
V[TENSE pres, +AUX] → 'can'
V[TENSE pres, –AUX] → 'walks'
```

We have spoken informally of attaching 'feature annotations' to syntactic categories. A more general approach is to treat the whole category — that is, the non-terminal symbol plus the annotation — as a bundle of features. Consider, for example, the object we have written as (94).

(94)
```
N[NUM sg]
```

The syntactic category N, as we have seen before, provides part of speech information. This information can itself be captured as a feature value pair, using POS to represent "part of speech":

(95)
```
[POS N, NUM sg]
```

In fact, we regard (95) as our "official" representation of a feature-based linguistic category, and (94) as a convenient abbreviation. A bundle of feature-value pairs is called a **feature structure** or an **attribute value matrix** (AVM). A feature structure which contains a specification for the feature POS is a **linguistic category**.

In addition to atomic-valued features, we allow features whose values are themselves feature structures. For example, we might want to group together agreement features (e.g., person, number and gender) as a distinguished part of a category, as shown in (96).

(96)
$$\begin{bmatrix} \text{POS} & N \\ \text{AGR} & \begin{bmatrix} \text{PER} & 3 \\ \text{NUM} & pl \\ \text{GND} & fem \end{bmatrix} \end{bmatrix}$$

In this case, we say that the feature AGR has a **complex** value.

There is no particular significance to the *order* of features in a feature structure. So (96) is equivalent to (96).

(97)
$$\begin{bmatrix} & \begin{bmatrix} \text{NUM} & pl \\ \text{PER} & 3 \\ \text{GND} & fem \end{bmatrix} \\ \text{AGR} & \\ \text{POS} & N \end{bmatrix}$$

Once we have the possibility of using features like AGR, we can refactor a grammar like (91) so that agreement features are bundled together. A tiny grammar illustrating this point is shown in (98).

(98)
```
S → NP[AGR ?n] VP[AGR ?n]
NP[AGR ?n] → PropN[AGR ?n]
VP[TENSE ?t, AGR ?n] → Cop[TENSE ?t, AGR ?n] Adj
Cop[TENSE pres, AGR [NUM sg, PER 3]] → 'is'
PropN[AGR [NUM sg, PER 3]] → 'Kim'
Adj → 'happy'
```

9.2.4 Exercises

1. ☼ What constraints are required to correctly parse strings like *I am happy* and *she is happy* but not **you is happy* or **they am happy*? Implement two solutions for the present tense paradigm of the verb *be* in English, first taking Grammar (82) as your starting point, and then taking Grammar (98) as the starting point.

2. ☼ Develop a variant of grammar (91) which uses a COUNT to make the distinctions shown below:

(99a) The boy sings.

(99b) *Boy sings.

(100a) The boys sing.

(100b) Boys sing.

(101a) The boys sing.

(101b) Boys sing.

(102a) The water is precious.

(102b) Water is precious.

3. ◑ Develop a feature-based grammar that will correctly describe the following Spanish noun phrases:

(103a)

un	cuadro	hermos-o
INDEF.SG.MASC	picture	beautiful-SG.MASC

'a beautiful picture'

(103b)	un-os INDEF-PL.MASC 'beautiful pictures'	cuadro-s picture-PL	hermos-os beautiful-PL.MASC
(103c)	un-a INDEF.SG.FEM 'a beautiful curtain'	cortina curtain	hermos-a beautiful-SG.FEM
(103d)	un-as INDEF.PL.FEM 'beautiful curtains'	cortina-s curtain	hermos-as beautiful-PL.FEM

4. ◑ Develop a wrapper for the `earley_parser` so that a trace is only printed if the input string fails to parse.

9.3 Computing with Feature Structures

In this section, we will show how feature structures can be constructed and manipulated in NLTK. We will also discuss the fundamental operation of unification, which allows us to combine the information contained in two different feature structures.

9.3.1 Feature Structures in NLTK

Feature structures in NLTK are declared with the `FeatureStructure()` constructor. Atomic feature values can be strings or integers.

```
>>> from nltk_lite.featurestructure import *
>>> fs1 = FeatureStructure(TENSE='past', NUM='sg')
>>> print fs1
[ NUM   = 'sg'   ]
[ TENSE = 'past' ]
```

We can think of a feature structure as being like a Python dictionary, and access its values by indexing in the usual way.

```
>>> fs1 = FeatureStructure(PER=3, NUM='pl', GND='fem')
>>> print fs1['GND']
fem
```

However, we cannot use this syntax to *assign* values to features:

```
>>> fs1['CASE'] = 'acc'
Traceback (most recent call last):
...
KeyError: 'CASE'
```

We can also define feature structures which have complex values, as discussed earlier.

```
>>> fs2 = FeatureStructure(POS='N', AGR=fs1)
>>> print fs2
[       [ GND = 'fem' ] ]
[ AGR = [ NUM  = 'pl'  ] ]
[       [ PER  = 3     ] ]
[                        ]
[ POS = 'N'              ]
>>> print fs2['AGR']
[ GND = 'fem' ]
[ NUM  = 'pl'  ]
[ PER  = 3     ]
>>> print fs2['AGR']['PER']
3
```

An alternative method of specifying feature structures in NLTK is to use the `parse` method of `FeatureStructure`. This gives us the facility to use square bracket notation for embedding one feature structure within another.

```
>>> FeatureStructure.parse("[POS='N', AGR=[PER=3, NUM='pl', GND='fem']]")
[AGR=[GND='fem', NUM='pl', PER=3], POS='N']
```

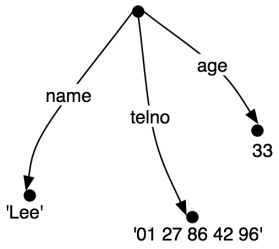
9.3.2 Feature Structures as Graphs

Feature structures are not inherently tied to linguistic objects; they are general purpose structures for representing knowledge. For example, we could encode information about a person in a feature structure:

```
>>> person01 = FeatureStructure(name='Lee', telno='01 27 86 42 96', age=33)
```

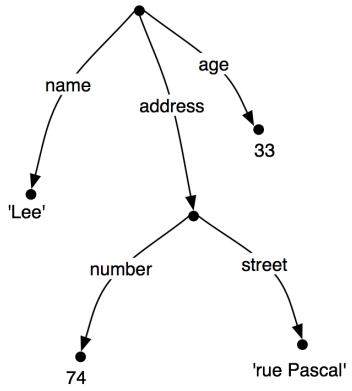
(104)
$$\begin{bmatrix} \text{NAME} & \textit{'Lee'} \\ \text{TELNO} & \textit{01 27 86 42 96} \\ \text{AGE} & \textit{33} \end{bmatrix}$$

It is sometimes helpful to view feature structures as graphs; more specifically, **directed acyclic graphs** (DAGs). (105) is equivalent to the AVM (104).

(105)


The feature names appear as labels on the directed arcs, and feature values appear as labels on the nodes which are pointed to by the arcs.

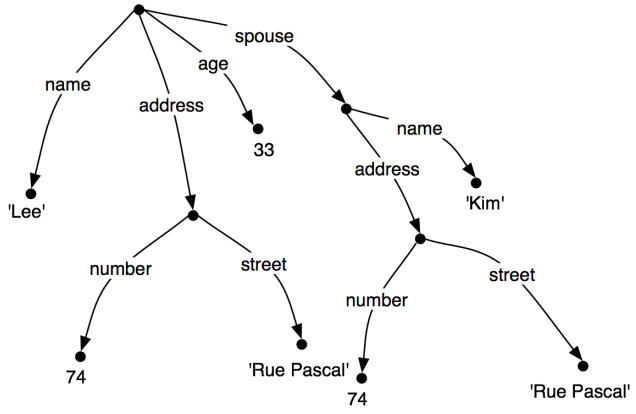
Just as before, feature values can be complex:



(106)

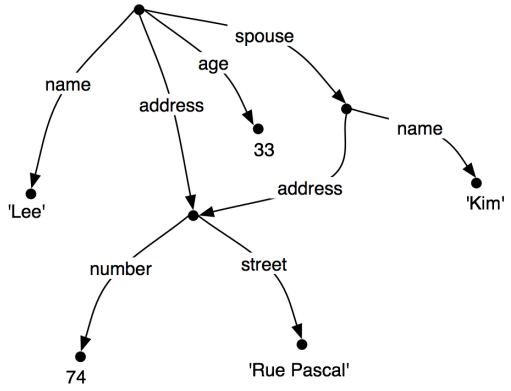
When we look at such graphs, it is natural to think in terms of paths through the graph. A **feature path** is a sequence of arcs that can be followed from the root node. We will represent paths in NLTK as tuples. Thus, `('address', 'street')` is a feature path whose value in (106) is the string 'rue Pascal'.

Now let's consider a situation where Lee has a spouse named "Kim", and Kim's address is the same as Lee's. We might represent this as (107).



(107)

However, rather than repeating the address information in the feature structure, we can "share" the same sub-graph between different arcs:



(108)

In other words, the value of the path `('ADDRESS')` in (108) is identical to the value of the path `('SPOUSE', 'ADDRESS')`. DAGs such as (108) are said to involve **structure sharing** or **reentrancy**. When two paths have the same value, they are said to be **equivalent**.

There are a number of notations for representing reentrancy in matrix-style representations of feature structures. In NLTK, we adopt the following convention: the first occurrence of a shared feature structure is prefixed with an integer in parentheses, such as `(1)`, and any subsequent reference to that structure uses the notation `->(1)`, as shown below.

```
>>> fs=FeatureStructure.parse("""[NAME='Lee', ADDRESS=(1)[NUMBER=74, STREET='rue Pascal'],
...                               SPOUSE=[NAME='Kim', ADDRESS->(1)]]""")
>>> print fs
[ ADDRESS = (1) [ NUMBER = 74           ] ]
[               [ STREET = 'rue Pascal' ] ]
[                                         ]
[ NAME    = 'Lee'                         ]
[                                         ]
[ SPOUSE  = [ ADDRESS -> (1)  ]           ]
[           [ NAME    = 'Kim' ]           ]
```

This is similar to more conventional displays of AVMs, as shown in (109).

(109)
$$\begin{bmatrix} \text{ADDRESS} & \boxed{1}\begin{bmatrix} \text{NUMBER} & 74 \\ \text{STREET} & \text{'rue Pascal'} \end{bmatrix} \\ \text{NAME} & \text{'Lee'} \\ \text{SPOUSE} & \begin{bmatrix} \text{ADDRESS} & \boxed{1} \\ \text{NAME} & \text{'Kim'} \end{bmatrix} \end{bmatrix}$$

The bracketed integer is sometimes called a **tag** or a **coindex**. The choice of integer is not significant. There can be any number of tags within a single feature structure.

```
>>> fs1 = FeatureStructure.parse("[A='a', B=(1)[C='c'], D->(1), E->(1)]")
```

(110)
$$\begin{bmatrix} \text{A} & \text{'a'} \\ \text{B} & \boxed{1}\begin{bmatrix} \text{C} & \text{'c'} \end{bmatrix} \\ \text{D} & \boxed{1} \\ \text{E} & \boxed{1} \end{bmatrix}$$

9.3.3 Subsumption and Unification

It is standard to think of feature structures as providing **partial information** about some object, in the sense that we can order feature structures according to how general they are. For example, (111a) is more general (less specific) than (111b), which in turn is more general than (111c).

(111a)
$$\begin{bmatrix} \text{NUMBER} & 74 \end{bmatrix}$$

(111b)
$$\begin{bmatrix} \text{NUMBER} & 74 \\ \text{STREET} & \text{'rue Pascal'} \end{bmatrix}$$

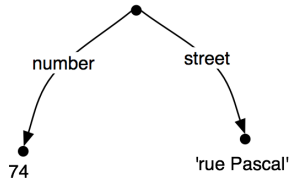
(111c)
$$\begin{bmatrix} \text{NUMBER} & 74 \\ \text{STREET} & \text{'rue Pascal'} \\ \text{CITY} & \text{'Paris'} \end{bmatrix}$$

This ordering is called **subsumption**; a more general feature structure **subsumes** a less general one. If FS_0 subsumes FS_1 (formally, we write FS_0 FS_1), then FS_1 must have all the paths and path equivalences of FS_0, and may have additional paths and equivalences as well. Thus, (107) subsumes (108), since the latter has additional path equivalences.. It should be obvious that subsumption only provides a partial ordering on feature structures, since some feature structures are incommensurable. For example, (112) neither subsumes nor is subsumed by (111a).

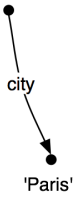
(112) $\left[\text{TELNO} \quad 01\ 27\ 86\ 42\ 96\right]$

So we have seen that some feature structures are more specific than others. How do we go about specialising a given feature structure? For example, we might decide that addresses should consist of not just a street number and a street name, but also a city. That is, we might want to *merge* graph (113b) with (113a) to yield (113c).

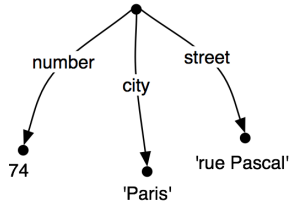
(113a)



(113b)



(113c)



Merging information from two feature structures is called **unification** and in NLTK is supported by the `unify()` method defined in the `FeatureStructure` class.

```
>>> fs1 = FeatureStructure(NUMBER=74, STREET='rue Pascal')
>>> fs2 = FeatureStructure(CITY='Paris')
>>> print fs1.unify(fs2)
[ CITY   = 'Paris'      ]
[ NUMBER = 74           ]
[ STREET = 'rue Pascal' ]
```

Unification is formally defined as a binary operation: FS_0 FS_1. Unification is symmetric, so

(114) FS_0 $FS_1 = FS_1$ FS_0.

The same is true in NLTK:

```
>>> print fs2.unify(fs1)
[ CITY   = 'Paris'      ]
[ NUMBER = 74           ]
[ STREET = 'rue Pascal' ]
```

If we unify two feature structures which stand in the subsumption relationship, then the result of unification is the most specific of the two:

(115) If $FS_0 \quad FS_1$, then $FS_0 \quad FS_1 = FS_1$

For example, the result of unifying (111b) with (111c) is (111c).

Unification between FS_0 and FS_1 will fail if the two feature structures share a path π, but the value of π in FS_0 is a distinct atom from the value of π in FS_1. In NLTK, this is implemented by setting the result of unification to be `None`.

```
>>> fs0 = FeatureStructure(A='a')
>>> fs1 = FeatureStructure(A='b')
>>> fs2 = fs0.unify(fs1)
>>> print fs2
None
```

Now, if we look at how unification interacts with structure-sharing, things become really interesting. First, let's define the NLTK version of (107).

```
>>> fs0=FeatureStructure.parse("""[NAME=Lee,
...                                ADDRESS=[NUMBER=74,
...                                         STREET='rue Pascal'],
...                                SPOUSE= [NAME=Kim,
...                                         ADDRESS=[NUMBER=74,
...                                                  STREET='rue Pascal']]]""")
```

(116)
$$\begin{bmatrix} \text{ADDRESS} & \begin{bmatrix} \text{NUMBER} & 74 \\ \text{STREET} & \text{'rue Pascal'} \end{bmatrix} \\ \text{NAME} & \text{'Lee'} \\ \text{SPOUSE} & \begin{bmatrix} \text{ADDRESS} & \begin{bmatrix} \text{NUMBER} & 74 \\ \text{STREET} & \text{'rue Pascal'} \end{bmatrix} \\ \text{NAME} & \text{'Kim'} \end{bmatrix} \end{bmatrix}$$

What happens when we augment Kim's address with a specification for CITY? (Notice that `fs1` includes the whole path from the root of the feature structure down to CITY.)

```
>>> fs1=FeatureStructure.parse("[SPOUSE = [ADDRESS = [CITY = Paris]]]")
```

(117) shows the result of unifying `fs0` with `fs1`:

(117)
$$\begin{bmatrix} \text{ADDRESS} & \begin{bmatrix} \text{NUMBER} & 74 \\ \text{STREET} & \text{'rue Pascal'} \end{bmatrix} \\ \text{NAME} & \text{'Lee'} \\ \text{SPOUSE} & \begin{bmatrix} \text{ADDRESS} & \begin{bmatrix} \text{CITY} & \text{'Paris'} \\ \text{NUMBER} & 74 \\ \text{STREET} & \text{'rue Pascal'} \end{bmatrix} \\ \text{NAME} & \text{'Kim'} \end{bmatrix} \end{bmatrix}$$

By contrast, the result is very different if `fs1` is unified with the structure-sharing version `fs2` (also shown as (108)):

```
>>> fs2=FeatureStructure.parse("""[NAME=Lee, ADDRESS=(1)[NUMBER=74, STREET='rue Pas
...                                 SPOUSE=[NAME=Kim, ADDRESS->(1)]]""")
```

(118)
$$\begin{bmatrix} & & \text{CITY} & \textit{'Paris'} \\ \text{ADDRESS} & \boxed{1} & \text{NUMBER} & 74 \\ & & \text{STREET} & \textit{'rue Pascal'} \\ \text{NAME} & & \textit{'Lee'} & \\ \text{SPOUSE} & & \begin{bmatrix} \text{ADDRESS} & \boxed{1} \\ \text{NAME} & \textit{'Kim'} \end{bmatrix} & \end{bmatrix}$$

Rather than just updating what was in effect Kim's "copy" of Lee's address, we have now updated *both* their addresses at the same time. More generally, if a unification involves specialising the value of some path π, then that unification simultaneously specialises the value of *any path that is equivalent to* π.

As we have already seen, structure sharing can also be stated in NLTK using variables such as `?x`.

```
>>> fs1=FeatureStructure.parse("[ADDRESS1=[NUMBER=74, STREET='rue Pascal']]")
>>> fs2=FeatureStructure.parse("[ADDRESS1=?x, ADDRESS2=?x]")
>>> print fs2
[ ADDRESS1 = ?x ]
[ ADDRESS2 = ?x ]
>>> print fs2.unify(fs1)
[ ADDRESS1 = (1) [ NUMBER = 74           ] ]
[                [ STREET = 'rue Pascal' ] ]
[                                          ]
[ ADDRESS2 -> (1)                          ]
```

9.3.4 Exercises

1. ☼ Write a function *subsumes()* which holds of two feature structures `fs1` and `fs2` just in case `fs1` subsumes `fs2`.

2. ◑ Consider the feature structures shown in Listing 9.2.

Listing 37

```
fs1 = FeatureStructure.parse("[A = (1)b, B= [C ->(1)]]")
fs2 = FeatureStructure.parse("[B = [D = d]]")
fs3 = FeatureStructure.parse("[B = [C = d]]")
fs4 = FeatureStructure.parse("[A = (1)[B = b], C->(1)]")
fs5 = FeatureStructure.parse("[A = [D = (1)e], C = [E -> (1)] ]")
fs6 = FeatureStructure.parse("[A = [D = (1)e], C = [B -> (1)] ]")
fs7 = FeatureStructure.parse("[A = [D = (1)e, F = (2)[]], C = [B -> (1), E -> (2)]
fs8 = FeatureStructure.parse("[A = [B = b], C = [E = [G = e]]]")
fs9 = FeatureStructure.parse("[A = (1)[B = b], C -> (1)]")
```

Work out on paper what the result is of the following unifications. (Hint: you might find it useful to draw the graph structures.)

- `fs1` and `fs2`
- `fs1` and `fs3`
- `fs4` and `fs5`
- `fs5` and `fs6`
- `fs7` and `fs8`
- `fs7` and `fs9`

Check your answers on the computer.

3. ◑ List two feature structures which subsume [A=?x, B=?x].

4. ◑ Ignoring structure sharing, give an informal algorithm for unifying two feature structures.

9.4 Extending a Feature-Based Grammar

9.4.1 Subcategorization

In Chapter 7, we proposed to augment our category labels in order to represent different subcategories of verb. More specifically, we introduced labels such as IV and TV for intransitive and transitive verbs respectively. This allowed us to write rules like the following:

(119)
```
VP → IV
VP → TV NP
```

Although it is tempting to think of IV and TV as two kinds of V, this is unjustified: from a formal point of view, IV has no closer relationship with TV than it does, say, with NP. As it stands, IV and TV are unanalyzable nonterminal symbols from a CFG. One unwelcome consequence is that we do not seem able to say anything about the class of verbs in general. For example, we cannot say something like "All lexical items of category V can be marked for tense", since *bark*, say, is an item of category IV, not V.

Using features gives us some useful room for manoeuvre but there is no obvious consensus on how to model subcategorization information. One approach which has the merit of simplicity is due to Generalized Phrase Structure Grammar (GPSG). GPSG stipulates that lexical categories may bear a SUBCAT whose values are integers. This is illustrated in a modified portion of (91), shown in (120).

(120)
```
VP[TENSE=?t, NUM=?n] -> V[SUBCAT=0, TENSE=?t, NUM=?n]
VP[TENSE=?t, NUM=?n] -> V[SUBCAT=1, TENSE=?t, NUM=?n] NP
VP[TENSE=?t, NUM=?n] -> V[SUBCAT=2, TENSE=?t, NUM=?n] Sbar

V[SUBCAT=0, TENSE=pres, NUM=sg] -> 'disappears' | 'walks'
V[SUBCAT=1, TENSE=pres, NUM=sg] -> 'sees' | 'likes'
V[SUBCAT=2, TENSE=pres, NUM=sg] -> 'says' | 'claims'

V[SUBCAT=0, TENSE=pres, NUM=pl] -> 'disappear' | 'walk'
V[SUBCAT=1, TENSE=pres, NUM=pl] -> 'see' | 'like'
```

```
V[SUBCAT=2, TENSE=pres, NUM=pl] -> 'say' | 'claim'

V[SUBCAT=0, TENSE=past, NUM=?n] -> 'disappeared' | 'walked'
V[SUBCAT=1, TENSE=past, NUM=?n] -> 'saw' | 'liked'
V[SUBCAT=2, TENSE=past, NUM=?n] -> 'said' | 'claimed'
```

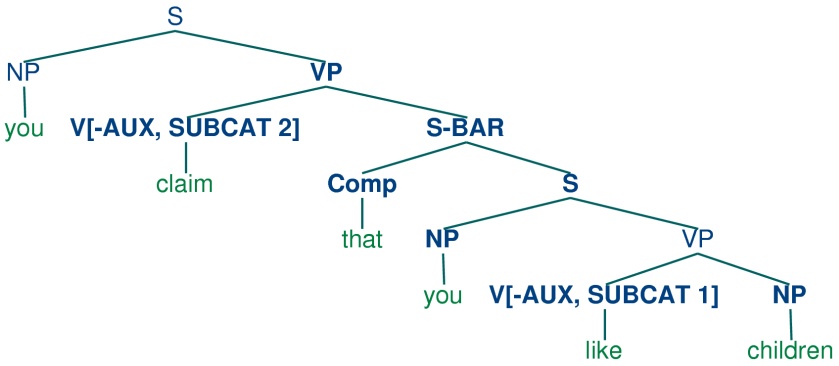
When we see a lexical category like V[SUBCAT *1*], we can interpret the SUBCAT specification as a pointer to the rule in which V[SUBCAT *1*] is introduced as the head daughter in a VP expansion rule. By convention, there is a one-to-one correspondence between SUBCAT values and rules which introduce lexical heads. It's worth noting that the choice of integer which acts as a value for SUBCAT is completely arbitrary — we could equally well have chosen 3999, 113 and 57 as our two values in (120). On this approach, SUBCAT can *only* appear on lexical categories; it makes no sense, for example, to specify a SUBCAT value on VP.

In our third class of verbs above, we have specified a category S-BAR. This is a label for subordinate clauses such as the complement of *claim* in the example *You claim that you like children*. We require two further rules to analyse such sentences:

(121)
```
S-BAR -> Comp S
Comp -> 'that'
```

The resulting structure is the following.

(122)



An alternative treatment of subcategorization, due originally to a framework known as categorial grammar, is represented in feature-based frameworks such as PATR and Head-driven Phrase Structure Grammar. Rather than using SUBCAT values as a way of indexing rules, the SUBCAT value directly encodes the valency of a head (the list of arguments that it can combine with). For example, a verb like *put* which takes NP and PP complements (*put the book on the table*) might be represented as (123):

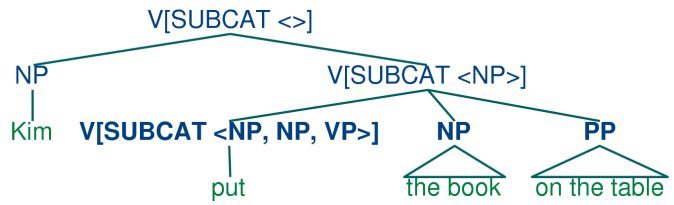
(123) V[SUBCAT NP, NP, PP]

This says that the verb can combine with three arguments. The leftmost element in the list is the subject NP, while everything else — an NP followed by a PP in this case — comprises the subcategorized-for complements. When a verb like *put* is combined with appropriate complements, the requirements which are specified in the SUBCAT are discharged, and only a subject NP is needed. This category, which corresponds to what is traditionally thought of as VP, might be represented as follows.

(124) V[SUBCAT NP]

Finally, a sentence is a kind of verbal category which has *no* requirements for further arguments, and hence has a SUBCAT whose value is the empty list. The tree (125) shows how these category assigments combine in a parse of *Kim put the book on the table*.

(125)

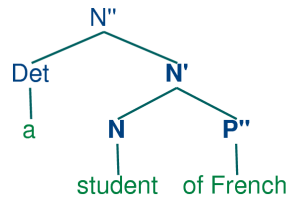


9.4.2 Heads Revisited

We noted in the previous section that by factoring subcategorization information out of the main category label, we could express more generalizations about properties of verbs. Another property of this kind is the following: expressions of category V are heads of phrases of category VP. Similarly (and more informally) Ns are heads of NPs, As (i.e., adjectives) are heads of APs, and Ps (i.e., adjectives) are heads of PPs. Not all phrases have heads — for example, it is standard to say that coordinate phrases (e.g., *the book and the bell*) lack heads — nevertheless, we would like our grammar formalism to express the mother / head-daughter relation where it holds. Now, although it looks as though there is something in common between, say, V and VP, this is more of a handy convention than a real claim, since V and VP formally have no more in common than V and DET.

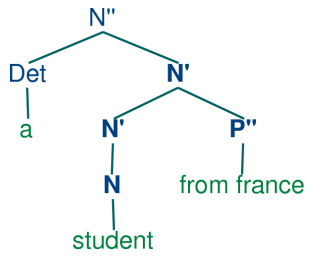
X-bar syntax (cf. [Chomsky, 1970], [Jackendoff, 1977]) addresses this issue by abstracting out the notion of **phrasal level**. It is usual to recognise three such levels. If N represents the lexical level, then N' represents the next level up, corresponding to the more traditional category NOM, while N" represents the phrasal level, corresponding to the category NP. (The primes here replace the typographically more demanding horizontal bars of [Chomsky, 1970]). (126) illustrates a representative structure.

(126)



The head of the structure (126) is N while N' and N" are called **(phrasal) projections** of N. N" is the **maximal projection**, and N is sometimes called the **zero projection**. One of the central claims of X-bar syntax is that all constituents share a structural similarity. Using X as a variable over N, V, A and P, we say that directly subcategorized *complements* of the head are always placed as sisters of the lexical head, whereas *adjuncts* are placed as sisters of the intermediate category, X'. Thus, the configuration of the P" adjunct in (127) contrasts with that of the complement P" in (126).

(127)



The productions in (128) illustrate how bar levels can be encoded using feature structures.

(128)
```
S → N[BAR 2] V[BAR 2]
N[BAR 2] → DET N[BAR 1]
N[BAR 1] → N[BAR 1] P[BAR 2]
N[BAR 1] → N[BAR 0] P[BAR 2]
```

9.4.3 Auxiliary verbs and Inversion

Inverted clauses — where the order of subject and verb is switched — occur in English interrogatives and also after 'negative' adverbs:

(129a) Do you like children?

(129b) Can Jody walk?

(130a) Rarely do you see Kim.

(130b) Never have I seen this dog.

However, we cannot place just any verb in pre-subject position:

(131a) *Like you children?

(131b) *Walks Jody?

(132a) *Rarely see you Kim.

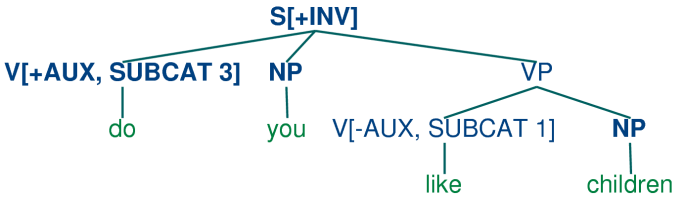
(132b) *Never saw I this dog.

Verbs which can be positioned inititally in inverted clauses belong to the class known as **auxiliaries**, and as well as *do*, *can* and *have* include *be*, *will* and *shall*. One way of capturing such structures is with the following rule:

(133)
```
S[+inv] -> V[+AUX] NP VP
```

That is, a clause marked as [+INV] consists of an auxiliary verb followed by a VP. (In a more detailed grammar, we would need to place some constraints on the form of the VP, depending on the choice of auxiliary.) (134) illustrates the structure of an inverted clause.

(134)



9.4.4 Unbounded Dependency Constructions

Consider the following contrasts:

(135a) You like Jody.

(135b) *You like.

(136a) You put the card into the slot.

(136b) *You put into the slot.

(136c) *You put the card.

(136d) *You put.

The verb *like* requires an NP complement, while *put* requires both a following NP and PP. Examples (135) and (136) show that these complements are *obligatory*: omitting them leads to ungrammaticality. Yet there are contexts in which obligatory complements can be omitted, as (137) and (138) illustrate.

(137a) Kim knows who you like.

(137b) This music, you really like.

(138a) Which card do you put into the slot?

(138b) Which slot do you put the card into?

That is, an obligatory complement can be omitted if there is an appropriate **filler** in the sentence, such as the question word *who* in (137a), the preposed topic *this music* in (137b), or the *wh* phrases *which card/slot* in (138). It is common to say that sentences like (137) – (138) contain **gaps** where the obligatory complements have been omitted, and these gaps are sometimes made explicit using an underscore:

(139a) Which card do you put __ into the slot?

(139b) Which slot do you put the card into __?

So, a gap can occur if it is **licensed** by a filler. Conversely, fillers can only occur if there is an appropriate gap elsewhere in the sentence, as shown by the following examples.

(140a) *Kim knows who you like Jody.

(140b) *This music, you really like hip-hop.

(141a) *Which card do you put this into the slot?

(141b) *Which slot do you put the card into this one?

The mutual co-occurence between filler and gap leads to (137) – (138) is sometimes termed a "dependency". One issue of considerable importance in theoretical linguistics has been the nature of the material that can intervene between a filler and the gap that it licenses; in particular, can we simply list a finite set of strings that separate the two? The answer is No: there is no upper bound on the distance between filler and gap. This fact can be easily illustrated with constructions involving sentential complements, as shown in (142).

(142a) Who do you like __?

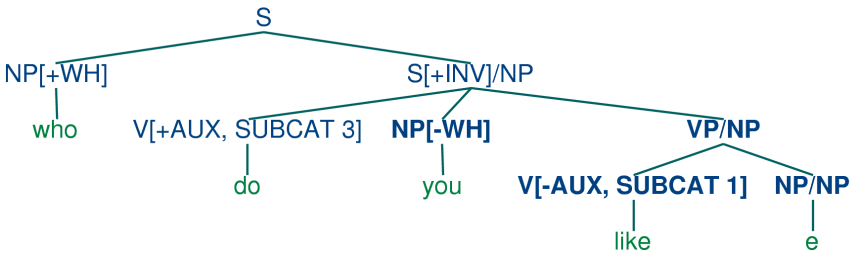
(142b) Who do you claim that you like __?

(142c) Who do you claim that Jody says that you like __?

Since we can have indefinitely deep recursion of sentential complements, the gap can be embedded indefinitely far inside the whole sentence. This constellation of properties leads to the notion of an **unbounded dependency construction**; that is, a filler-gap dependency where there is no upper bound on the distance between filler and gap.

A variety of mechanisms have been suggested for handling unbounded dependencies in formal grammars; we shall adopt an approach due to Generalized Phrase Structure Grammar that involves something called **slash categories**. A slash category is something of the form Y/XP; we interpret this as a phrase of category Y which is somewhere missing a sub-constituent of category XP. For example, S/NP is an S which is missing an NP. The use of slash categories is illustrated in (143).

(143)



The top part of the tree introduces the filler *who* (treated as an expression of category NP[+WH]) together with a corresponding gap-containing constituent S/NP. The gap information is then "percolated" down the tree via the VP/NP category, until it reaches the category NP/NP. At this point, the dependency is discharged by realizing the gap information as the empty string *e* immediately dominated by NP/NP.

Do we need to think of slash categories as a completely new kind of object in our grammars? Fortunately, no, we don't — in fact, we can accommodate them within our existing feature-based framework. We do this by treating slash as a feature, and the category to its right as a value. In other words, our "official" notation for S/NP will be S[SLASH = *NP*]. Once we have taken this step, it is straightforward to write a small grammar in NLTK for analyzing unbounded dependency constructions. (144) illustrates the main principles of slash categories, and also includes rules for inverted clauses. To simplify presentation, we have omitted any specification of tense on the verbs.

(144)

```
% start S
############################
# Grammar Rules
############################
```

```
S[-INV] -> NP  S/NP
S[-INV]/?x -> NP VP/?x
S[+INV]/?x -> V[+AUX] NP VP/?x
S-BAR/?x -> Comp S[-INV]/?x

NP/NP ->

VP/?x -> V[SUBCAT=1, -AUX] NP/?x
VP/?x -> V[SUBCAT=2, -AUX] S-BAR/?x
VP/?x -> V[SUBCAT=3, +AUX] VP/?x

############################
# Lexical Rules
############################
V[SUBCAT=1, -AUX] -> 'see' | 'like'
V[SUBCAT=2, -AUX] -> 'say' | 'claim'
V[SUBCAT=3, +AUX] -> 'do' | 'can'

NP[-WH] -> 'you' | 'children' | 'girls'
NP[+WH] -> 'who'

Comp -> 'that'
```

(144) contains one gap-introduction rule, namely

(145) `S[-INV] → NP S/NP`

In order to percolate the slash feature correctly, we need to add slashes with variable values to both sides of the arrow in rules which expand S, VP and NP. For example,

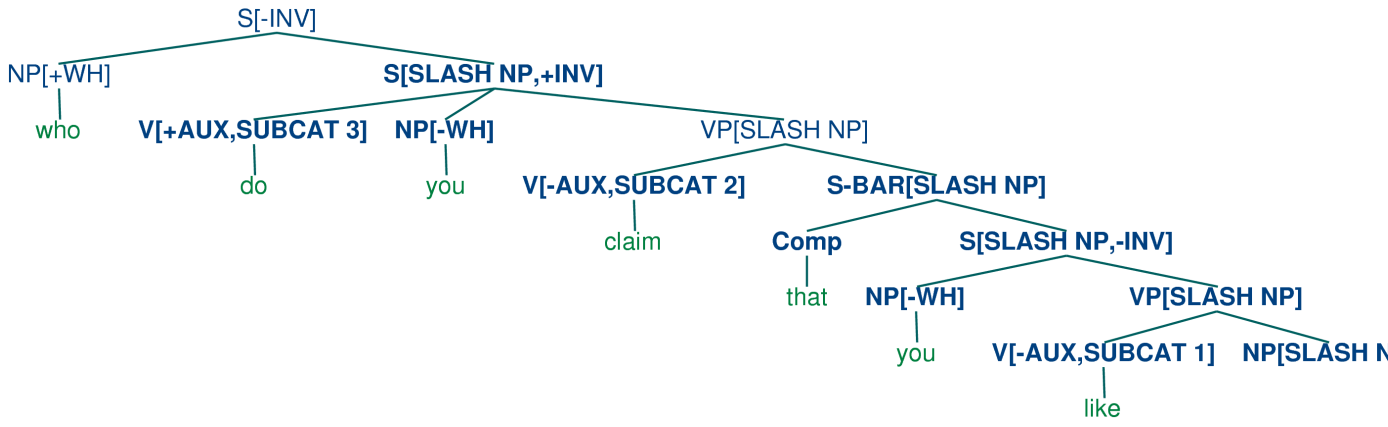
(146) `VP/?X → V S-BAR/?X`

says that a slash value can be specified on the VP mother of a constituent if the same value is also specified on the S-BAR daughter. Finally, (147) allows the slash information on NP to be discharged as the empty string.

(147) `NP/NP →`

Using (144), we can parse the string *who do you claim that you like* into the tree shown in (148).

(148)



9.4.5 Case and Gender in German

Compared with English, German has a relatively rich morphology for agreement. For example, the definite article in German varies with case, gender and number, as shown in Table 9.2.

Case	Masc	Fem	Neut	Plural
Nom	der	die	das	die
Gen	des	der	des	der
Dat	dem	der	dem	den
Acc	den	die	das	die

Table 9.2: Morphological Paradigm for the German definite Article

Subjects in German take the nominative case, and most verbs govern their objects in the accusative case. However, there are exceptions like *helfen* which govern the dative case.

(149a)	Die the.NOM.FEM.SG 'the cat sees the dog'	katze cat.3.FEM.SG	sieht see.3.SG	dem the.ACC.MASC.SG	hund dog.3.MASC.SG
(149b)	*Die the.NOM.FEM.SG	katze cat.3.FEM.SG	sieht see.3.SG	den the.DAT.MASC.SG	hund dog.3.MASC.SG
(150a)	Die the.NOM.FEM.SG 'the cat helps the dog'	katze cat.3.FEM.SG	hilft help.3.SG	den the.DAT.MASC.SG	hund dog.3.MASC.SG
(150b)	*Die the.NOM.FEM.SG	katze cat.3.FEM.SG	hilft help.3.SG	dem the.ACC.MASC.SG	hund dog.3.MASC.SG

The grammar (151) illustrates the interaction of agreement (comprising person, number and gender) with case.

(151)

```
% start S
################
# Grammar Rules
################
S -> NP[CASE=nom, AGR=?a] VP[AGR=?a]

NP[CASE=?c, AGR=?a] -> PRO[CASE=?c, AGR=?a]
NP[CASE=?c, AGR=?a] -> Det[CASE=?c, AGR=?a] N[CASE=?c, AGR=?a]

VP[AGR=?a] -> IV[AGR=?a]
VP[AGR=?a] -> TV[OBJCASE=?c, AGR=?a] NP[CASE=?c]

###############
# Lexical Rules
###############
```

```
Det[CASE=nom, AGR=[GND=masc,PER=3,NUM=sg]] -> 'der'
Det[CASE=acc, AGR=[GND=masc,PER=3,NUM=sg]] -> 'den'
Det[CASE=dat, AGR=[GND=masc,PER=3,NUM=sg]] -> 'dem'
Det[AGR=[PER=3,NUM=pl]] -> 'die'
Det[CASE=nom, AGR=[GND=fem,PER=3]] -> 'die'
Det[CASE=acc, AGR=[GND=fem,PER=3]] -> 'die'
Det[CASE=dat, AGR=[GND=fem,PER=3]] -> 'der'

N[AGR=[GND=masc,PER=3,NUM=sg]] -> 'hund'
N[AGR=[GND=masc,PER=3,NUM=pl]] -> 'hunde'
N[AGR=[PER=3,NUM=pl]] -> 'hunde'
N[AGR=[GND=fem,PER=3,NUM=sg]] -> 'katze'
N[AGR=[GND=fem,PER=3,NUM=pl]] -> 'katzen'

PRO[CASE=nom, AGR=[PER=1,NUM=sg]] -> 'ich'
PRO[CASE=acc, AGR=[PER=1,NUM=sg]] -> 'mich'

TV[OBJCASE=acc, AGR=[NUM=sg,PER=1]] -> 'sehe'
TV[OBJCASE=acc, AGR=[NUM=sg,PER=3]] -> 'sieht' | 'mag'
TV[OBJCASE=acc, AGR=[NUM=pl]] -> 'siehen' | 'moegen'
TV[OBJCASE=dat, AGR=[NUM=sg,PER=1]] -> 'folge' | 'helfe'
TV[OBJCASE=dat, AGR=[NUM=sg,PER=3]] -> 'folgt' | 'hilft'
TV[OBJCASE=dat, AGR=[NUM=pl]] -> 'folgen' | 'helfen'

IV[AGR=[NUM=sg,PER=3]] -> 'kommt'
IV[AGR=[NUM=sg,PER=1]] -> 'komme'
IV[AGR=[NUM=pl]] -> 'kommen'
```

As you will see, the feature OBJCASE is used to specify the case which the verb governs on its object.

9.4.6 Exercises

1. ☼ Modify the grammar illustrated in (120) to incorporate a BAR feature for dealing with phrasal projections.

2. ☼ Modify the German grammar in (151) to incorporate the treatment of subcategorization presented in 9.4.1.

3. ◑ Extend the German grammar in (151) so that it can handle so-called verb-second structures like the following:

   (152) Heute sieht der hund die katze.

4. ★ Morphological paradigms are rarely completely regular, in the sense of every cell in the matrix having a different realisation. For example, the present tense conjugation of the lexeme WALK only has two distinct forms: *walks* for the 3rd person singular, and *walk* for all other combinations of person and number. A successful analysis should not require redundantly specifying that 5 out of the 6 possible morphological combinations have the same realization. Propose and implement a method for dealing with this.

5. ★ So-called **head features** are shared between the mother and head daughter. For example, TENSE is a head feature that is shared between a VP and its head V daughter. See [Gazdar et al., 1985] for more details. Most of the features we have looked at are head features — exceptions are SUBCAT and SLASH. Since the sharing of head features is predictable, it should not need to be stated explicitly in the grammar rules. Develop an approach which automatically accounts for this regular behaviour of head features.

9.5 Summary

- The traditional categories of context-free grammar are atomic symbols. An important motivation feature structures is to capture fine-grained distinctions which would otherwise require a masssive multiplication of atomic categories.
- By using variables over feature values, we can express constraints in grammar rules which allow the realization of different feature specifications to be inter-dependent.
- Typically we specify fixed values of features at the lexical level and constrain the values of features in phrases to unify with the corresponding values in their daughters.
- Feature values are either atomic or complex. A particular subcase of atomic value is the Boolean value, represented by convention as [+/- F].
- Two features can share a value (either atomic or complex). Structures with shared values are said to be re-entrant. Shared values are represented by numerical indices (or tags) in AVMs.
- A path in a feature structure is a tuple of features corresponding to the labels on a sequence of arcs from the root of the graph representation.
- Two paths are equivalent if they share a value.
- Feature structures are partially ordered by subsumption. FS_0 subsumes FS_1 when FS_0 is more general (less informative) than FS_1.
- The unification of two structures FS_0 and FS_1, if successful, is the feature structure FS_2 which contains the combined information of both FS_0 and FS_1.
- If unification specialises a path π in FS, then it also specialises every path π' equivalent to π.
- We can use feature structures to build succinct analyses of a wide variety of linguistic phenomena, including verb subcategorization, inversion constructions, unbounded dependency constructions and case government.

9.6 Further Reading

The earliest use of features in theoretical linguistics was designed to capture phonological properties of phonemes. For example, a sound like **/b/** might be decomposed into the structure [+LABIAL, +VOICE]. An important motivation was to capture generalizations across classes of segments; for example, that **/n/** gets realized as **/m/** preceding any +LABIAL consonant. Within Chomskyan grammar, it was standard to use atomic features for phenomena like agreement, and also to capture generalizations across syntactic

categories, by analogy with phonology. A radical expansion of the use of features in theortical syntax was advocated by Generalized Phrase Structure Grammar (GPSG; [Gazdar et al., 1985]), particularly in the use of features with complex values.

Coming more from the perspective of computational linguistics, [Kay, 1985] proposed that functional aspects of language could be captured by unification of attribute-value structures, and a similar approach was elaborated by [Shieber et al., 1983] within the PATR-II formalism. Early work in Lexical-Functional grammar (LFG; [Kaplan and Bresnan, 1982]) introduced the notion of an **f-structure** which was primarily intended to represent the grammatical relations and predicate-argument structure associated with a constituent structure parse. [Shieber, 1986] provides an excellent introduction to this phase of research into feature-based grammars.

One conceptual difficulty with algebraic approaches to feature structures arose when researchers attempted to model negation. An alternative perspective, pioneered by [Kasper and Rounds, 1986] and [Johnson, 1988], argues that grammars involve *descriptions* of feature structures rather than the structures themselves. These descriptions are combined using logical operations such as conjunction, and negation is just the usual logical operation over feature descriptions. This description-oriented perspective was integral to LFG from the outset (cf. [Kaplan, 1989], and was also adopted by later versions of Head-Driven Phrase Structure Grammar (HPSG; [Sag and Wasow, 1999]).

Feature structures, as presented in this chapter, are unable to capture important constraints on linguistic information. For example, there is no way of saying that the only permissible values for NUM are *sg* and *pl*, while a specifcation such as [NUM *masc*] is anomalous. Similarly, we cannot say that the complex value of AGR *must* contain specifications for the features PER, NUM and GND, but *cannot* contain a specification such as [SUBCAT 3]. **Typed feature structures** were developed to remedy this deficiency. To begin with, we stipulate that feature values are always typed. In the case of atomic values, the values just are types. For example, we would say that the value of NUM is the type *num*. Moreover, *num* is the most general type of value for NUM. Since types are organized hierarchically, we can be more informative by specifying the value of NUM is a **subtype** of *num*, namely either *sg* or *pl*.

In the case of complex values, we say that feature structures are themselves typed. So for example the value of AGR will be a feature structure of type *agr*. We also stipulate that all and only PER, NUM and GND are **appropriate** features for a structure of type *agr*. A good early review of work on typed feature structures is [Emele and Zajac, 1990]. A more comprehensive examination of the formal foundations can be found in [Carpenter, 1992], while [Copestake, 2002] focusses on implementing an HPSG-oriented approach to typed feature structures.

There is a copious literature on the analysis of German within feature-based grammar frameworks. [Nerbonne et al., 1994] is a good starting point for the HPSG literature on this topic, while [Müller, 1999] gives a very extensive and detailed analysis of German syntax in HPSG.

Part III

ADVANCED TOPICS

Chapter 10

Advanced Programming in Python

This chapter introduces concepts in algorithms, data structures, program design, and advanced Python programming. It also contains review of the basic mathematical notions of set, relation, and function, and illustrates them in terms of Python data structures. It contains many working program fragments which you should try yourself.

10.1 Object-Oriented Programming in Python

Object-Oriented Programming is a programming paradigm in which complex structures and processes are decomposed into modules, each encapsulating a single data type and the legal operations on that type.

10.1.1 Variable scope (notes)

- local and global variables
- scope rules
- global variables introduce dependency on context and limits the reusability of a function
- importance of avoiding side-effects
- functions hide implementation details

10.1.2 Modules

10.1.3 Data Classes: Trees in NLTK

An important data type in language processing is the syntactic tree. Here we will review the parts of the NLTK code which defines the `Tree` class.

The first line of a class definition is the `class` keyword followed by the class name, in this case `Tree`. This class is derived from Python's built-in `list` class, permitting us to use standard list operations to access the children of a tree node.

```
>>> class Tree(list):
```

Next we define the initializer, also known as the *constructor*. It has a special name, starting and ending with double underscores; Python knows to call this function when you asks for a new tree object by writing `t = Tree(node, children)`. The constructor's first argument is special, and is standardly called `self`, giving us a way to refer to the current object from inside the definition. This constructor calls the list initializer (similar to calling `self = list(children)`), then defines the `node` property of a tree.

```
...     def __init__(self, node, children):
...         list.__init__(self, children)
...         self.node = node
```

Next we define another special function that Python knows to call when we index a Tree. The first case is the simplest, when the index is an integer, e.g. `t[2]`, we just ask for the list item in the obvious way. The other cases are for handling slices, like `t[1:2]`, or `t[:]`.

```
...     def __getitem__(self, index):
...         if isinstance(index, int):
...             return list.__getitem__(self, index)
...         else:
...             if len(index) == 0:
...                 return self
...             elif len(index) == 1:
...                 return self[int(index[0])]
...             else:
...                 return self[int(index[0])][index[1:]]
...
```

This method was for accessing a child node. Similar methods are provided for setting and deleting a child (using `__setitem__`) and `__delitem__`).

Two other special member functions are `__repr__()` and `__str__()`. The `__repr__()` function produces a string representation of the object, one which can be executed to re-create the object, and is accessed from the interpreter simply by typing the name of the object and pressing 'enter'. The `__str__()` function produces a human-readable version of the object; here we call a pretty-printing function we have defined called `pp()`.

```
...     def __repr__(self):
...         import string
...         childstr = string.join([repr(c) for c in self])
...         return '(%s: %s)' % (self.node, childstr)
...     def __str__(self):
...         return self.pp()
```

Next we define some member functions that do other standard operations on trees. First, for accessing the leaves:

```
...     def leaves(self):
...         leaves = []
...         for child in self:
...             if isinstance(child, Tree):
...                 leaves.extend(child.leaves())
...             else:
...                 leaves.append(child)
...         return leaves
```

Next, for computing the height:

```
...     def height(self):
...         max_child_height = 0
...         for child in self:
...             if isinstance(child, Tree):
...                 max_child_height = max(max_child_height, child.height())
...             else:
...                 max_child_height = max(max_child_height, 1)
...         return 1 + max_child_height
```

And finally, for enumerating all the subtrees (optionally filtered):

```
...     def subtrees(self, filter=None):
...         if not filter or filter(self):
...             yield self
...         for child in self:
...             if isinstance(child, Tree):
...                 for subtree in child.subtrees(filter):
...                     yield subtree
```

10.1.4 Processing Classes: N-gram Taggers in NLTK

This section will discuss the `tag.ngram` module.

10.2 Program Development

Programming is a skill which is acquired over several years of experience with a variety of programming languages and tasks. Key high-level abilities are *algorithm design* and its manifestation in *structured programming*. Key low-level abilities include familiarity with the syntactic constructs of the language, and knowledge of a variety of diagnostic methods for trouble-shooting a program which does not exhibit the expected behaviour.

10.2.1 Programming Style

We have just seen how the same task can be performed in different ways, with implications for efficiency. Another factor influencing program development is *programming style*. Consider the following program to compute the average length of words in the Brown Corpus:

```
>>> from nltk_lite.corpora import brown
>>> count = 0
>>> total = 0
>>> for sent in brown.raw('a'):
...     for token in sent:
...         count += 1
...         total += len(token)
>>> print float(total) / count
4.2765382469
```

In this program we use the variable `count` to keep track of the number of tokens seen, and `total` to store the combined length of all words. This is a low-level style, not far removed from machine code,

the primitive operations performed by the computer's CPU. The two variables are just like a CPU's registers, accumulating values at many intermediate stages, values which are almost meaningless. We say that this program is written in a *procedural* style, dictating the machine operations step by step. Now consider the following program which computes the same thing:

```
>>> tokens = [token for sent in brown.raw('a') for token in sent]
>>> total = sum(map(len, tokens))
>>> print float(total)/len(tokens)
4.2765382469
```

The first line uses a list comprehension to construct the sequence of tokens. The second line *maps* the `len` function to this sequence, to create a list of length values, which are summed. The third line computes the average as before. Notice here that each line of code performs a complete, meaningful action. Moreover, they do not dictate how the computer will perform the computations; we state high level relationships like "`total` is the sum of the lengths of the tokens" and leave the details to the Python interpreter. Accordingly, we say that this program is written in a *declarative* style.

Here is another example to illustrate the procedural/declarative distinction. Notice again that the procedural version involves low-level steps and a variable having meaningless intermediate values:

```
>>> word_list = []
>>> for sent in brown.raw('a'):
...     for token in sent:
...         if token not in word_list:
...             word_list.append(token)
>>> word_list.sort()
```

The declarative version (given second) makes use of higher-level built-in functions:

```
>>> tokens = [word for sent in brown.raw('a') for word in sent]
>>> word_list = list(set(tokens))
>>> word_list.sort()
```

What do these programs compute? Which version did you find easier to interpret?

Consider one further example, which sorts three-letter words by their final letters. The words come from the widely-used Unix word-list, made available as an NLTK corpus called `words`. Two words ending with the same letter will be sorted according to their second-last letters. The result of this sort method is that many rhyming words will be contiguous. Two programs are given; Which one is more declarative, and which is more procedural?

As an aside, for readability we define a function for reversing strings that will be used by both programs:

```
>>> def reverse(word):
...     return word[::-1]
```

Here's the first program. We define a helper function `reverse_cmp` which calls the built-in `cmp` comparison function on reversed strings. The `cmp` function returns `-1`, `0`, or `1`, depending on whether its first argument is less than, equal to, or greater than its second argument. We tell the list sort function to use `reverse_cmp` instead of `cmp` (the default).

```
>>> from nltk_lite.corpora import words
>>> def reverse_cmp(x,y):
...     return cmp(reverse(x), reverse(y))
```

```
>>> word_list = [word for word in words.raw('en') if len(word) == 3]
>>> word_list.sort(reverse_cmp)
>>> print word_list[-12:]
['toy', 'spy', 'cry', 'dry', 'fry', 'pry', 'try', 'buy', 'guy', 'ivy',
'Paz', 'Liz']
```

Here's the second program. In the first loop it collects up all the three-letter words in reversed form. Next, it sorts the list of reversed words. Then, in the second loop, it iterates over each position in the list using the variable `i`, and replaces each item with its reverse. We have now re-reversed the words, and can print them out.

```
>>> word_list = []
>>> for word in words.raw('en'):
...     if len(word) == 3:
...         word_list.append(reverse(word))
>>> word_list.sort()
>>> for i in range(len(word_list)):
...     word_list[i] = reverse(word_list[i])
>>> print word_list[-12:]
['toy', 'spy', 'cry', 'dry', 'fry', 'pry', 'try', 'buy', 'guy', 'ivy',
'Paz', 'Liz']
```

Choosing between procedural and declarative styles is just that, a question of style. There are no hard boundaries, and it is possible to mix the two. Readers new to programming are encouraged to experiment with both styles, and to make the extra effort required to master higher-level constructs, such as list comprehensions, and built-in functions like `map` and `filter`.

10.2.2 Debugging

```
>>> import pdb
>>> import mymodule
>>> pdb.run('mymodule.test()')
```

Commands:
list [first [,last]]: list sourcecode for the current file
next: continue execution until the next line in the current function is reached
cont: continue execution until a breakpoint is reached (or the end of the program)
break: list the breakpoints
break n: insert a breakpoint at this line number in the current file
break file.py:n: insert a breakpoint at this line in the specified file
break function: insert a breakpoint at the first executable line of the function

10.2.3 Case Study: T9

10.2.4 Exercises

1. ☼ Write a program to sort words by length. Define a helper function `cmp_len` which uses the `cmp` comparison function on word lengths.

2. ◑ Consider the tokenized sentence `['The', 'dog', 'gave', 'John', 'the', 'newspaper']`. Using the `map()` and `len()` functions, write a single line program to convert this list of tokens into a list of token lengths: `[3, 3, 4, 4, 3, 9]`

Figure 10.1: T9: Text on 9 Keys

10.3 XML

1. ◑ Write a recursive function to produce an XML representation for a tree, with non-terminals represented as XML elements, and leaves represented as text content, e.g.:

```
<S>
    <NP type="SBJ">
        <NP>
            <NNP>Pierre</NNP>
            <NNP>Vinken</NNP>
        </NP>
        <COMMA>,</COMMA>
        <ADJP>
            <NP>
                <CD>61</CD>
                <NNS>years</NNS>
            </NP>
            <JJ>old</JJ>
        <COMMA>,</COMMA>
    </NP>
    <VP>
        <MD>will</MD>
        <VP>
            <VB>join</VB>
            <NP>
                <DT>the</DT>
                <NN>board</NN>
            </NP>
            <PP type="CLR">
                <IN>as</IN>
                <NP>
                    <DT>a</DT>
                    <JJ>nonexecutive</JJ>
```

<NN>director</NN>
</NP>
</PP>
<NP type="TMP">
<NNP>Nov.</NNP>
<CD>29</CD>
</NP>
</VP>
</VP>
<PERIOD>.</PERIOD>
</S>

10.4 Algorithm Design

An *algorithm* is a "recipe" for solving a problem. For example, to multiply 16 by 12 we might use any of the following methods:

1. Add 16 to itself 12 times over
2. Perform "long multiplication", starting with the least-significant digits of both numbers
3. Look up a multiplication table
4. Repeatedly halve the first number and double the second, 16*12 = 8*24 = 4*48 = 2*96 = 192
5. Do 10*12 to get 120, then add 6*12

Each of these methods is a different algorithm, and requires different amounts of computation time and different amounts of intermediate information to store. A similar situation holds for many other superficially simple tasks, such as sorting a list of words. Now, as we saw above, Python provides a built-in function `sort()` that performs this task efficiently. However, NLTK-Lite also provides several algorithms for sorting lists, to illustrate the variety of possible methods. To illustrate the difference in efficiency, we will create a list of 1000 numbers, randomize the list, then sort it, counting the number of list manipulations required.

```
>>> from random import shuffle
>>> a = range(1000)                    # [0,1,2,...999]
>>> shuffle(a)                         # randomize
```

Now we can try a simple sort method called *bubble sort*, which scans through the list many times, exchanging adjacent items if they are out of order. It sorts the list `a` in-place, and returns the number of times it modified the list:

```
>>> from nltk_lite.misc import sort
>>> sort.bubble(a)
250918
```

We can try the same task using various sorting algorithms. Evidently *merge sort* is much better than bubble sort, and *quicksort* is better still.

```
>>> shuffle(a); sort.merge(a)
6175
>>> shuffle(a); sort.quick(a)
2378
```

Readers are encouraged to look at `nltk_lite.misc.sort` to see how these different methods work. The collection of NLTK-Lite modules exemplify a variety of algorithm design techniques, including brute-force, divide-and-conquer, dynamic programming, and greedy search. Readers who would like a systematic introduction to algorithm design should consult the resources mentioned at the end of this tutorial.

10.4.1 Decorate-Sort-Undecorate

In Chapter 6 we saw how to sort a list of items according to some property of the list.

```
>>> words = 'I turned off the spectroroute'.split()
>>> words.sort(cmp)
>>> words
['I', 'off', 'spectroroute', 'the', 'turned']
>>> words.sort(lambda x, y: cmp(len(y), len(x)))
>>> words
['spectroroute', 'turned', 'off', 'the', 'I']
```

This is inefficient when the list of items gets long, as we compute `len()` twice for every comparison (about 2nlog(n) times). The following is more efficient:

```
>>> [pair[1] for pair in sorted((len(w), w) for w in words)[::-1]]
['spectroroute', 'turned', 'the', 'off', 'I']
```

This technique is called **decorate-sort-undecorate**. We can compare its performance by timing how long it takes to execute it a million times.

```
>>> from timeit import Timer
>>> Timer("sorted(words, lambda x, y: cmp(len(y), len(x)))",
...       "words='I turned off the spectroroute'.split()").timeit()
8.3548779487609863
>>> Timer("[pair[1] for pair in sorted((len(w), w) for w in words)]",
...      "words='I turned off the spectroroute'.split()").timeit()
9.9698889255523682
```

MORE: consider what happens as the lists get longer...

10.4.2 Problem Transformation (aka Transform-and-Conquer)

Find words which, when reversed, make legal words. Extremely wasteful solution:

```
>>> from nltk_lite.corpora import words
>>> for word1 in words.raw():
...     for word2 in words.raw():
...         if word1 == word2[::-1]:
...             print word1
```

More efficient:

```
>>> from nltk_lite.corpora import words
>>> wordlist = set(words.raw())
>>> rev_wordlist = set(w[::-1] for w in wordlist)
>>> sorted(wordlist.intersection(rev_wordlist))
['ah', 'are', 'bag', 'ban', 'bard', 'bat', 'bats', 'bib', 'bob', 'boob', 'brag',
'bud', 'buns', 'bus', 'but', 'civic', 'dad', 'dam', 'decal', 'deed', 'deeps', 'deer',
'deliver', 'denier', 'desserts', 'deus', 'devil', 'dial', 'diaper', 'did', 'dim',
'dog', 'don', 'doom', 'drab', 'draw', 'drawer', 'dub', 'dud', 'edit', 'eel', 'eke',
'em', 'emit', 'era', 'ere', 'evil', 'ewe', 'eye', 'fires', 'flog', 'flow', 'gab',
'gag', 'garb', 'gas', 'gel', 'gig', 'gnat', 'god', 'golf', 'gulp', 'gum', 'gums',
'guns', 'gut', 'ha', 'huh', 'keel', 'keels', 'keep', 'knits', 'laced', 'lager',
'laid', 'lap', 'lee', 'leek', 'leer', 'leg', 'leper', 'level', 'lever', 'liar',
'live', 'lived', 'loop', 'loops', 'loot', 'loots', 'mad', 'madam', 'me', 'meet',
'mets', 'mid', 'mood', 'mug', 'nab', 'nap', 'naps', 'net', 'nip', 'nips', 'no',
'nod', 'non', 'noon', 'not', 'now', 'nun', 'nuts', 'on', 'pal', 'pals', 'pan',
'pans', 'par', 'part', 'parts', 'pat', 'paws', 'peek', 'peels', 'peep', 'pep',
'pets', 'pin', 'pins', 'pip', 'pit', 'plug', 'pool', 'pools', 'pop', 'pot', 'pots',
'pup', 'radar', 'rail', 'rap', 'rat', 'rats', 'raw', 'redder', 'redraw', 'reed',
'reel', 'refer', 'regal', 'reined', 'remit', 'repaid', 'repel', 'revel', 'reviled',
'reviver', 'reward', 'rotator', 'rotor', 'sag', 'saw', 'sees', 'serif', 'sexes',
'slap', 'sleek', 'sleep', 'sloop', 'smug', 'snap', 'snaps', 'snip', 'snoops',
'snub', 'snug', 'solos', 'span', 'spans', 'spat', 'speed', 'spin', 'spit', 'spool',
'spoons', 'spot', 'spots', 'stab', 'star', 'stem', 'step', 'stew', 'stink', 'stool',
'stop', 'stops', 'strap', 'straw', 'stressed', 'stun', 'sub', 'sued', 'swap', 'tab',
'tang', 'tap', 'taps', 'tar', 'teem', 'ten', 'tide', 'time', 'timer', 'tip', 'tips',
'tit', 'ton', 'tool', 'top', 'tops', 'trap', 'tub', 'tug', 'war', 'ward', 'warder',
'warts', 'was', 'wets', 'wolf', 'won']
```

Observe that this output contains redundant information; each word and its reverse is included. How could we remove this redundancy?

Presorting, sets:

Find words which have at least (or exactly) one instance of all vowels. Instead of writing extremely complex regular expressions, some simple preprocessing does the trick:

```
>>> words = ["sequoia", "abacadabra", "yiieeaouuu!"]
>>> vowels = "aeiou"
>>> [w for w in words if set(w).issuperset(vowels)]
['sequoia', 'yiieeaouuu!']
>>> [w for w in words if sorted(c for c in w if c in vowels) == list(vowels)]
['sequoia']
```

10.4.3 Exercises

1. ◑ Consider again the problem of hyphenation across linebreaks. Suppose that you have successfully written a tokenizer that returns a list of strings, where some strings may contain a hyphen followed by a newline character, e.g. `long-\nterm`. Write a function which iterates over the tokens in a list, removing the newline character from each, in each of the following ways:

   a) Use doubly-nested for loops. The outer loop will iterate over each token in the list, while the inner loop will iterate over each character of a string.

   b) Replace the inner loop with a call to `re.sub()`

c) Finally, replace the outer loop with call to the `map()` function, to apply this substitution to each token.

d) Discuss the clarity (or otherwise) of each of these approaches.

2. ★ Develop a simple extractive summarization tool, which prints the sentences of a document which contain the highest total word frequency. Use `FreqDist` to count word frequencies, and use `sum` to sum the frequencies of the words in each sentence. Rank the sentences according to their score. Finally, print the *n* highest-scoring sentences in document order. Carefully review the design of your program, especially your approach to this double sorting. Make sure the program is written as clearly as possible.

10.5 Search

Many NLP tasks can be construed as search problems. For example, the task of a parser is to identify one or more parse trees for a given sentence. As we saw in Part II, there are several algorithms for parsing. A *recursive descent parser* performs **backtracking search**, applying grammar productions in turn until a match with the next input word is found, and backtracking when there is no match. We saw in Chapter 8 that the space of possible parse trees is very large; a parser can be thought of as providing a relatively efficient way to find the right solution(s) within a very large space of candidates.

As another example of search, suppose we want to find the most complex sentence in a text corpus. Before we can begin we have to be explicit about how the complexity of a sentence is to be measured: word count, verb count, character count, parse-tree depth, etc. In the context of learning this is known as the **objective function**, the property of candidate solutions we want to optimize.

In this section we will explore some other search methods which are useful in NLP. For concreteness we will apply them to the problem of learning word segmentations in text, following the work of [Brent and Cartwright, 1995]. Put simply, this is the problem faced by a language learner in dividing a continuous speech stream into individual words. We will consider this problem from the perspective of a child hearing utterances from a parent, e.g.

(153a) doyouseethekitty

(153b) seethedoggy

(153c) doyoulikethekitty

(153d) likethedoggy

Our first challenge is simply to represent the problem: we need to find a way to separate the text content from the segmentation. We will borrow an idea from IOB-tagging (Chapter 5), by annotating each character with a boolean value to indicate whether or not a word-break appears after the character. We will assume that the learner is given the utterance breaks, since these often correspond to extended pauses. Here is a possible representation, including the initial and target segmentations:

```
>>> text = "doyouseethekittyseethedoggydoyoulikethekittylikethedoggy"
>>> seg1 = "000000000000001000000000010000000000000000100000000000"
>>> seg2 = "010010010010000100100100001010010001001000010001010000"
```

Observe that the segmentation strings consist of zeros and ones. They are one character shorter than the source text, since a text of length n can only be broken up in $n - 1$ places.

Now let's check that our chosen representation is effective. We need to make sure we can read segmented text from the representation. The following function, `segment()`, takes a text string and a segmentation string, and returns a list of strings.

Listing 38 Program to Reconstruct Segmented Text from String Representation

```
    def segment(text, segs):
        words = []
        last = 0
        for i in range(len(segs)):
            if segs[i] == '1':
                words.append(text[last:i+1])
                last = i+1
        words.append(text[last:])
        return words

>>> segment(text, seg1)
['doyouseethekitty', 'seethedoggy', 'doyoulikethekitty', 'likethedoggy']
>>> segment(text, seg2)
['do', 'you', 'see', 'the', 'kitty', 'see', 'the', 'doggy', 'do', 'you',
 'like', 'the', kitty', 'like', 'the', 'doggy']
```

Now the learning task becomes a search problem: find the bit string that causes the text string to be correctly segmented into words. Our first task is done: we have represented the problem in a way that allows us to reconstruct the data, and to focus on the information to be learned.

Now that we have effectively represented the problem we need to choose the objective function. We assume the learner is acquiring words and storing them in an internal lexicon. Given a suitable lexicon, it is possible to reconstruct the source text as a sequence of lexical items. Following [Brent and Cartwright, 1995], we can use the size of the lexicon and the amount of information needed to reconstruct the source text as the basis for an objective function, as shown in Figure 10.2.

It is a simple matter to implement this objective function, as shown in Listing 10.2.

10.5.1 Exhaustive Search

- brute-force approach
- enumerate search space, evaluate at each point
- this example: search space size is 2^{55} = 36,028,797,018,963,968

For a computer that can do 100,000 evaluations per second, this would take over 10,000 years!

10.5.2 Hill-Climbing Search

Starting from a given location in the search space, evaluate nearby locations and move to a new location only if it is an improvement on the current location.

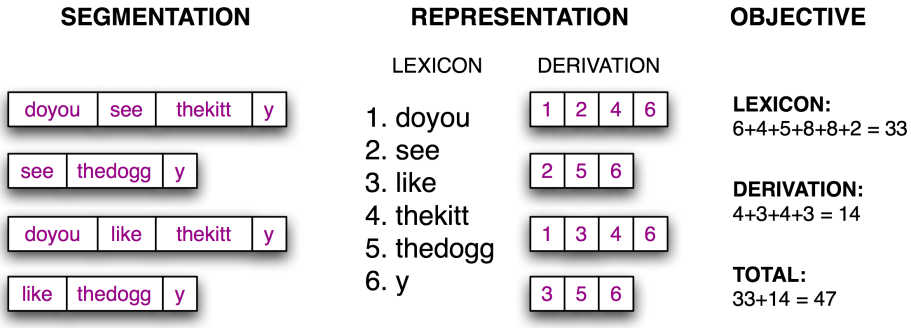


Figure 10.2: Calculation of Objective Function for Given Segmentation

Listing 39 Computing the Cost of Storing the Lexicon and Reconstructing the Source Text

```
def evaluate(text, segs):
    import string
    words = segment(text, segs)
    text_size = len(words)
    lexicon_size = len(string.join(list(set(words))))
    return text_size + lexicon_size

>>> text = "doyouseethekittyseethedoggydoyoulikethekittylikethedoggy"
>>> seg3 = "000010010000001100100000011000010001000000110001000000 1"
>>> segment(text, seg3)
['doyou', 'see', 'thekitt', 'y', 'see', 'thedogg', 'y', 'doyou', 'like',
 'thekitt', 'y', 'like', 'thedogg', 'y']
>>> evaluate(text, seg3)
47
```

Listing 40 Hill-Climbing Search

```
def flip(segs, pos):
    return segs[:pos] + `1-int(segs[pos])` + segs[pos+1:]
def hill_climb(text, segs, iterations):
        for i in range(iterations):
            pos, best = 0, evaluate(text, segs)
            for i in range(len(segs)):
                score = evaluate(text, flip(segs, i))
                if score < best:
                    pos, best = i, score
            if pos != 0:
                segs = flip(segs, pos)
                print evaluate(text, segs), segment(text, segs)
        return segs

>>> print evaluate(text, seg1), segment(text, seg1)
63 ['doyouseethekitty', 'seethedoggy', 'doyoulikethekitty', 'likethedoggy']
>>> hill_climb(text, segs1, 20)
61 ['doyouseethekittyseethedoggy', 'doyoulikethekitty', 'likethedoggy']
59 ['doyouseethekittyseethedoggydoyoulikethekitty', 'likethedoggy']
57 ['doyouseethekittyseethedoggydoyoulikethekittylikethedoggy']
```

10.5.3 Non-Deterministic Search

- Simulated annealing

10.6 Miscellany

10.6.1 Named Arguments

One of the difficulties in re-using functions is remembering the order of arguments. Consider the following function, which finds the `n` most frequent words that are at least `min_len` characters long:

```
>>> from nltk_lite.probability import FreqDist
>>> from nltk_lite import tokenize
>>> def freq_words(file, min, num):
...     freqdist = FreqDist()
...     text = open(file).read()
...     for word in tokenize.wordpunct(text):
...         if len(word) >= min:
...             freqdist.inc(word)
...     return freqdist.sorted_samples()[:num]
>>> freq_words('programming.txt', 4, 10)
['string', 'word', 'that', 'this', 'phrase', 'Python', 'list', 'words',
'very', 'using']
```

This function has three arguments. It follows the convention of listing the most basic and substantial argument first (the file). However, it might be hard to remember the order of the second and third

Listing 41 Non-Deterministic Search Using Simulated Annealing

```
    def flip_n(segs, n):
        for i in range(n):
            segs = flip(segs, randint(0,len(segs)-1))
        return segs
    def anneal(text, segs, iterations, rate):
        distance = float(len(segs))
        while distance > 0.5:
            best_segs, best = segs, evaluate(text, segs)
            for i in range(iterations):
                guess = flip_n(segs, int(round(distance)))
                score = evaluate(text, guess)
                if score < best:
                    best = score
                    best_segs = guess
            segs = best_segs
            score = best
            distance = distance/rate
            print evaluate(text, segs),
        print
        return segs

>>> anneal(text, segs, 5000, 1.2)
60 ['doyouseetheki', 'tty', 'see', 'thedoggy', 'doyouliketh', 'ekittylike', 'thedo
58 ['doy', 'ouseetheki', 'ttysee', 'thedoggy', 'doy', 'o', 'ulikethekittylike', 't
56 ['doyou', 'seetheki', 'ttysee', 'thedoggy', 'doyou', 'liketh', 'ekittylike', 't
54 ['doyou', 'seethekit', 'tysee', 'thedoggy', 'doyou', 'likethekittylike', 'thedo
53 ['doyou', 'seethekit', 'tysee', 'thedoggy', 'doyou', 'like', 'thekitty', 'like'
51 ['doyou', 'seethekittysee', 'thedoggy', 'doyou', 'like', 'thekitty', 'like', 't
42 ['doyou', 'see', 'thekitty', 'see', 'thedoggy', 'doyou', 'like', 'thekitty', 'l
```

arguments on subsequent use. We can make this function more readable by using **keyword arguments**. These appear in the function's argument list with an equals sign and a default value:

```
>>> def freq_words(file, min=1, num=10):
...     freqdist = FreqDist()
...     text = open(file).read()
...     for word in tokenize.wordpunct(text):
...         if len(word) >= min:
...             freqdist.inc(word)
...     return freqdist.sorted_samples()[:num]
```

Now there are several equivalent ways to call this function:

```
>>> freq_words('programming.txt', 4, 10)
['string', 'word', 'that', 'this', 'phrase', 'Python', 'list', 'words', 'very', 'using']
>>> freq_words('programming.txt', min=4, num=10)
['string', 'word', 'that', 'this', 'phrase', 'Python', 'list', 'words', 'very', 'using']
>>> freq_words('programming.txt', num=10, min=4)
['string', 'word', 'that', 'this', 'phrase', 'Python', 'list', 'words', 'very', 'using']
```

When we use an integrated development environment such as IDLE, simply typing the name of a function at the command prompt will list the arguments. Using named arguments helps someone to re-use the code...

A side-effect of having named arguments is that they permit optionality. Thus we can leave out any arguments for which we are happy with the default value.

```
>>> freq_words('programming.txt', min=4)
['string', 'word', 'that', 'this', 'phrase', 'Python', 'list', 'words', 'very', 'using']
>>> freq_words('programming.txt', 4)
['string', 'word', 'that', 'this', 'phrase', 'Python', 'list', 'words', 'very', 'using']
```

Another common use of optional arguments is to permit a flag, e.g.:

```
>>> def freq_words(file, min=1, num=10, trace=False):
...     freqdist = FreqDist()
...     if trace: print "Opening", file
...     text = open(file).read()
...     if trace: print "Read in %d characters" % len(file)
...     for word in tokenize.wordpunct(text):
...         if len(word) >= min:
...             freqdist.inc(word)
...             if trace and freqdist.N() % 100 == 0: print "."
...     if trace: print
...     return freqdist.sorted_samples()[:num]
```

10.6.2 Accumulative Functions

These functions start by initializing some storage, and iterate over input to build it up, before returning some final object (a large structure or aggregated result). The standard way to do this is to initialize an empty list, accumulate the material, then return the list:

```
>>> def find_nouns(tagged_text):
...     nouns = []
...     for word, tag in tagged_text:
```

```
...         if tag[:2] == 'NN':
...             nouns.append(word)
...     return nouns
```

We can apply this function to some tagged text to extract the nouns:

```
>>> tagged_text = [('the', 'DT'), ('cat', 'NN'), ('sat', 'VBD'),
...                ('on', 'IN'), ('the', 'DT'), ('mat', 'NN')]
>>> find_nouns(tagged_text)
['cat', 'mat']
```

However, a superior way to do this is to define a **generator**

```
>>> def find_nouns(tagged_text):
...     for word, tag in tagged_text:
...         if tag[:2] == 'NN':
...             yield word
```

The first time this function is called, it gets as far as the `yield` statement and stops. The calling program gets the first word and does any necessary processing. Once the calling program is ready for another word, execution of the function is continued from where it stopped, until the next time it encounters a `yield` statement.

Let's see what happens when we call the function:

```
>>> find_nouns(tagged_text)
<generator object at 0x14b2f30>
```

We cannot call it directly. Instead, we can convert it to a list.

```
>>> list(find_nouns(tagged_text))
['cat', 'mat']
```

We can also iterate over it in the usual way:

```
>>> for noun in find_nouns(tagged_text):
...     print noun,
cat mat
```

[Efficiency]

10.7 Sets and Mathematical Functions

10.7.1 Sets

Knowing a bit about sets will come in useful when you look at Chapter 11. A set is a collection of entities, called the **members** of the set. Sets can be finite or infinite, or even empty. In Python, we can define a set just by listing its members; the notation is similar to specifying a list:

```
>>> set1 = set(['a', 'b', 1, 2, 3])
>>> set1
set(['a', 1, 2, 'b', 3])
```

In mathematical notation, we would specify this set as:

(154) {'a', 'b', 1, 2, 3}

Set membership is a relation — we can ask whether some entity x belongs to a set A (in mathematical notation, written $x \in A$).

```
>>> 'a' in set1
True
>>> 'c' in set1
False
```

However, sets differ from lists in that they are *unordered* collections. Two sets are equal if and only if they have exactly the same members:

```
>>> set2 = set([3, 2, 1, 'b', 'a'])
>>> set1 == set2
True
```

The **cardinality** of a set A (written $|A|$) is the number of members in A. We can get this value using the `len()` function:

```
>>> len(set1)
5
```

The argument to the `set()` constructor can be any sequence, including a string, and just calling the constructor with no argument creates the empty set (written).

```
>>> set('123')
set(['1', '3', '2'])
>>> a = set()
>>> b = set()
>>> a == b
True
```

We can construct new sets out of old ones. The **union** of two sets A and B (written $A \cup B$) is the set of elements which belong to A or B. Union is represented in Python with `|`:

```
>>> odds = set('13579')
>>> evens = set('02468')
>>> numbers = odds | evens
>>> numbers
set(['1', '0', '3', '2', '5', '4', '7', '6', '9', '8'])
```

The **intersection** of two sets A and B (written $A \cap B$) is the set of elements which belong to both A and B. Intersection is represented in Python with `&`. If the intersection of two sets is empty, they are said to be **disjoint**.

```
>>> ints
set(['1', '0', '2', '-1', '-2'])
>>> ints & nats
set(['1', '0', '2'])
>>> odds & evens
set([])
```

The **(relative) complement** of two sets A and B (written $A - B$) is the set of elements which belong to A but not B. Complement is represented in Python with `-`.

```
>>> nats - ints
set(['3', '5', '4', '7', '6', '9', '8'])
>>> odds == nats - evens
True
>>> odds == odds - set()
True
```

So far, we have described how to define 'basic' sets and how to form new sets out of those basic ones. All the basic sets have been specified by listing all their members. Often we want to specify set membership more succinctly:

(155) the set of positive integers less than 10

(156) the set of people in Melbourne with red hair

We can informally write these sets using the following **predicate notation**:

(157) $\{x \mid x \text{ is a positive integer less than } 10\}$

(158) $\{x \mid x \text{ is a person in Melbourne with red hair}\}$

In axiomatic set theory, the axiom schema of comprehension states that given a one-place predicate P, there is set A such that for all x, x belongs to A if and only if (written $\equiv$) $P(x)$ is true:

(159) $A \forall x.(x \in A \equiv P(x))$

From a computational point of view, (159) is problematic: we have to treat sets as finite objects in the computer, but there is nothing to stop us defining infinite sets using comprehension. Now, there is a variant of (159), called the axiom of restricted comprehension, which allows us to specify a set A with a predicate P so long as we only consider xs which belong to some *already defined set* B:

(160) $\forall B\, A \forall x.\ (x \in A \equiv x \in B \wedge P(x))$

(For all sets B there is a set A such that for all x, x belongs to A if and only if x belongs to B and $P(x)$ is true.) This is equivalent to the following set in predicate notation:

(161) $\{x \mid x \in B \wedge P(x))$

(160) corresponds pretty much to what we get with list comprehension in Python: if you already have a list, then you can define a new list in terms of the old one, using an `if` condition. In other words, (162) is the Python counterpart of (160).

(162) `set([x for x in B if P(x)])`

To illustrate this further, the following list comprehension relies on the existence of the previously defined set `nats` (`n % 2` is the remainder when `n` is divided by `2`):

```
>>> nats = set(range(10))
>>> evens1 = set([n for n in nats if n % 2 == 0])
>>> evens1
set([0, 8, 2, 4, 6])
```

Now, when we defined `evens` before, what we actually had was a set of *strings*, rather than Python integers. But we can use `int` to coerce the strings to be of the right type:

```
>>> evens2 = set([int(n) for n in evens])
>>> evens1 == evens2
True
```

If every member of A is also a member of B, we say that A is a subset of B (written $A \subseteq B$). The subset relation is represented in Python with <=.

```
>>> evens1 <= nats
True
>>> set() <= nats
True
>>> evens1 <= evens1
True
```

As the above examples show, B can contain more members than A for $A \subseteq B$ to hold, but this need not be so. Every set is a subset of itself. To exclude the case where a set is a subset of itself, we use the relation **proper subset** (written $A \subset B$). In Python, this relation is represented as <.

```
>>> evens1 < nats
True
>>> evens1 < evens1
False
```

Sets can contain other sets. For instance, the set $A = \{\{a\}, \{b\}\}$ contains the two singleton sets $\{a\}$ and $\{b\}$. Note that $\{a\} \subseteq A$ does not hold, since a belongs to $\{a\}$ but not to A. In Python, it is a bit more awkward to specify sets whose members are also sets; the latter have to be defined as `frozensets`, i.e., immutable objects.

```
>>> a = frozenset('a')
>>> aplus = set([a])
>>> aplus
set([frozenset(['a'])])
```

We also need to be careful to distinguish between the empty set and the set whose only member is the empty set: $\{\}$.

10.7.2 Exercises

1. ☼ For each of the following sets, write a specification by hand in predicate notation, and an implementation in Python using list comprehension.

   a. {2, 4, 8, 16, 32, 64}
   b. {2, 3, 5, 7, 11, 13, 17}
   c. {0, 2, -2, 4, -4, 6, -6, 8, -8}

2. ☼ The **powerset** of a set A (written A) is the set of all subsets of A, including the empty set. List the members of the following sets:

   a. $\{a, b, c\}$:
   b. $\{a\}$
   c. $\{\}$
   d.

3. ◑ Write a Python function to compute the powerset of an arbitrary set. Remember that you will have to use `frozenset` for this.

10.7.3 Tuples

We write $x_1, \ldots, x_n$ for the **ordered n-tuple** of objects $x_1, \ldots, x_n$, where $n \geq 0$. These are exactly the same as Python tuples. Two tuples are equal only if they have the same lengths, and the same objects in the same order.

```
>>> tup1 = ('a', 'b', 'c')
>>> tup2 = ('a', 'c', 'b')
>>> tup1 == tup2
False
```

A tuple with just 2 elements is called an **ordered pair**, with just three elements, an **ordered triple**, and so on.

Given two sets A and B, we can form a set of ordered pairs by drawing the first member of the pair from A and the second from B. The *Cartesian product* of A and B, written $A \times B$, is the set of all such pairs. More generally, we have for any sets $S_1, \ldots, S_n$,

(163) $S_1 \times \ldots \times S_n = \{ \quad x_1, \ldots, x_n \quad \mid x_i \in S_i \}$

In Python, we can build Cartesian products using list comprehension. As you can see, the sets in a Cartesian product don't have to be distinct.

```
>>> A = set([1, 2, 3])
>>> B = set('ab')
>>> AxB = set([(a, b) for a in A for b in B])
>>> AxB
set([(1, 'b'), (3, 'b'), (3, 'a'), (2, 'a'), (2, 'b'), (1, 'a')])
>>> AxA = set([(a1, a2) for a1 in A for a2 in A])
>>> AxA
set([(1, 2), (3, 2), (1, 3), (3, 3), (3, 1), (2, 1),
(2, 3), (2, 2), (1, 1)])
```

10.7.4 Relations and Functions

In general, a **relation** R is a set of tuples. For example, in set-theoretic terms, the binary relation *kiss* is the set of all ordered pairs x, y such that *x kisses y*. More formally, an **n-ary relation** over sets S_1, $\ldots$, S_n is any set $R \subseteq S_1 \times \ldots \times S_n$.

Given a binary relation R over two sets A and B, not everything in A need stand in the R relation to something in B. As an illustration, consider the set `evens` and the relation `mod` defined as follows:

```
>>> evens = set([2, 4, 6, 8, 10])
>>> mod = set([(m,n) for m in evens for n in evens if n % m == 0 and m < n])
>>> mod
set([(4, 8), (2, 8), (2, 6), (2, 4), (2, 10)])
```

Now, `mod` $\subseteq$ `evens` $\times$ `evens`, but there are elements of `evens`, namely `6`, `8` and `10`, which do not stand in the `mod` relation to anything else in `evens`. In this case, we say that only `2` and `4` are in the **domain** of the `mod` relation. More formally, for a relation R over $A \times B$, we define

(164) $dom(R) = \{x \mid y. \quad x, y \quad \in A \times B\}$

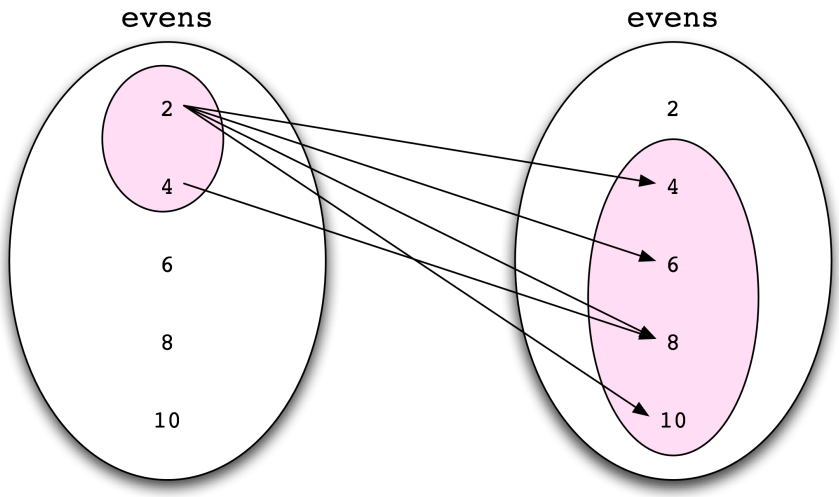


Figure 10.3: Visual Representation of a relation

Correspondingly, the set of entities in B which are the second member of a pair in R is called the **range** of R, written $ran(R)$.

We can visually represent the relation `mod` by drawing arrows to indicate elements that stand in the relation, as shown in Figure 10.3.

The domain and range of the relation are shown as shaded areas in Figure 10.3.

A relation $R \subseteq A \times B$ is a (set-theoretic) **function** just in case it meets the following two conditions:

1. For every $a \in A$ there is at most one $b \in B$ such that $\langle a, b \rangle$.
2. The domain of R is equal to A.

Thus, the `mod` relation defined earlier is not a function, since the element `2` is paired with four items, not just one. By contrast, the relation `doubles` defined as follows *is* a function:

```
>>> odds = set([1, 2, 3, 4, 5])
>>> doubles = set([(m,n) for m in odds for n in evens if n == m * 2])
>>> doubles
set([(1, 2), (5, 10), (2, 4), (3, 6), (4, 8)])
```

If f is a function $\subseteq A \times B$, then we also say that f is a function from A to B. We also write this as $f\colon A \rightarrow B$. If $\langle x, y \rangle \in f$, then we write $f(x) = y$. Here, x is called an **argument** of f and y is a **value**. In such a case, we may also say that f maps x to y.

Given that functions always map a given argument to a single value, we can also represent them in Python using dictionaries (which incidentally are also known as **mapping** objects). The `update()` method on dictionaries can take as input any iterable of key/value pairs, including sets of two-membered tuples:

```
>>> d = {}
>>> d.update(doubles)
>>> d
{1: 2, 2: 4, 3: 6, 4: 8, 5: 10}
```

A function $f\colon S_1 \times \ldots \times S_n \rightarrow T$ is called an **n-ary** function; we usually write $f(s_1, \ldots, s_n)$ rather than $f(\quad s_1, \ldots, s_n \quad)$. For sets A and B, we write A^B for the set of all functions from A to B, that is $\{f \mid f : A \rightarrow B\}$. If S is a set, then we can define a corresponding function f_S called the **characteristic function** of S, defined as follows:

(165) $f_S(x) = \text{True if } x \in S$
$f_S(x) = \text{False if } x \quad S$

f_S is a member of the set $\{True, False\}^S$.

It can happen that a relation meets condition (1) above but fails condition (2); such relations are called **partial functions**. For instance, let's slightly modify the definition of `doubles`:

```
>>> doubles2 = set([(m,n) for m in evens for n in evens if n == m * 2])
>>> doubles2
set([(2, 4), (4, 8)])
```

`doubles2` is a partial function since its domain is a proper subset of `evens`. In such a case, we say that `doubles2` is **defined** for `2` and `4` but **undefined** for the other elements in `evens`.

10.7.5 Exercises

1. ☼ Consider the relation `doubles`, where `evens` is defined as in the text earlier:

   ```
   >>> doubles = set([(m,m*2) for m in evens])
   ```

   Is `doubles` a relation over `evens`? Explain your answer.

2. ◑ What happens if you try to update a dictionary with a relation which is *not* a function?

3. ☼ Write a couple of Python functions which for any set of pairs R, return the domain and range of R.

4. ◑ Let S be a family of three children, {Bart, Lisa, Maggie}. Define relations $R \subseteq S \times S$ such that:

   a. $dom(R) \subset S$;
   b. $dom(R) = S$;
   c. $ran(R) = S$;
   d. $ran(R) = S$;
   e. R is a total function on S.
   f. R is a partial function on S.

5. ◑ Write a Python function which for any set of pairs R, returns `True` if and only if R is a function.

10.8 Further Reading

[Brent1995]

[Hunt and Thomas, 1999]

Chapter 11

Semantic Interpretation

11.1 Introduction

There are many NLP applications where it would be useful to have some representation of the *meaning* of a natural language sentence. For instance, as we pointed out in Chapter 1, current search engine technology can only take us so far in giving concise and correct answers to many questions that we might be interested in. Admittedly, Google does a good job in answering (166a), since its first hit is (166b).

(166a) What is the population of Saudi Arabia?

(166b) Saudi Arabia - Population: 26,417,599

By contrast, the result of sending (167) to Google is less helpful:

(167) Which countries border the Mediterranean?

This time, the topmost hit (and the only relevant one in the top ten) presents the relevant information as a map of the Mediterranean basin. Since the map is an image file, it is not easy to extract the required list of countries from the returned page.

Even if Google succeeds in finding documents which contain information relevant to our question, there is no guarantee that it will be in a form which can be easily converted into an appropriate answer. One reason for this is that the information may have to be inferred from more than one source. This is likely to be the case when we seek an answer to more complex questions like (168):

(168) Which Asian countries border the Mediterranean?

Here, we would probably need to combine the results of two subqueries, namely (167) and *Which countries are in Asia?*.

The example queries we have just given are based on a paper dating back to 1982 [Warren and Pereira, 1982]; this describes a system, *Chat-80*, which converts natural language questions into a semantic representation, and uses the latter to retrieve answers from a knowledge base. A knowledge base is usually taken to be a set of sentences in some formal language; in the case of Chat-80, it is a set of Prolog clauses. However, we can encode knowledge in a variety of formats, including relational databases, various kinds of graph, and first-order models. In NLTK, we have used the third of these options to re-implement a limited version of Chat-80:

```
Sentence: which Asian countries border the_Mediterranean
------------------------------
\x.(((contain x asia) and (country x)) and (border mediterranean x))
set(['turkey', 'syria', 'israel', 'lebanon'])
```

As we will explain later in this chapter, a semantic representation of the form `\x.(P x)` denotes a set of entities *u* that meet some condition `(P x)`. We then ask our knowledge base to enumerate all the entities in this set.

Let's assume more generally that knowledge is available in some structured fashion, and that it can be interrogated by a suitable query language. Then the challenge for NLP is to find a method for converting natural language questions into the target query language. An alternative paradigm for question answering is to take something like the pages returned by a Google query as our 'knowledge base' and then to carry out further analysis and processing of the textual information contained in the returned pages to see whether it does in fact provide an answer to the question. In either case, it is very useful to be able to build a semantic representation of questions. This NLP challenge intersects in interesting ways with one of the key goals of linguistic theory, namely to provide a systematic correspondence between form and meaning.

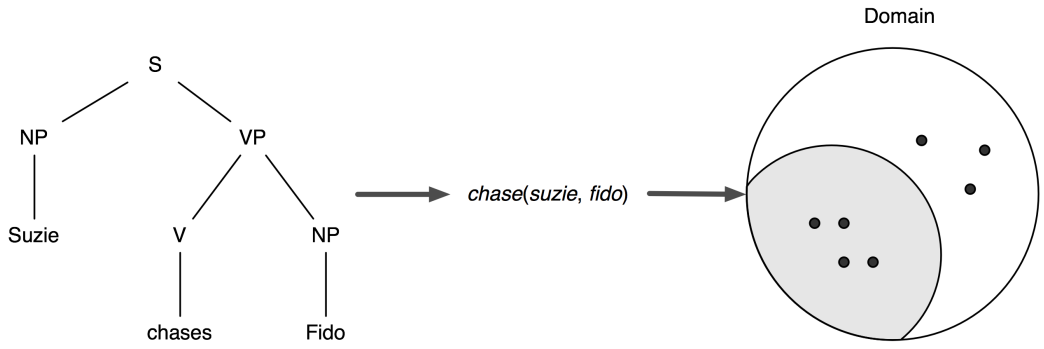
A widely adopted approach to representing meaning — or at least, some aspects of meaning — involves translating expressions of natural language into first-order logic (FOL). From a computational point of view, a strong argument in favour of FOL is that it strikes a reasonable balance between expressiveness and logical tractability. On the one hand, it is flexible enough to represent many aspects of the logical structure of natural language. On the other hand, automated theorem proving for FOL has been well studied, and although inference in FOL is not decidable, in practice many reasoning problems are efficiently solvable using modern theorem provers (cf. [Blackburn and Bos, 2005] for discussion).

While there are numerous subtle and difficult issues about how to translate natural language constructions into FOL, we will largely ignore these. The main focus of our discussion will be on a different issue, namely building semantic representations which conform to some version of the **Principle of Compositionality**. (See [Partee, 1995] for this formulation.)

Principle of Compositionality: The meaning of a whole is a function of the meanings of the parts and of the way they are syntactically combined.

There is an assumption here that the semantically relevant parts of a complex expression will be determined by a theory of syntax. Within this chapter, we will take it for granted that expressions are parsed against a context-free grammar. However, this is not entailed by the Principle of Compositionality. To summarize, we will be concerned with the task of systematically constructing a semantic representation in a manner that can be smoothly integrated with the process of parsing.

The overall framework we are assuming is illustrated in Figure (169). Given a syntactic analysis of a sentence, we can build one or more semantic representations for the sentence. Once we have a semantic representation, we can also check whether it is true in a model.



(169) syntactic analysis semantic representation truth in a model

A **model** for a logical language is a set-theoretic construction which provides a very simplified picture of how the world is. For example, in this case, the model should contain individuals (indicated in the diagram by small dots) corresponding to Suzie and Fido, and it should also specify that these individuals belong to the *chase* relation.

The order of sections in this chapter is not what you might expect from looking at the diagram. We will start off in the middle of (169) by presenting a logical language FSRL that will provide us with semantic representations in NLTK. Next, we will show how formulas in the language can be systematically evaluated in a model. At the end, we will bring everything together and describe a simple method for constructing semantic representations as part of the parse process in NLTK.

11.2 The Lambda Calculus

In a functional programming language, computation can be carried out by reducing an expression E according to specified rewrite rules. This reduction is carried out on subparts of E, and terminates when no further subexpressions can be reduced. The resulting expression $E*$ is called the **Normal Form** of E. Table 11.1 gives an example of reduction involving a simple Python expression (where ' ' means 'reduces to'):

	`len(max(['cat', 'zebra', 'rabbit'] + ['gopher']))`
	`len(max(['cat', 'zebra', 'rabbit', 'gopher']))`
	`len('zebra')`
	`5`

Table 11.1: Reduction of functions

Thus, working from the inside outwards, we first reduce list concatenation to the normal form shown in the second row, we then take the `max()` element of the list (under alphabetic ordering), and then compute the length of that string. The final expression, `5`, is considered to be the output of the program. This fundamental notion of computation is modeled in an abstract way by something called the λ-calculus (λ is a Greek letter pronounced 'lambda').

The first basic concept in the λ-calculus is **application**, represented by an expression of the form (F A), where F is considered to be a function, and A is considered to be an argument (or input) for

F. For example, (walk x) is an application. Moreover, application expressions can be applied to other expressions. So in a functional framework, binary addition might represented as ((+ x) y) rather than (x + y). Note that + is being treated as a function which is applied to its first argument x to yield a function (+ x) that is then applied to the second argument y.

The second basic concept in the λ-calculus is **abstraction**. If M[x] is an expression containing the variable x, then λx.M[x] denotes the function x → M[x]. Abstraction and application are combined in the expression (λx.((+ x) 3) 4), which denotes the function x → x + 3 applied to 4, giving 4 + 3, which is 7. In general, we have

(170) (λx.M[x] N) = M[N],

where M[N] is the result of replacing all occurences of x in M by N. This axiom of the lambda calculus is known as **β-conversion**. β-conversion is the primary form of reduction in the λ-calculus.

The module `nltk_lite.semantics.logic` can parse expressions of the λ-calculus. The λ symbol is represented as `'\'`. In order to avoid having to escape this with a second `'\'`, we use raw strings in parsable expressions.

```
>>> from nltk_lite.semantics import logic
>>> lp = logic.Parser()
>>> lp.parse(r'(walk x)')
ApplicationExpression('walk', 'x')
>>> lp.parse(r'\x.(walk x)')
LambdaExpression('x', '(walk x)')
```

An `ApplicationExpression` has subparts consisting of the function and the argument; a `LambdaExpression` has subparts consisting of the variable (e.g., `x`) that is bound by the λ and the body of the expression (e.g., `walk`).

The λ-calculus is a calculus of functions; by itself, it says nothing about logical structure. Although it is possible to define logical operators within the λ-calculus, it is much more convenient to adopt a hybrid approach which supplements the λ-calculus with logical and non-logical constants as primitives. In order to show how this is done, we turn first to the language of propositional logic.

11.3 Propositional Logic

The language of propositional logic represents certain aspects of natural language, but at a high level of abstraction. The only structure that is made explicit involves **logical connectives**; these correspond to 'logically interesting' expressions such as *and* and *not*. The basic expressions of the language are **propositional variables**, usually written p, q, r, etc. Let A be a finite set of such variables. There is a disjoint set of logical connectives which contains the unary operator $\neg$ (*not*), and binary operators $\wedge$ (*and*), $\vee$ (*or*), $\rightarrow$ (*implies*) and $\equiv$ (*iff*).

The set of formulas of L_{prop} is described inductively:

1. Every element of A is a formula of L_{prop}.
2. If φ is a formula of L_{prop} , then so is $\neg \varphi$.
3. If φ and ψ are formulas, then so are $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$, $(\varphi \rightarrow \psi)$ and $(\varphi \equiv \psi)$.
4. Nothing else is a formula of L_{prop}.

Within L_{prop}, we can construct formulas such as $p \rightarrow q \vee r$, which might represent the logical structure of an English sentence such as *if it is raining, then Kim will take an umbrella or Lee will get wet*. p stands for *it is raining*, q for *Kim will take an umbrella* and q for *Lee will get wet*.

The Boolean connectives of propositional logic are supported by `nltk_lite.semantics.logic`, and are parsed as objects of the class `ApplicationExpression` (i.e., function expressions). However, infix notation is also allowed as an input format. The connectives themselves belong to the `Operator` class of expressions.

```
>>> lp.parse('(and p q)')
ApplicationExpression('(and p)', 'q')
>>> lp.parse('(p and q)')
ApplicationExpression('(and p)', 'q')
>>> lp.parse('and')
Operator('and')
>>>
```

Since a negated proposition is syntactically an application, the unary operator `not` and its argument must be surrounded by parentheses.

```
>>> lp.parse('(not (p and q))')
ApplicationExpression('not', '(and p q)')
>>>
```

To make the `print` output easier to read, we can invoke the `infixify()` method, which places binary Boolean operators in infix position.

```
>>> e = lp.parse('(and p (not a))')
>>> e
ApplicationExpression('(and p)', '(not a)')
>>> print e
(and p (not a))
>>> print e.infixify()
(p and (not a))
```

As the name suggests, propositional logic only studies the logical structure of formulas made up of atomic propositions. We saw, for example, that propositional variables stood for whole clauses in English. In order to look at how predicates combine with arguments, we need to look at a more complex language for semantic representation, namely first-order logic. In order to show how this new language interacts with the λ-calculus, it will be useful to introduce the notion of types into our syntactic definition, in departure from the rather simple approach to defining the clauses of L_{prop}.

11.4 First-Order Logic

11.4.1 Predication

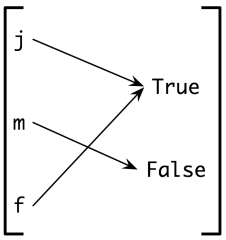
In first-order logic (FOL), propositions are analysed into predicates and arguments, which takes us a step closer to the structure of natural languages. The standard construction rules for FOL recognize **terms** such as individual variables and individual constants, and **predicates** which take differing numbers of arguments. For example, *Jane walks* might be formalized as walk(jane) and *Jane sees Mike* as see(jane, mike). We will call walk a **unary predicate**, and see a **binary predicate**. Semantically, see is modeled as a relation, i.e., a set of pairs, and the proposition is true in a situation just in case the

pair j, m belongs to this set. In order to make it explicit that we are treating *see* as a relation, we'll use the symbol see_R as its semantic representation, and we'll call $\mathsf{see}_R(\mathsf{jane}, \mathsf{mike})$ an instance of the 'relational style' of representing predication.

Within the framework of the λ-calculus, there is an alternative approach in which predication is treated as function application. In this functional style of representation, *Jane sees Mike* is formalized as ((see_f m) j) or — as a shorthand with less brackets — as (see_f m j). Rather than being modeled as a relation, see_f denotes a function. Before going into detail about this function, let's first look at a simpler case, namely the different styles of interpreting a unary predicate such as walk.

In the relational approach, walk_R denotes some set W of individuals. The formula $\mathsf{walk}_R(\mathsf{j})$ is true in a situation if and only if the individual denoted by j belongs to W. As we saw in Chapter 10, corresponding to every set S is the *characteristic function* f_S of that set. To be specific, suppose in some situation our domain of discourse D is the set containing the individuals j (Jane), m (Mike) and f (Fido); and the set of individuals that walk is $W = \{j, f\}$. So in this situation, the formulas $\mathsf{walk}_R(\mathsf{j})$ and $\mathsf{walk}_R(\mathsf{f})$ are both true, while $\mathsf{walk}_R(\mathsf{m})$ is false. Now we can use the characteristic function f_W as the interpretation of walk_f in the functional style. The diagram (171) gives a graphical representation of the mapping f_W.

(171)

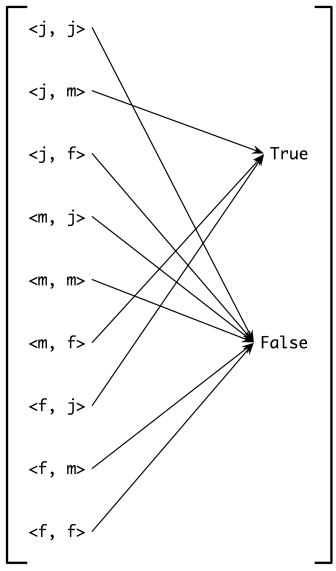


Binary relations can be converted into functions in a very similar fashion. Suppose for example that on the relational style of interpretation, see_R denotes the following set of pairs:

(172) { j, m , m, f , f, j }

That is, Jane sees Mike, Mike sees Fido, and Fido sees Jane. One option on the functional style would be to treat see_f as the expected characteristic function of this set, i.e., a function f_S: $D \times D \rightarrow$ {*True*, *False*} (i.e., from pairs of individuals to truth values). This mapping is illustrated in (173).

(173)



However, recall that we are trying to build up our semantic analysis compositionally; i.e., the meaning of a complex expression is a function of the meaning of its parts. In the case of a sentence, what are its parts? Presumably they are the subject NP and the VP. So let's consider what would be a suitable value for the VP *sees Fido*. It cannot be see_f denoting a function $D \times D \rightarrow \{True, False\}$, since this is looking for a *pair* of arguments. A better meaning representation would be $\lambda x.\mathsf{see}_R(\mathsf{x}, \mathsf{Fido})$, which is a function of a single argument, and can thus be applied to semantic representation of the subject *np*:gc. This invites the question: how should we represent the meaning of the transitive verb *see*? A possible answer is shown in (174).

(174) $\mathsf{see}_f = \lambda \mathsf{y}.\lambda x.\mathsf{see}_R(\mathsf{x}, \mathsf{y})$.

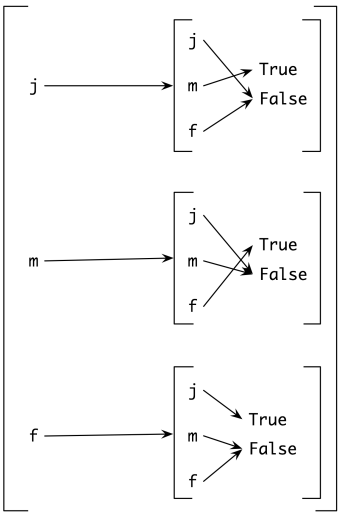
This defines see_f to be a function expression which can be applied first to the argument coming from the NP object and then to the argument comig from the NP subject. In (175), we show how the application of (174) to f and then to m gets reduced.

(175) $(\lambda \mathsf{y}.\lambda x.\mathsf{see}_R(\mathsf{x}, \mathsf{y})\ \mathsf{f}\ \mathsf{m})$ $(\lambda x.\mathsf{see}_R(\mathsf{x}, \mathsf{f})\ \mathsf{m})$ $\mathsf{see}_R(\mathsf{m}, \mathsf{f})$

(174) adopts a technique known as 'currying' (named after Haskell B. Curry), in which a binary function is converted into a function of one argument. As you can see, when we apply see_f to an argument such as f, the value is another function, namely the function denoted by $\lambda x.\mathsf{see}_R(\mathsf{x}, \mathsf{f})$.

Diagram (176) shows the curryed counterpart of (173). It presents a function *F* such that given the argument *j*, *F*(*j)* is a characteristic function that maps m to *True* and *j* and *f* to *False*. (While there are $2^3 = 8$ characteristic functions from our domain of three individuals into {*True*, *False*}, we have only shown the functions which are in the range of the function denoted by see_f.)

(176)



Now, rather than define see_f by abstracting over a formula containing see_R, we can interpret it directly as the function f: (**Ind** → (**Ind** → **Bool**)), as illustrated in (176). Table 11.2 summarizes the different approaches to predication that we have just examined.

English	Relational	Functional
Jane walks	walk(j)	(walk j)
Mike sees Fido	see(m, f)	((see f) m), (see f m)

Table 11.2: Representing Predication

In particular, one has to be careful to remember that in (see f m), the order of arguments is the reverse of what is found in see(m, f).

In order to be slightly more formal about how we are treating the syntax of first-order logic, it is helpful to look first at the **typed lambda calculus**. We will take as our basic types **Ind** and **Bool**, corresponding to the domain of individuals and {*True*, *False*} respectively. We define the set of types recursively. First, every basic type is a type. Second, If σ and τ are types, then (σ → τ) is also a type; this corresponds to the set of functions from things of type σ to things of type τ. We omit the parentheses around σ → τ if there is no ambiguity. For any type τ, we have a set **Var**(τ) of variables of type τ and **Con**(τ) of constants of type τ. We now define the set **Term**(τ) of λ-terms of type τ.

1. **Var**(τ) ⊆ **Term**(τ).
2. **Con**(τ) ⊆ **Term**(τ).
3. If α ∈ **Term**(σ → τ) and β ∈ **Term**(σ), then (α β) ∈ **Term**(τ) (function application).
4. If x ∈ **Var**(σ) and α ∈ **Term**(ϱ), then λx.α ∈ **Term**(τ), where τ = (σ → ϱ) (λ-abstraction).

We replace our earlier definition of formulas containing Boolean connectives (that is, in L_{prop}) by adding the following clause:

5. not ∈ **Con**(**Bool** → **Bool**), and and, or, implies and iff ∈ **Con**(**Bool** → (**Bool** → **Bool**)).

We also add a clause for equality between individual terms.

6. If $\alpha, \beta \in$ **Term**(**Ind**), then $\alpha = \beta \in$ **Term**(**Bool**).

If we return now to NLTK, we can see that our previous implementation of function application already does service for predication. We also note that λ-abstraction can be combined with terms that are conjoined by Boolean operators. For example, the following can be thought of as the property of being an *x* who walks and talks:

```
>>> from nltk_lite.semantics import logic
>>> lp = logic.Parser()
>>> lp.parse(r'\x.((walk x) and (talk x))')
LambdaExpression('x', '(and (walk x) (talk x))')
```

β-conversion can be invoked with the `simplify()` method of `ApplicationExpressions`. As we noted earlier, the"infixify()" method will place binary Boolean connectives in infix position.

```
>>> e = lp.parse(r'(\x.((walk x) and (talk x)) john)')
>>> e
ApplicationExpression('\x.(and (walk x) (talk x))', 'john')
>>> print e.simplify()
(and (walk john) (talk john))
>>> print e.simplify().infixify()
((walk john) and (talk john))
```

Up to this point, we have restricted ourselves to looking at formulas where all the arguments are individual constants (i.e., expressions in **Term**(**Ind**)), corresponding to proper names such as *Jane*, *Mike* and *Fido*. Yet a crucial ingredient of first-order logic is the ability to make general statements involving quantified expressions such as *all dogs* and *some cats*. We turn to this topic in the next section.

11.4.2 Quantification and Scope

First-order logic standardly offers us two quantifiers, *every* (or *all*) and *some*. These are formally written as $\forall$ and , respectively. The following two sets of examples show a simple English example, a logical representation, and the encoding which is accepted by the NLTK `logic` module.

(177a) Every dog barks.

(177b) $\forall$x.((dog x) $\rightarrow$ (bark x))

(177c) `all x.((dog x) implies (bark x))`

(178a) Some cat sleeps.

(178b) x.((cat x) $\wedge$ (sleep x))

(178c) `some x.((cat x) and (sleep x))`

The inclusion of first-order quantifiers motivates the final clause of the definition of our version of first-order logic.

7. If x $\in$ **Var**(**Ind**) and $\varphi \in$ **Term**(**Bool**), then $\forall$x.φ, x.$\varphi \in$ **Term**(**Bool**).

One important property of (177b) often trips people up. The logical rendering in effect says that *if* something is a dog, then it barks, but makes no commitment to the existence of dogs. So in a situation where nothing is a dog, (177b) will still come out true. (Remember that `'(p implies q)'` is true when `'p'` is false.) Now you might argue that (177b) does presuppose the existence of dogs, and that the logic formalization is wrong. But it is possible to find other examples which lack such a presupposition. For instance, we might explain that the value of the Python expression `re.sub('ate', '8', astring)` is the result of replacing all occurrences of `'ate'` in `astring` by `'8'`, even though there may in fact be no such occurrences.

What happens when we want to give a formal representation of a sentence with *two* quantifiers, such as the following?

(179) Every girl chases a dog.

There are (at least) two ways of expressing (179) in FOL:

(180a) $\forall$x.((girl x) $\rightarrow$ y.((dog y) $\wedge$ (chase y x)))

(180b) y.((dog y) $\wedge$ $\forall$x.((every x) $\rightarrow$ (chase y x)))

Can we use both of these? Then answer is Yes, but they have different meanings. (180b) is logically stronger than (180a): it claims that there is a unique dog, say Fido, which is chased by every girl. (180a), on the other hand, just requires that for every girl g, we can find some dog which d chases; but this could be a different dog in each case. We distinguish between (180a) and (180b) in terms of the **scope** of the quantifiers. In the first, $\forall$ has wider scope than , while in (180b), the scope ordering is reversed. So now we have two ways of representing the meaning of (179), and they are both quite legitimate. In other words, we are claiming that (179) is *ambiguous* with respect to quantifier scope, and the formulas in (180) give us a formal means of making the two readings explicit. However, we are not just interested in associating two distinct representations with (179). We also want to show in detail how the two representations lead to different conditions for truth in a formal model. This will be taken up in the next section.

11.4.3 Alphabetic Variants

When carrying out β-reduction, some care has to be taken with variables. Consider, for example, the λ terms (181a) and (181b), which differ only in the identity of a free variable.

(181a) λy.(see x y)

(181b) λy.(see z y)

Suppose now that we apply the λ-term λP.x.(P x) to each of these terms:

(182a) (λP.x.(P x) λy.(see x y))

(182b) (λP.x.(P x) λy.(see z y))

In principle, the results of the application should be semantically equivalent. But if we let the free variable x in (181a) be 'captured' by the existential quantifier in (182b), then after reduction, the results will be different:

(183a) x.(see x x)

(183b) x.(see z x)

(183a) means there is some x that sees him/herself, whereas (183b) means that there is some x that sees an unspecified individual y. What has gone wrong here? Clearly, we want to forbid the kind of variable capture shown in (183a), and it seems that we have been too literal about the label of the particular variable bound by the existential quantifier in the functor expression of (182a). In fact, given any variable-binding expression (involving $\forall$, or λ), the particular name chosen for the bound variable is completely arbitrary. For example, (184a) and (184b) are equivalent; they are called α **equivalents** (or **alphabetic variants**).

(184a) x.(P x)

(184b) z_0.(P z_0)

The process of relabeling bound variables (which takes us from (184a) to (184b)) is known as α-**conversion**. When we test for equality of `VariableBinderExpressions` in the `logic` module (i.e., using ==), we are in fact testing for α-equivalence:

```
>>> from nltk_lite.semantics import *
>>> lp = Parser()
>>> e1 = lp.parse('some x.(P x)')
>>> print e1
some x.(P x)
>>> e2 = e1.alpha_convert(Variable('z'))
>>> print e2
some z.(P z)
>>> e1 == e2
True
```

When β-reduction is carried out on an application (M N), we check whether there are free variables in N which also occur as bound variables in any subterms of M. Suppose, as in the example discussed above, that x is free in N, and that M contains the subterm x.(P x). In this case, we produce an alphabetic variant of x.(P x), say, z.(P z), and then carry on with the reduction. This relabeling is carried out automatically by the β-reduction code in `logic`, and the results can be seen in the following example.

```
>>> e3 = lp.parse('(\P.some x.(P x) \y.(see x y))')
>>> print e3
(\P.some x.(P x) \y.(see x y))
>>> print e3.simplify()
some z2.(see x z2)
```

11.4.4 Types and the Untyped Lambda Calculus

For convenience, let's give a name to language for semantic representations that we are using in `nltk_lite.semantics.logic`: FSRL (for Functional Semantic Representation Language). So far, we have glossed over the fact that the FSRL is based on an implementation of the *untyped* lambda calculus. That is, although we have introduced typing in order to aid exposition, FSRL is not constrained to honour that typing. In particular, there is no formal distinction between predicate expressions and individual expressions; anything can be applied to anything. Indeed, functions can be applied to themselves:

```
>>> lp.parse('(walk walk)')
ApplicationExpression('walk', 'walk')
```

By contrast, most standard approaches to natural language semantics forbid self-application (e.g., applications such as (walk walk)) by adopting a typed language of the kind presented above.

It is also standard to allow constants as basic expressions of the language, as indicated by our use **Con**(τ) is our earlier definitions. Correspondingly, we have used a mixture of convention and supplementary stipulations to bring FSRL closer to this more standard framework for natural language semantics. In particular, we use expressions like `x`, `y`, `z` or `x0`, `x1`, `x2` to indicate individual variables. In FSRL, we assign such strings to the class `IndVariableExpression`.

```
>>> lp.parse('x')
IndVariableExpression('x')
>>> lp.parse('x01')
IndVariableExpression('x01')
```

English-like expressions such as *dog*, *walk* and *john* will be non-logical constants (non-logical in contrast to logical constants such as *not* and *and*). In order to force `logic.Parser()` to recognize non-logical constants, we can initialize the parser with a list of identifiers.

```
>>> lp = Parser(constants=['dog', 'walk', 'see'])
>>> lp.parse('walk')
ConstantExpression('walk')
```

To sum up, while the untyped λ-calculus only recognizes one kind of basic expression other than λ, namely the class of variables (the class `VariableExpression` in `logic`), FSRL adds three further classes of basic expression: `IndVariableExpression`, `ConstantExpression` and `Operator` (Boolean connectives plus the equality relation =).

This completes our discussion of using a first-order language as a basis for semantic representation in NLTK. In the next section, we will study how FSRL is interpreted.

11.5 Formal Semantics

In the preceding sections, we presented some basic ideas about defining a semantic representation FSRL. We also showed informally how expressions of FSRL are paired up with natural language expressions. Later on, we will investigate a more systematic method for carrying out that pairing. But let's suppose for a moment that for any sentence S of English, we have a method of building a corresponding expression of first-order logic that represents the meaning of S (still a fairly distant goal, unfortunately). Would this be enough? Within the tradition of formal semantics, the answer would be No. To be concrete, consider (185a) and (185b).

(185a) Melbourne is an Australian city.

(185b) (((in australia) melbourne) ∧ (city melbourne))

(185a) makes a claim about the world. To know the meaning of (185a), we at least have to know the conditions under which it is true. Translating (185a) into (185b) may clarify some aspects of the structure of (185a), but we can still ask what the meaning of (185b) is. So we want to take the further

[3] When combined with logic, unrestricted self-application leads to Russell's Paradox.

step of giving truth conditions for (185b). To know the conditions under which a sentence is true or false is an essential component of knowing the meaning of that sentence. To be sure, truth conditions do not exhaust meaning. But if we can find some situation in which sentence *A* is true while sentence *B* is false, then we can be certain that *A* and *B* do not have the same meaning.

Now there are infinitely many sentences in **Term**(**Bool**) and consequently it is not possible to simply list the truth conditions. Instead, we give a *recursive* definition of truth. For instance, one of the clauses in the definition might look roughly like this:

(186) $(\varphi \wedge \psi)$ is True iff φ is True and ψ is True.

(186) is applicable to (185b); it allows us to decompose it into its conjuncts, and then proceed further with each of these, until we reach expressions — constants and variables — that cannot be broken down any further.

As we have already seen, all of our non-logical constants are interpreted either as individuals or as curried functions. What we are now going to do is make this notion of interpretation more precise by defining a **valuation** for non-logical constants, building on a set of predefined individuals in a **domain of discourse**. Together, the valuation and domain of discourse make up the main components of a *model* for sentences in our semantic representation language. The framework of model-theoretic semantics provides the tools for making the recursive definition of truth both formally and computationally explicit.

Our models stand in for possible worlds — or ways that the world could actually be. Within these models, we adopt the fiction that our knowledge is completely clearcut: sentences are either true or false, rather than probably true or true to some degree. (The only exception is that there may be expressions which do not receive any interpretation.)

More formally, a **model** for a first-order language *L* is a pair $<D, V>$, where *D* is a domain of discourse and *V* is a valuation function for the non-logical constants of *L*. Non-logical constants are interpreted by *V* as follows (recall that **Ind** is the type of entities and **Bool** is the type of truth values):

- if α is an individual constant, then $V(\alpha) \in D$.
- If γ is an expression of type (**Ind** $\rightarrow$ … (**Ind** $\rightarrow$ **Bool**)…), then $V(\gamma)$ is a function f : $D \rightarrow \ldots (D \rightarrow \{True, False\})\ldots)$.

As explained earlier, expressions of FSRL are not in fact explicitly typed. We leave it to you, the grammar writer, to assign 'sensible' values to expressions rather than enforcing any type-to-denotation consistency.

11.5.1 Characteristic Functions

Within the `semantics` package, curryed characteristic functions are implemented as a subclass of dictionaries, using the `CharFun` constructor.

```
>>> from nltk_lite.semantics import *
>>> cf = CharFun({'d1': CharFun({'d2': True}), 'd2': CharFun({'d1': True})})
```

Values of a `CharFun` are accessed by indexing in the usual way:

```
>>> cf['d1']
{'d2': True}
>>> cf['d1']['d2']
True
```

CharFuns are 'abbreviated' data structures in the sense that they omit key-value pairs of the form (e : False). In fact, they behave just like ordinary dictionaries on keys which are out of their domain, rather than yielding the value False:

```
>>> cf['not in domain']
Traceback (most recent call last):
...
KeyError: 'not in domain'
```

The assignment of False values is delegated to a wrapper method app() of the Model class. app`()` embodies the Closed World assumption; i.e., where ``m is an instance of Model:

```
>>> m.app(cf,'not in domain')
False
```

In practise, it is often more convenient to specify interpretations as n-ary relations (i.e., sets of n-tuples) rather than as n-ary functions. A CharFun object has a read() method which will convert such relations into curried characteristic functions, and a tuples() method which will perform the inverse conversion.

```
>>> s = set([('d1', 'd2'), ('d3', 'd4')])
>>> cf = CharFun()
>>> cf.read(s)
>>> cf
{'d2': {'d1': True}, 'd4': {'d3': True}}
>>> cf.tuples()
set([('d1', 'd2'), ('d3', 'd4')])
```

The function flatten() returns a set of the entities used as keys in a CharFun instance. The same information can be accessed via the domain attribute of CharFun.

```
>>> cf = CharFun({'d1' : {'d2': True}, 'd2' : {'d1': True}})
>>> flatten(cf)
set(['d2', 'd1'])
>>> cf.domain
set(['d2', 'd1'])
```

11.5.2 Valuations

A **Valuation** is a mapping from non-logical constants to appropriate semantic values in the model. Valuations are created using the Valuation constructor.

```
>>> val = Valuation({'Fido': 'd1', 'dog': {'d1': True, 'd2': True}})
>>> val['dog']
{'d2': True, 'd1': True}
>>> val['dog']['d1']
True
```

As with CharFun, an instance of Valuation has a read() method that allows valuations to be specified as relations rather than characteristic functions.

```
>>> setval = [('adam', 'b1'), ('betty', 'g1'),
... ('girl', set(['g2', 'g1'])), ('boy', set(['b1', 'b2'])),
... ('see', set([('b1', 'g1'), ('b2', 'g2'), ('g1', 'b1'), ('g2', 'b1')]))]
>>> val = Valuation()
>>> val.read(setval)
>>> print val
{'adam': 'b1',
'betty': 'g1',
'boy': {'b1': True, 'b2': True},
'girl': {'g2': True, 'g1': True},
'see': {'b1': {'g2': True, 'g1': True},
        'g1': {'b1': True},
        'g2': {'b2': True}}}
```

Valuations have a `domain` attribute, like `CharFun`, and also a `symbols` attribute.

```
>>> val.domain
set(['g1', 'g2', 'b2', 'b1'])
>>> val.symbols
['boy', 'girl', 'see, 'adam', 'betty']
```

11.5.3 Assignments

A variable **Assignment** is a mapping from individual variables to entities in the domain. As indicated earlier, individual variables are written with the letters `'x'`, `'y'`, `'w'` and `'z'`, optionally followed by an integer (e.g., `'x0'`, `'y332'`). Assignments are created using the `Assignment` constructor, which also takes the model's domain of discourse as a parameter.

```
>>> dom = set(['u1', 'u2', 'u3', 'u4'])
>>> g = Assignment(dom, {'x': 'u1', 'y': 'u2'})
>>> g
{'y': 'u2', 'x': 'u1'}
```

In addition, there is a `print()` format for assignments which uses a notation closer to that in logic textbooks:

```
>>> print g
g[u2/y][u1/x]
```

It is possible to update an assignment using the `add()` method; this checks that the variable really is an individual variable, and also checks that the new value belongs to the domain of discourse.

```
>>> dom = set(['u1', 'u2', 'u3', 'u4'])
>>> g = Assignment(dom, {})
>>> g.add('u1', 'x')
{'x': 'u1'}
>>> g.add('u1', 'xyz')
Traceback (most recent call last):
...
AssertionError: Wrong format for an Individual Variable: 'xyz'
>>> g.add('u2', 'x').add('u3', 'y').add('u4', 'x0')
{'y': 'u3', 'x': 'u2', 'x0': 'u4'}
>>> g.add('u5', 'x')
Traceback (most recent call last):
...
AssertionError: u5 is not in the domain set(['u4', 'u1', 'u3', 'u2'])
```

11.5.4 `evaluate()` and `satisfy()`

The `Model` constructor takes two parameters, of type `set` and `Valuation` respectively. Assuming that we have already defined a `Valuation` `val`, it is convenient to use `val`'s domain as the domain for the model constructor.

```
>>> dom = val.domain
>>> m = Model(dom, val)
>>> g = Assignment(dom, {})
```

The top-level method of a `Model` instance is `evaluate()`, which assigns a semantic value to expressions of the `logic` module, under an assignment `g`:

```
>>> m.evaluate('all x. ((boy x) implies (not (girl x)))', g)
True
```

The function `evaluate()` is essentially a convenience for handling expressions whose interpretation yields the `Undefined` value. It then calls the recursive function `satisfy()`. Since most of the interesting work is carried out by `satisfy()`, we shall concentrate on the latter.

The `satisfy()` function needs to deal with the following kinds of expression:

- non-logical constants and variables;
- Boolean connectives;
- function applications;
- quantified formulas;
- lambda-abstracts.

We shall look at each of these in turn.

11.5.5 Evaluating Non-Logical Constants and Variables

When it encounters expressions which cannot be analysed into smaller components, `satisfy()` calls two subsdiary functions. The function `i()` is used to interpret non-logical constants and individual variables, while the variable assignment `g` is used to assign values to individual variables, as seen above.

Any atomic expression which cannot be assigned a value by `i()` or `g` raises an `Undefined` exception; this is caught by `evaluate()`, which returns the string `'Undefined'`. In the following examples, we have set tracing to `2` to give a verbose analysis of the processing steps.

```
>>> m.evaluate('(boy adam)', g, trace=2)
    i, g('boy') = {'b1': True, 'b2': True}
    i, g('adam') = b1
 '(boy adam)': {'b1': True, 'b2': True} applied to b1 yields True
 '(boy adam)' evaluates to True under M, g
 True
 >>> m.evaluate('(girl adam)', g, trace=2)
    i, g('girl') = {'g2': True, 'g1': True}
    i, g('adam') = b1
 '(girl adam)': {'g2': True, 'g1': True} applied to b1 yields False
 '(girl adam)' evaluates to False under M, g
 False
```

```
>>> m.evaluate('(walk adam)', g, trace=2)
   ... checking whether 'walk' is an individual variable
Expression 'walk' can't be evaluated by i and g[b1/x].
'Undefined'
```

11.5.6 Evaluating Boolean Connectives

The `satisfy()` function assigns semantic values to complex expressions according to their syntactic structure, as determined by the method `decompose()`; this calls the parser from the `logic` module to return a 'normalized' parse structure for the expression. In the case of a Boolean connectives, `decompose()` produces a pair consisting of the connective and a list of arguments:

```
>>> m.decompose('((boy adam) and (dog fido))')
('and', ['(boy adam)', '(dog fido)'])
```

Following the functional style of interpretation, Boolean connectives are interpreted quite literally as truth functions; for example, the connective `and` can be interpreted as the function `AND`:

```
>>> AND = {True:  {True: True,
...                 False: False},
...        False: {True: False,
...                 False: False}}
```

We define `OPS` as a mapping between the Boolean connectives and their associated truth functions. Then the simplified clause for the satisfaction of Boolean formulas looks as follows:

```
>>> def satisfy(expr, g):
...   if parsed(expr) == (op, args):
...      if args == (phi, psi):
...         val1 = self.satisfy(phi, g)
...         val2 = self.satisfy(psi, g)
...         return OPS[op][val1][val2]
```

A formula such as `(and p q)` is interpreted by indexing the value of `and` with the values of the two propositional arguments, in the following manner:

```
>>> m.AND[m.evaluate('p', g)][m.evaluate('q', g)]
```

We can use these definitions to generate **truth tables** for the Boolean connectives:

```
>>> from nltk_lite.semantics import Model
>>> ops = ['and', 'or', 'implies', 'iff']
>>> pairs = [(p, q) for p in [True,False] for q in [True,False]]
>>> for o in ops:
...    print "%8s %8s |  p %s q" % ('p', 'q', o)
...    print "-" * 30
...    for (p, q) in pairs:
...        value = Model.OPS[o][p][q]
...        print "%8s %8s | %8s" % (p, q, value)
...    print
...
```

The output is as follows:

```
       p        q |  p and q
-----------------------------
    True     True |     True
    True    False |    False
   False     True |    False
   False    False |    False

       p        q |  p or q
-----------------------------
    True     True |     True
    True    False |     True
   False     True |     True
   False    False |    False

       p        q |  p implies q
-----------------------------
    True     True |     True
    True    False |    False
   False     True |     True
   False    False |     True

       p        q |  p iff q
-----------------------------
    True     True |     True
    True    False |    False
   False     True |    False
   False    False |     True
```

Although these interpretations are close to the informal understanding of the connectives, there are some differences. Thus, `'(p or q)'` is true even when both `'p'` and `'q'` are true. `'(p implies q)'` is true even when `'p'` is false; it only excludes the situation where *'p'* is true and `'q'` is false. `'(p iff q)'` is true if `'p'` and `'q'` have the same truth value, and false otherwise.

11.5.7 Evaluating Function Application

The `satisfy()` clause for function application is similar to that for the connectives. In order to handle type errors, application is delegated to a wrapper function `app()` rather than by directly indexing the curried characteristic function as described earlier. The definition of `satisfy()` started above continues as follows:

```
...  elif parsed(expr) == (fun, arg):
...      funval = self.satisfy(fun, g)
...      argval = self.satisfy(psi, g)
...      return app(funval, argval)
```

11.5.8 Evaluating Quantified Formulas

Let's consider now how to interpret quantified formulas, such as (187).

(187) `some x.(see x betty)`

We decompose (187) into two parts, the quantifier prefix `some x` and the body of the formula, (188).

(188) `(see x betty)`

Although the variable `x` in (187) is **bound** by the quantifier `some`, `x` is not bound by any quantifiers within (188); in other words, it is **free**. A formula containing at least one free variable is said to be **open**. How should open formulas be interpreted? We can think of `x` as being similar to a variable in Python, in the sense that we cannot evaluate an expression containing a variable unless it has already been assigned a value. As mentioned earlier, the task of assigning values to individual variables is undertaken by an `Assignment` object `g`. However, our variable assignments are partial: `g` may well not give a value to `x`.

```
>>> dom = val.domain
>>> g = Assignment(dom)
>>> m.evaluate('(see x betty)', g)
'Undefined'
```

We can use the `add()` method to explicitly add a binding to an assignment, and thereby ensure that `g` gives `x` a value.

```
>>> g.add('b1', 'x')
{'x': 'b1'}
>>> m.evaluate('(see x betty)', g)
True
```

In a case like this, we say that the entity `b1` **satisfies** the open formula `(see x betty)`, or that `(see x betty)` is **satisfied under the assignment** `g['b1'/'x']`.

When we interpret a quantified formula, we depend on the notion of an open subformula being satisfied under a variable assignment. However, to capture the force of the quantifier, we need to abstract away from arbitrary specific assignments. The first step is to define the set of **satisfiers** of a formula that is open in some variable. Formally, given an open formula φ[x] dependent on x and a model with domain D, we define the set *sat*(φ[x], g) of **satisfiers** of φ[x] to be:

(189) $\{u \in D \mid satisfy(\varphi[x], g[u/x]) = True\}$

We use $g[u/x]$ to mean that assignment which is just like g except that $g(x) = u$. Here is a Python definition of `satisfiers()`:

```
>>> def satisfiers(expr, var, g):
...     candidates = []
...     if freevar(var, expr):
...         for u in domain:
...             g.add(u, var)
...             if satisfy(expr, g):
...                 candidates.append(u)
...     return set(candidates)
```

The satisfiers of an arbitrary open formula can be inspected using the `satisfiers()` method.

```
>>> m.satisfiers('some y.((girl y) and (see x y))', 'x', g)
set(['b1'])
>>> m.satisfiers('some y.((girl y) and (see y x))', 'x', g)
set(['b1', 'b2'])
>>> m.satisfiers('(((girl x) and (boy x)) or (dog x))', 'x', g)
set(['d1'])
>>> m.satisfiers('((girl x) and ((boy x) or (dog x)))', 'x', g)
set([])
```

Now that we have put the notion of satisfiers in place, we can use this to determine a truth value for quantified expressions. An existentially quantified formula x.φ[x] is held to be true if and only if $sat(\varphi[x], g)$ is nonempty. We use the length function `len()` to return the cardinality of a set.

```
...    elif parsed(expr) == (binder, body):
...       if binder = ('some', var):
...          sat = self.satisfiers(body, var, g)
...          return len(sat) > 0
```

In other words, a formula x.φ[x] has the same value in model M as the statement that the number of satisfiers in M of φ[x] is greater than 0.

A universally quantified formula $\forall$x.φ[x] is held to be true if and only if every u in the model's domain D belongs to $sat(\varphi[x], g)$; equivalently, if D $\subseteq$ $sat(\varphi[x], g)$. The `satisfy()` clause above for existentials can therefore be extended with the clause:

```
...      elif binder == ('all', var):
...          sat = self.satisfiers(body,var,g)
...          return domain.issubset(sat)
```

Although our approach to interpreting quantified formulas has the advantage of being transparent and conformant to classical logic, it is not computationally efficient. To verify an existentially quantified formula, it suffices to find just one satisfying individual and then return `True`. But the method just presented requires us to test satisfaction for every individual in the domain of discourse for each quantifier. This requires m^n evaluations, where m is the cardinality of the domain and n is the number of nested quantifiers.

11.5.9 Evaluating lambda abstracts

Finally, we can also evaluate λ-abstracts; not surprisingly, these are interpreted as `CharFuns`. To illustrate, we can construct the binary relation of individuals who see each other, or the ternary relation of distinct individuals a and b such for some c, a sees c and c sees b.

```
>>> m.evaluate(r'\x y. ((see x y) and (see y x))', g)
{'b1': {'g1': True}, 'g1': {'b1': True}}
>>> r = m.evaluate(r"""\x z y. (((see x z) and (see z y))
...                             and (not (x = y)))""", g)
>>> r.tuples()
set([('g2', 'b1', 'g1'), ('b2', 'g2', 'b1')])
```

Note that λ-abstracts can only be explicitly evaluated when the bound variable is an individual variable. Variables which range over functions, such as the `'P'` in `'\x.  (P suzie)'`, are called **higher-order** variables, and quantification over higher-order variables lies outside first-order logic.

If you attempt to evaluate an expression such as `'\x.  (P suzie)'`, the `semantics` package will raise an error. Since we only allow ourselves to quantify over individuals in FSRL, a variable assignment only give values to individual variables, and variable assignment is crucial for interpreting λ-abstraction. So though we do allow abstracts with higher-order variables in the language, they are not 'first-class citizens': they are only used as a stepping stone on the way to building up semantic representations in a compositional manner, and are eliminated prior to evaluation by β-reduction.

11.5.10 Exercises

1. ☼ Define a denotation for exclusive `or` (i.e., `'(p xor q)'` is equivalent to `'((p or q) and (not (p and q)))'`.)

2. ☼ Evaluate the expressions `'\x.(boy adam)'` and `'\x.(boy fido)'` in the model given above. Explain your results.

3. ◑ Use the `satisfiers()` method for determing the set of satisfiers of the open formula `'((dog x) implies (x = fido))'` in the model given above. Explain why the result is the way that it is.

4. ◑ Develop a set of around 10 sentences, using FSRL. Build a model for the sentences which makes then all true, and verify the results.

5. ◑ Build a model for a relation `rel` which is **transitively closed** and **reflexive**. That is, it satisfies the following two sentences:

   a.) `all x y z.(((rel y x) and (rel z y)) implies (rel z x))`

   b.) `all x.(rel x x)`

11.6 Quantifier Scope Revisited

You may recall that we discussed earlier an example of quantifier scope ambiguity, repeated here as (190).

(190) Every girl chases a dog.

The two readings are represented as follows.

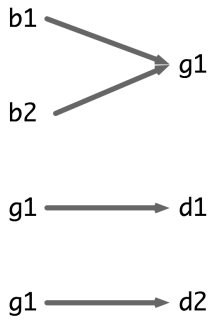
```
>>> sr1 = 'all x.((girl x) implies some z.((dog z) and (chase z x)))'
>>> sr2 = 'some z.((dog z) and all x.((girl x) implies (chase z x)))'
```

In order to examine the ambiguity more closely, let's fix our valuation as follows:

```
>>> val = Valuation()
>>> v = [('john', 'b1'),
... ('mary', 'g1'),
... ('suzie', 'g2'),
... ('fido', 'd1'),
... ('tess', 'd2'),
... ('noosa', 'n'),
... ('girl', set(['g1', 'g2'])),
... ('boy', set(['b1', 'b2'])),
... ('dog', set(['d1', 'd2'])),
... ('bark', set(['d1', 'd2'])),
... ('walk', set(['b1', 'g2', 'd1'])),
... ('chase', set([('b1', 'g1'), ('b2', 'g1'), ('g1', 'd1'), ('g2', 'd2')])),
... ('see', set([('b1', 'g1'), ('b2', 'd2'), ('g1', 'b1'),
... ('d2', 'b1'), ('g2', 'n')])),
... ('in', set([('b1', 'n'), ('b2', 'n'), ('d2', 'n')])),
... ('with', set([('b1', 'g1'), ('g1', 'b1'), ('d1', 'b1'), ('b1', 'd1')]))]
>>> val.read(v)
```

Using a slightly different graph from before, we can also visualise the chase relation as in (191).

(191)



In (191), an arrow between two individuals *x* and *y* indicates that *x* chases *y*. So `b1` and `b2` both chase `g1`, while `g1` chases `d1` and `g2` chases `d2`. In this model, formula `sr1` above is true but `sr2` is false. One way of exploring these results is by using the `satisfiers()` method of `Model` objects.

```
>>> dom = val.domain
>>> m = Model(dom, val)
>>> g = Assignment(dom)
>>> fmla1 = '((girl x) implies some y.((dog y) and (chase y x)))'
>>> m.satisfiers(fmla1, 'x', g)
set(['g2', 'g1', 'n', 'b1', 'b2', 'd2', 'd1'])
>>>
```

This gives us the set of individuals that can be assigned as the value of `x` in `fmla1`. In particular, every girl is included in this set. By contrast, consider the formula `fmla2` below; this has no satisfiers for the variable `y`.

```
>>> fmla2 = '((dog y) and all x.((girl x) implies (chase y x)))'
>>> m.satisfiers(fmla2, 'y', g)
set([])
>>>
```

That is, there is no dog that is chased by both `g1` and `g2`. Taking a slightly different open formula, `fmla3`, we can verify that there is a girl, namely `g1`, who is chased by every boy.

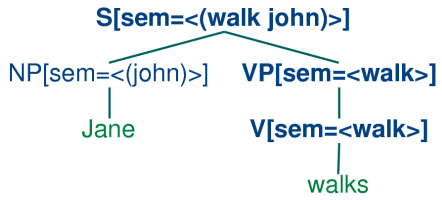
```
>>> fmla3 = '((girl y) and all x.((boy x) implies (chase y x)))'
>>> m.satisfiers(fmla3, 'y', g)
set(['g1'])
>>>
```

11.7 Evaluating English Sentences

11.7.1 Using the `sem` feature

Until now, we have taken for granted that we have some appropriate logical formulas to interpret. However, ideally we would like to derive these formulas from natural language input. One relatively easy way of achieving this goal is to build on the grammar framework developed in Chapter 9. Our first step is to introduce a new feature, `sem`. Because values of `sem` generally need to be treated differently from other feature values, we use the convention of enclosing them in angle brackets. (192) illustrates a first approximation to the kind of analyses we would like to build.

(192)



Thus, the `sem` value at the root node shows a semantic representation for the whole sentence, while the `sem` values at lower nodes show semantic representations for constituents of the sentence. So far, so good, but how do we write grammar rules which will give us this kind of result? To be more specific, suppose we have a NP and VP constituents with appropriate values for their `sem` nodes? If you reflect on the machinery that was introduced in discussing the λ calculus, you might guess that function application will be central to composing semantic values. You will also remember that our feature-based grammar framework gives us the means to refer to *variable* values. Putting this together, we can postulate a rule like (193) for building the `sem` value of an S. (Observe that in the case where the value of `sem` is a variable, we omit the angle brackets.)

(193)
```
S[sem = <app(?vp,?subj)>] -> NP[sem=?subj] VP[sem=?vp]
```

(193) tells us that given some `sem` value `?subj` for the subject NP and some `sem` value `?vp` for the VP, the `sem` value of the S mother is constructed by applying `?vp` as a functor to `?np`. From this, we can conclude that `?vp` has to denote a function which has the denotation of `?np` in its domain; in fact, we are going to assume that `?vp` denotes a curryed characteristic function on individuals. (193) is a nice example of building semantics using **the principle of compositionality**: that is, the principle that the semantics of a complex expression is a function of the semantics of its parts.

To complete the grammar is very straightford; all we require are the rules shown in (194).

(194)
```
VP[sem=?v] -> IV[sem=?v]
NP[sem=<john>] -> 'Jane'
IV[sem=<walk>] -> 'walks'
```

The VP rule says that the mother's semantics is the same as the head daughter's. The two lexical rules just introduce non-logical constants to serve as the semantic values of *Jane* and *walks* respectively. This grammar can be parsed using the chart parser in `nltk_lite.parse.featurechart`, and the trace in (195) shows how semantic values are derived by feature unification in the process of building a parse tree.

(195)
```
Predictor |> . .| S[sem='(?vp ?subj)'] -> * NP[sem=?subj] VP[sem=?vp]
Scanner   |[-] .| [0:1] 'Jane'
Completer |[-> .| S[sem='(?vp john)'] -> NP[sem='john'] * VP[sem=?vp]
Predictor |. > .| VP[sem=?v] -> * IV[sem=?v]
Scanner   |. [-]| [1:2] 'walks'
Completer |. [-]| VP[sem='walk'] -> IV[sem='walk'] *
Completer |[===]| S[sem='(walk john)'] -> NP[sem='john'] VP[sem='walk'] *
Completer |[===]| [INIT] -> S *
```

11.7.2 Quantified NPs

You might be thinking this is all too easy — surely there is a bit more to building compositional semantics. What about quantifiers, for instance? Right, this is a crucial issue. For example, we want (196a) to be given a semantic representation like (196b). How can this be accomplished?

(196a) A dog barks.

(196b) `'some x.((dog x) and (bark x))'`

Let's make the assumption that our *only* operation for building complex semantic representations is `'app()'` (corresponding to function application). Then our problem is this: how do we give a semantic representation to quantified NPs such as *a dog* so that they can be combined with something like `'walk'` to give a result like (196b)? As a first step, let's make the subject's `sem` value act as the functor rather than the argument in `'app()'`. Now we are looking for way of instantiating `?np` so that (197a) is equivalent to (197b).

(197a) [sem=<app(?np, bark)>]

(197b) [sem=<some x.((dog x) and (bark x))>]

This is where λ abstraction comes to the rescue; doesn't (197) look a bit reminiscent of carrying out β-reduction in the λ-calculus? In other words, we want a λ term M to replace `'?np'` so that applying M to `'bark'` yields (196b). To do this, we replace the occurence of `'bark'` in (196b) by a variable `'P'`, and bind the variable with λ, as shown in (198).

(198) `'\P.some x.((dog x) and (P x))'`

As a point of interest, we have used a different style of variable in (198), that is `'P'` rather than `'x'` or `'y'`. This is to signal that we are abstracting over a different kind of thing — not an individual, but a function from **Ind** to **Bool**. So the type of (198) as a whole is ((**Ind** → **Bool**) → **Bool**). We will take this to be the type of NPs in general. To illustrate further, a universally quantified NP will look like (199).

(199) `'\P.all x.((dog x) implies (P x))'`

We are pretty much done now, except that we also want to carry out a further abstraction plus application for the process of combining the semantics of the determiner *a* with the semantics of *dog*. Applying (198) as a functor to `'bark'` gives us `'(\P.some x.((dog x) and (P x)) bark)'`, and carrying out β-reduction yields just what we wanted, namely (196b).

NLTK provides some utilities to make it easier to derive and inspect semantic interpretations. `text_interpret()` is intended for batch interpretation of a list of input sentences. It builds a dictionary `d` where for each sentence `sent` in the input, `d[sent]` is a list of paired trees and semantic representations for `sent`. The value is a list, since `sent` may be syntactically ambiguous; in the following example, we just look at the first member of the list.

```
>>> from nltk_lite.semantics import *
>>> grammar = GrammarFile.read_file('sem1.cfg')
>>> result  = text_interpret(['a dog barks'], grammar, beta_reduce=0)
>>> (syntree, semrep) = result['a dog barks'][0]
>>> print syntree
([INIT][]:
  (S[ sem = ApplicationExpression('(\Q P.some x.(and (Q x) (P x)) dog)', 'bark') ]:
    (NP[ sem = ApplicationExpression('\Q P.some x.(and (Q x) (P x))', 'dog') ]:
      (Det[ sem = LambdaExpression('Q', '\P.some x.(and (Q x) (P x))') ]: 'a')
      (N[ sem = VariableExpression('dog') ]: 'dog'))
    (VP[ sem = VariableExpression('bark') ]:
```

```
      (IV[ sem = VariableExpression('bark') ]: 'barks'))))
>>> print semrep
some x.(and (dog x) (bark x))
>>>
```

By default, the semantic representation that is produced by `text_interpret()` has already undergone β-reduction, but in the above example, we have overriden this. Subsequent reduction is possible using the `simplify()` method, and Boolean connectives can be placed in infix position with the `infixify()` method.

```
>>> print semrep.simplify()
some x.(and (dog x) (bark x))
>>> print semrep.simplify().infixify()
some x.((dog x) and (bark x))
```

11.7.3 Transitive Verbs

Our next challenge is to deal with sentences containing transitive verbs, such as (200).

(200) Suzie chases a dog.

The output semantics that we want to build is shown in (201).

(201) `'some x.((dog x) and (chase x suzie))'`

Let's look at how we can use λ-abstraction to get this result. A significant constraint on possible solutions is to require that the semantic representation of *a dog* be independent of whether the NP acts as subject or object of the sentence. In other words, we want to get (201) as our output while sticking to (198) as the NP semantics. A second constraint is that VPs should have a uniform type of interpretation regardless of whether they consist of just an intransitive verb or a transitive verb plus object. More specifically, we stipulate that VPs always denote characteristic functions on individuals. Given these constraints, here's a semantic representation for *chases a dog* which does the trick.

(202) `'\y.some x.((dog x) and (chase x y))'`

Think of (202) as the property of being a *y* such that for some dog *x*, *y* chases *x*; or more colloquially, being a *y* who chases a dog. Our task now resolves to designing a semantic representation for *chases* which can combine via `app` with (198) so as to allow (202) to be derived.

Let's carry out a kind of inverse β-reduction on (202), giving rise to (203).

Let Then we are part way to the solution if we can derive (203), where `'X'` is applied to `'\z.(chase z y)'`.

(203) `'(\P.some x.((dog x) and (P x)) \z.(chase z y))'`

(203) may be slightly hard to read at first; you need to see that it involves applying the quantified NP representation from (198) to `'\z.(chase z y))'`. (203) is of course equivalent to (202).

Now let's replace the functor in (203) by a variable `'X'` of the same type as an NP; that is, of type ((**Ind** → **Bool**) → **Bool**).

(204) `'(X \z.(chase z y))'`

The representation of a transitive verb will have to apply to an argument of the type of `'X'` to yield a functor of the type of VPs, that is, of type (**Ind → Bool**). We can ensure this by abstracting over both the `'X'` variable in (204) and also the subject variable `'y'`. So the full solution is reached by giving *chases* the semantic representation shown in (205).

(205)
```
'\X y.(X \x.(chase x y))'
```

If (205) is applied to (198), the result after β-reduction is equivalent to (202), which is what we wanted all along:

(206)
```
'(\X y.(X \x.(chase x y)) \P.some x.((dog x) and (P x))'

'(\y.(\P.some x.((dog x) and (P x)) \x.(chase x y))'

'\y.(some x.((dog x) and (chase x y)))'
```

In order to build a semantic representation for a sentence, we also need to combine in the semantics of the subject NP. If the latter is a quantified expression like *every girl*, everything proceeds in the same way as we showed for *a dog barks* earlier on; the subject is translated as a functor which is applied to the semantic representation of the VP. However, we now seem to have created another problem for ourselves with proper names. So far, these have been treated semanntically as individual constants, and these cannot be applied as functors to expressions like (202). Consequently, we need to come up with a different semantic representation for them. What we do in this case is re-interpret proper names so that they too are functors, like quantified NPs. (207) shows the required λ expression for *Suzie*.

(207)
```
'\P.(P suzie)'
```

(207) denotes the characteristic function corresponding to the set of all properties which are true of Suzie. Converting from an individual constant to an expression like (205) is known as **type raising**, and allows us to flip functors with arguments. That is, type raising means that we can replace a Boolean-valued application such as (f a) with an equivalent application (λP.(P a) f).

One important limitation of the approach we have presented here is that it does not attempt to deal with scope ambiguity. Instead, quantifier scope ordering directly reflects scope in the parse tree. As a result, a sentence like (179), repeated here, will always be translated as (209a), not (209b).

(208) Every girl chases a dog.

(209a)
```
'all x.((girl x) implies some y.  ((dog y) and (chase y x)))'
```

(209b)
```
'some y.  (dog y) and all x.  ((girl x) implies (chase y x)))'
```

This limitation can be overcome, for example using the hole semantics described in [Blackburn and Bos, 2005], but discussing the details would take us outside the scope of the current chapter.

Now that we have looked at some slightly more complex constructions, we can evaluate them in a model. In the following example, we derive two parses for the sentence *every boy chases a girl in Noosa*, and evaluate each of the corresponding semantic representations in the model `model0.py` which we have imported.

```
>>> from nltk_lite.semantics import *
>>> from model0.py import *
>>> grammar = GrammarFile.read_file('sem2.cfg')
>>> sent = 'every boy chases a girl in Noosa'
>>> result = text_evaluate([sent], grammar, m, g)
>>> for (syntree, semrep, value) in result[sent]:
...  print "'%s' is %s in Model m\n" % (semrep.infixify(), value)
...
'all x.((boy x) implies (some z4.((girl z4) and (chase z4 x)) and
(in noosa x)))' is True in Model m
...
'all x.((boy x) implies some z5.(((girl z5) and (in noosa z5)) and
(chase z5 x)))' is False in Model m
```

11.8 Case Study: Extracting Valuations from Chat-80

Building `Valuation` objects by hand becomes rather tedious once we consider larger examples. This raises the question of whether the relation data in a `Valuation` could be extracted from some pre-existing source. The `chat80` module in `ntlk_lite.corpora` provides an example of extracting data from the Chat-80 Prolog knowledge base (which included as part of the NLTK `corpora` distribution).

Chat-80 data is organized into collections of clauses, where each collection functions as a table in a relational database. The predicate of the clause provides the name of the table; the first element of the tuple acts as the 'key'; and subsequent elements are further columns in the table.

In general, the name of the table provides a label for a unary relation whose extension is all the keys. For example, the table in `cities.pl` contains triples such as (210).

(210) `'city(athens,greece,1368).'`

Here, `'athens'` is the key, and will be mapped to a member of the unary relation city.

The other two columns in the table are mapped to binary relations, where the first argument of the relation is filled by the table key, and the second argument is filled by the data in the relevant column. Thus, in the `city` table illustrated by the tuple in (210), the data from the third column is extracted into a binary predicate population_of, whose extension is a set of pairs such as `'(athens, 1368)'`.

In order to encapsulate the results of the extraction, a class of `Concepts` is introduced. A `Concept` object has a number of attributes, in particular a `prefLabel` and `extension`, which make it easier to inspect the output of the extraction. The `extension` of a `Concept` object is incorporated into a `Valuation` object.

As well as deriving unary and binary relations from the Chat-80 data, we also create a set of individual constants, one for each entity in the domain. The individual constants are string-identical to the entities. For example, given a data item such as `'zloty'`, we add to the valuation a pair `('zloty', 'zloty')`. In order to parse English sentences that refer to these entities, we also create a lexical item such as the following for each individual constant:

(211) `PropN[num=sg, sem=<\P.(P zloty)>] -> 'Zloty'`

The `chat80` module can be found in the `corpora` package. The attribute `chat80.items` gives us a list of Chat-80 relations:

```
>>> from nltk_lite.corpora import chat80
>>> chat80.items
['borders', 'contains', 'city', 'country', 'circle_of_lat',
'circle_of_long', 'continent', 'region', 'ocean', 'sea']
```

The `concepts()` method shows the list of `Concepts` that can be extracted from a `chat80` relation, and we can then inspect their extensions.

```
>>> concepts = chat80.concepts('city')
>>> concepts
[Concept('city'), Concept('country_of'), Concept('population_of')]
>>> rel = concepts[1].extension
>>> list(rel)[:5]
[('chungking', 'china'), ('karachi', 'pakistan'),
('singapore_city', 'singapore'), ('athens', 'greece'),
('birmingham', 'united_kingdom')]
```

In order to convert such an extension into a valuation, we use the `make_valuation()` method; setting `read=True` creates and returns a new `Valuation` object which contains the results.

```
>>> val = chat80.make_valuation(concepts, read=True)
>>> val['city']['calcutta']
True
>>> val['country_of']['india']
{'hyderabad': True, 'delhi': True, 'bombay': True,
'madras': True, 'calcutta': True}
>>> from nltk_lite.semantics import *
>>> g = Assignment(dom)
>>> m = Model(dom, val)
>>> m.evaluate(r'\x . (population_of x jakarta)', g)
{'533': True}
```

> **Note**
>
> Population figures are given in thousands. Bear in mind that the geographical data used in these examples dates back at least to the 1980s, and was already somewhat out of date at the point when [Warren and Pereira, 1982] was published.

11.9 Summary

- Semantic Representations (SRs) for English are constructed using a language based on the λ-calculus, together with Boolean connectives, equality, and first-order quantifiers.
- β-reduction in the λ-calculus corresponds semantically to application of a function to an argument. Syntactically, it involves replacing a variable bound by λ in the functor with the expression that provides the argument in the function application.
- If two λ-abstracts differ only in the label of the variable bound by λ, they are said to be α equivalents. Relabeling a variable bound by a λ is called α-conversion.
- Currying of a binary function turns it into a unary function whose value is again a unary function.

- FSRL has both a syntax and a semantics. The semantics is determined by recursively evaluating expressions in a model.
- A key part of constructing a model lies in building a valuation which assigns interpretations to non-logical constants. These are interpreted as either curried characteristic functions or as individual constants.
- The interpretation of Boolean connectives is handled by the model; these are interpreted as characteristic functions.
- An open expression is an expression containing one or more free variables. Open expressions only receive an interpretation when their free variables receive values from a variable assignment.
- Quantifiers are interpreted by constructing, for a formula φ[x] open in variable x, the set of individuals which make φ[x] true when an assignment g assigns them as the value of x. The quantifier then places constraints on that set.
- A closed expression is one that has no free variables; that is, the variables are all bound. A closed sentence is true or false with respect to all variable assignments.
- Given a formula with two nested quantifiers Q_1 and Q_2, the outermost quantifier Q_1 is said to have wide scope (or scope over Q_2). English sentences are frequently ambiguous with respect to the scope of the quantifiers they contain.
- English sentences can be associated with an SR by treating `sem` as a feature. The `sem` value of a complex expressions typically involves functional application of the `sem` values of the component expressions.
- Model valuations need not be built by hand, but can also be extracted from relational tables, as in the Chat-80 example.

11.10 Exercises

1. ◑ Modify the `nltk_lite.semantics.evaluate` code so that it will give a helpful error message if an expression is not in the domain of a model's valuation function.
2. ★ Specify and implement a typed functional language with quantifiers, Boolean connectives and equality. Modify `nltk_lite.semantics.evaluate` to interpret expressions of this language.
3. ★ Extend the `chat80` code so that it will extract data from a relational database using SQL queries.
4. ★ Taking [WarrenPereira1982] as a starting point, develop a technique for converting a natural language query into a form that can be evaluated more efficiently in a model. For example, given a query of the form `'((P x) and (Q x)'`, convert it to `'((Q x) and (P x)'` if the extension of `'Q'` is smaller than the extension of `'P'`.

11.11 Further Reading

The use of characteristic functions for interpreting expressions of natural language was primarily due to Richard Montague. [Dowty et al., 1981] gives a comprehensive and reasonably approachable introduction to Montague's grammatical framework.

A more recent and wide-reaching study of the use of a λ based approach to natural language can be found in [Carpenter, 1997].

[Heim and Kratzer, 1998] is a thorough application of formal semantics to transformational grammars in the Government-Binding model.

[Blackburn and Bos, 2005] is the first textbook devoted to computational semantics, and provides an excellent introduction to the area.

Chapter 12

Language Engineering

This chapter will cover evaluation, trade-offs between methods that create large vs small models (e.g. n-gram tagging vs Brill tagging).

12.1 Problems with Tagging

- what context is relevant?
- how best to combine taggers?
- sensitivity to lexical identity?
- ngram tagging produces large models, uninterpretable cf Brill tagging, which has smaller, linguistically-interpretable models

12.1.1 Exercises

1. ★ **Tagger context**:

   N-gram taggers choose a tag for a token based on its text and the tags of the *n-1* preceding tokens. This is a common context to use for tagging, but certainly not the only possible context.

   a) Construct a new tagger, sub-classed from `SequentialTagger`, that uses a different context. If your tagger's context contains multiple elements, then you should combine them in a tuple. Some possibilities for elements to include are: (i) the current word or a previous word; (ii) the length of the current word text or of the previous word; (iii) the first letter of the current word or the previous word; or (iv) the previous tag. Try to choose context elements that you believe will help the tagger decide which tag is appropriate. Keep in mind the trade-off between more specific taggers with accurate results; and more general taggers with broader coverage. Combine your tagger with other taggers using the backoff method.

   b) How does the combined tagger's accuracy compare to the basic tagger?

   c) How does the combined tagger's accuracy compare to the combined taggers you created in the previous exercise?

2. ★ **Reverse sequential taggers** Since sequential taggers tag tokens in order, one at a time, they can only use the predicted tags to the *left* of the current token to decide what tag to assign to a token. But in some cases, the *right* context may provide more useful information than the left context. A reverse sequential tagger starts with the last word of the sentence and, proceeding in right-to-left order, assigns tags to words on the basis of the tags it has already predicted to the right. By reversing texts at appropriate times, we can use NLTK's existing sequential tagging classes to perform reverse sequential tagging: reverse the training text before training the tagger; and reverse the text being tagged both before and after.

   a) Use this technique to create a bigram reverse sequential tagger.

   b) Measure its accuracy on a tagged section of the Brown corpus. Be sure to use a different section of the corpus for testing than you used for training.

   c) How does its accuracy compare to a left-to-right bigram tagger, using the same training data and test data?

3. ★ **Alternatives to backoff**: Create a new kind of tagger that combines several taggers using a new mechanism other than backoff (e.g. voting). For robustness in the face of unknown words, include a regexp tagger, a unigram tagger that removes a small number of prefix or suffix characters until it recognizes a word, or an n-gram tagger that does not consider the text of the token being tagged.

12.2 Evaluating Taggers

As we experiment with different taggers, it is important to have an objective performance measure. Fortunately, we already have manually verified training data (the original tagged corpus), so we can use that to score the accuracy of a tagger, and to perform systematic error analysis.

12.2.1 Scoring Accuracy

Consider the sentence from the Brown Corpus in Table 12.1. The 'Gold Standard' tags from the corpus are given in the second column, while the tags assigned by a unigram tagger appear in the third column. Two mistakes made by the unigram tagger are italicized.

Sentence	Gold Standard	Unigram Tagger
The	at	at
President	nn-tl	nn-tl
said	vbd	vbd
he	pps	pps
will	md	md
ask	vb	vb
Congress	np	np
to	to	to
increase	vb	*nn*
grants	nns	nns

Sentence	Gold Standard	Unigram Tagger
to	in	*to*
states	nns	nns
for	in	in
vocational	jj	jj
rehabilitation	nn	nn
.	.	.

Table 12.1: Evaluating Taggers

The tagger correctly tagged 14 out of 16 words, so it gets a score of 14/16, or 87.5%. Of course, accuracy should be judged on the basis of a larger sample of data. NLTK provides a function called `tag.accuracy` to automate the task. In the simplest case, we can test the tagger using the same data it was trained on:

```
>>> from nltk_lite import tag
>>> from nltk_lite.corpora import brown
>>> from itertools import islice
>>> train_sents = list(islice(brown.tagged(), 500))  # sents 0..499
>>> unigram_tagger = tag.Unigram()
>>> unigram_tagger.train(train_sents)
>>> acc = tag.accuracy(unigram_tagger, train_sents)
>>> print 'Accuracy = %4.1f%%' % (100 * acc)
Accuracy = 79.6%
```

However, testing a language processing system over its training data is unwise. A system which simply memorized the training data would get a perfect score without doing any linguistic modeling. Instead, we would like to reward systems that make good generalizations, so we should test against *unseen data*, and replace `train_sents` above with `unseen_sents`. We can then define the two sets of data as follows:

```
>>> train_sents  = list(brown.tagged('a'))[:500]
>>> unseen_sents = list(brown.tagged('a'))[500:600] # sents 500-599
```

Now we train the tagger using `train_sents` and evaluate it using `unseen_sents`, as follows:

```
>>> unigram_tagger = tag.Unigram(backoff=tag.Default('nn'))
>>> unigram_tagger.train(train_sents)
>>> acc = tag.accuracy(unigram_tagger, unseen_sents)
>>> print 'Accuracy = %4.1f%%' % (100 * acc)
Accuracy = 74.6%
```

The accuracy scores produced by this evaluation method are lower, but they give a more realistic picture of the performance of the tagger. Note that the performance of any statistical tagger is highly dependent on the quality of its training set. In particular, if the training set is too small, it will not be able to reliably estimate the most likely tag for each word. Performance will also suffer if the training set is significantly different from the texts we wish to tag.

In the process of developing a tagger, we can use the accuracy score as an objective measure of the improvements made to the system. Initially, the accuracy score will go up quickly as we fix obvious shortcomings of the tagger. After a while, however, it becomes more difficult and improvements are small.

12.2.2 Baseline Performance

It is difficult to interpret an accuracy score in isolation. For example, is a person who scores 25% in a test likely to know a quarter of the course material? If the test is made up of 4-way multiple choice questions, then this person has not performed any better than chance. Thus, it is clear that we should *interpret* an accuracy score relative to a *baseline*. The choice of baseline is somewhat arbitrary, but it usually corresponds to minimal knowledge about the domain.

In the case of tagging, a possible baseline score can be found by tagging every word with `NN`, the most likely tag.

```
>>> baseline_tagger = tag.Default('nn')
>>> acc = tag.accuracy(baseline_tagger, brown.tagged('a'))
>>> print 'Accuracy = %4.1f%%' % (100 * acc)
Accuracy = 13.1%
```

Unfortunately this is not a very good baseline. There are many high-frequency words which are not nouns. Instead we could use the standard unigram tagger to get a baseline of 75%. However, this does not seem fully legitimate: the unigram's model covers all words seen during training, which hardly seems like 'minimal knowledge'. Instead, let's only permit ourselves to store tags for the most frequent words.

The first step is to identify the most frequent words in the corpus, and for each of these words, identify the most likely tag:

```
>>> from nltk_lite.probability import *
>>> wordcounts = FreqDist()
>>> wordtags = ConditionalFreqDist()
>>> for sent in brown.tagged('a'):
...     for (w,t) in sent:
...         wordcounts.inc(w)     # count the word
...         wordtags[w].inc(t)    # count the word's tag
>>> frequent_words = wordcounts.sorted_samples()[:1000]
```

Now we can create a lookup table (a dictionary) which maps words to likely tags, just for these high-frequency words. We can then define a new baseline tagger which uses this lookup table:

```
>>> table = dict((word, wordtags[word].max()) for word in frequent_words)
>>> baseline_tagger = tag.Lookup(table, tag.Default('nn'))
>>> acc = tag.accuracy(baseline_tagger, brown.tagged('a'))
>>> print 'Accuracy = %4.1f%%' % (100 * acc)
Accuracy = 72.5%
```

This, then, would seem to be a reasonable baseline score for a tagger. When we build new taggers, we will only credit ourselves for performance exceeding this baseline.

12.2.3 Error Analysis

While the accuracy score is certainly useful, it does not tell us how to improve the tagger. For this we need to undertake error analysis. For instance, we could construct a *confusion matrix*, with a row and a column for every possible tag, and entries that record how often a word with tag T_i is incorrectly tagged as T_j Another approach is to analyze the context of the errors, which is what we do now.

Consider the following program, which catalogs all errors along with the tag on the left and their frequency of occurrence.

```
>>> errors = {}
>>> for i in range(len(unseen_sents)):
...     raw_sent = tag.untag(unseen_sents[i])
...     test_sent = list(unigram_tagger.tag(raw_sent))
...     unseen_sent = unseen_sents[i]
...     for j in range(len(test_sent)):
...         if test_sent[j][1] != unseen_sent[j][1]:
...             test_context = test_sent[j-1:j+1]
...             gold_context = unseen_sent[j-1:j+1]
...             if None not in test_context:
...                 pair = (tuple(test_context), tuple(gold_context))
...                 if pair not in errors:
...                     errors[pair] = 0
...                 errors[pair] += 1
```

The `errors` dictionary has keys of the form `((t1,t2),(g1,g2))`, where `(t1,t2)` are the test tags, and `(g1,g2)` are the gold-standard tags. The values in the `errors` dictionary are simple counts of how often each error occurred. With some further processing, we construct the list `counted_errors` containing tuples consisting of counts and errors, and then do a reverse sort to get the most significant errors first:

```
>>> counted_errors = [(errors[k], k) for k in errors.keys()]
>>> counted_errors.sort()
>>> counted_errors.reverse()
>>> for err in counted_errors[:5]:
...     print err
(32, ((), ()))
(5, ((('the', 'at'), ('Rev.', 'nn')),
     (('the', 'at'), ('Rev.', 'np'))))
(5, ((('Assemblies', 'nn'), ('of', 'in')),
     (('Assemblies', 'nns-tl'), ('of', 'in-tl'))))
(4, ((('of', 'in'), ('God', 'nn')),
     (('of', 'in-tl'), ('God', 'np-tl'))))
(3, ((('to', 'to'), ('form', 'nn')),
     (('to', 'to'), ('form', 'vb'))))
```

The fifth line of output records the fact that there were 3 cases where the unigram tagger mistakenly tagged a verb as a noun, following the word *to*. (We encountered the inverse of this mistake for the word *increase* in the above evaluation table, where the unigram tagger tagged *increase* as a verb instead of a noun since it occurred more often in the training data as a verb.) Here, when *form* appears after the word *to*, it is invariably a verb. Evidently, the performance of the tagger would improve if it was modified to consider not just the word being tagged, but also the tag of the word on the left. Such taggers are known as bigram taggers, and we consider them in the next section.

12.2.4 Exercises

1. ◑ **Evaluating a Unigram Tagger**: Apply our evaluation methodology to the unigram tagger developed in the previous section. Discuss your findings.

12.3 Sparse Data and Smoothing

[Introduction to NLTK's support for statistical estimation.]

12.4 The Brill Tagger

A potential issue with n-gram taggers is the size of their n-gram table (or language model). If tagging is to be employed in a variety of language technologies deployed on mobile computing devices, it is important to strike a balance between model size and tagger performance. An n-gram tagger with backoff may store trigram and bigram tables, large sparse arrays which may have hundreds of millions of entries.

A second issue concerns context. The only information an n-gram tagger considers from prior context is tags, even though words themselves might be a useful source of information. It is simply impractical for n-gram models to be conditioned on the identities of words in the context. In this section we examine Brill tagging, a statistical tagging method which performs very well using models that are only a tiny fraction of the size of n-gram taggers.

Brill tagging is a kind of *transformation-based learning*. The general idea is very simple: guess the tag of each word, then go back and fix the mistakes. In this way, a Brill tagger successively transforms a bad tagging of a text into a better one. As with n-gram tagging, this is a *supervised learning* method, since we need annotated training data. However, unlike n-gram tagging, it does not count observations but compiles a list of transformational correction rules.

The process of Brill tagging is usually explained by analogy with painting. Suppose we were painting a tree, with all its details of boughs, branches, twigs and leaves, against a uniform sky-blue background. Instead of painting the tree first then trying to paint blue in the gaps, it is simpler to paint the whole canvas blue, then "correct" the tree section by over-painting the blue background. In the same fashion we might paint the trunk a uniform brown before going back to over-paint further details with even finer brushes. Brill tagging uses the same idea: begin with broad brush strokes then fix up the details, with successively finer changes. Table 12.2 illustrates this process, first tagging with the unigram tagger, then fixing the errors.

Sentence:	**Gold:**	**Unigram:**	**Replace `nn` with `vb` when the previous word is `to`**	**Replace `to` with `in` when the next tag is `nns`**
The	at	at		
President	nn-tl	nn-tl		
said	vbd	vbd		
he	pps	pps		
will	md	md		
ask	vb	vb		
Congress	np	np		
to	to	to		
increase	vb	*nn*	*vb*	
grants	nns	nns		
to	in	*to*	*to*	*in*
states	nns	nns		
for	in	in		
vocational	jj	jj		
rehabilitation	nn	nn		

Table 12.2: Steps in Brill Tagging

In this table we see two rules. All such rules are generated from a template of the following form: form "replace T_1 with T_2 in the context C". Typical contexts are the identity or the tag of the preceding or following word, or the appearance of a specific tag within 2-3 words of of the current word. During its training phase, the tagger guesses values for T_1, T_2 and C, to create thousands of candidate rules. Each rule is scored according to its net benefit: the number of incorrect tags that it corrects, less the number of correct tags it incorrectly modifies. This process is best illustrated by a listing of the output from the NLTK Brill tagger (here run on tagged Wall Street Journal text from the Penn Treebank).

```
Loading tagged data...
Training unigram tagger: [accuracy: 0.820940]
Training Brill tagger on 37168 tokens...

Iteration 1: 1482 errors; ranking 23989 rules;
  Found: "Replace POS with VBZ if the preceding word is tagged PRP"
  Apply: [changed 39 tags: 39 correct; 0 incorrect]

Iteration 2: 1443 errors; ranking 23662 rules;
  Found: "Replace VBP with VB if one of the 3 preceding words is tagged MD"
  Apply: [changed 36 tags: 36 correct; 0 incorrect]

Iteration 3: 1407 errors; ranking 23308 rules;
  Found: "Replace VBP with VB if the preceding word is tagged TO"
  Apply: [changed 24 tags: 23 correct; 1 incorrect]

Iteration 4: 1384 errors; ranking 23057 rules;
  Found: "Replace NN with VB if the preceding word is to"
  Apply: [changed 67 tags: 22 correct; 45 incorrect]
  ...
Iteration 20: 1138 errors; ranking 20717 rules;
  Found: "Replace RBR with JJR if one of the 2 following words is tagged NNS"
  Apply: [changed 14 tags: 10 correct; 4 incorrect]

Iteration 21: 1128 errors; ranking 20569 rules;
  Found: "Replace VBD with VBN if the preceding word is tagged VBD"
[insufficient improvement; stopping]

Brill accuracy: 0.835145
```

Brill taggers have another interesting property: the rules are linguistically interpretable. Compare this with the n-gram taggers, which employ a potentially massive table of n-grams. We cannot learn much from direct inspection of such a table, in comparison to the rules learned by the Brill tagger.

12.4.1 Exercises

1. ☼ Try the Brill tagger demonstration, as follows:

   ```
   from nltk_lite.tag import brill
   brill.demo()
   ```

2. ◑ Consult the documentation for the demo function, using `help(brill.demo)`. Experiment with the tagger by setting different values for the parameters. Is there any trade-off between training time (corpus size) and performance?

3. ★ Inspect the diagnostic files created by the tagger `rules.out` and `errors.out`. Obtain the demonstration code (`nltk_lite/tag/brill.py`) and create your own version of the Brill tagger.

   a) Delete some of the rule templates, based on what you learned from inspecting `rules.out`.

   b) Add some new rule templates which employ contexts that might help to correct the errors you saw in `errors.out`.

(We are grateful to Christopher Maloof for developing NLTK's Brill tagger, and Trevor Cohn for developing NLTK's HMM tagger.)

12.5 The HMM Tagger

[Overview of NLTK's HMM tagger.]

12.6 Evaluating Chunk Parsers

An easy way to evaluate a chunk parser is to take some already chunked text, strip off the chunks, rechunk it, and compare the result with the original chunked text. The `ChunkScore.score()` function takes the correctly chunked sentence as its first argument, and the newly chunked version as its second argument, and compares them. It reports the fraction of actual chunks that were found (recall), the fraction of hypothesized chunks that were correct (precision), and a combined score, the F-measure (the harmonic mean of precision and recall).

A number of different metrics can be used to evaluate chunk parsers. We will concentrate on a class of metrics that can be derived from two sets:

- **guessed**: The set of chunks returned by the chunk parser.
- **correct**: The correct set of chunks, as defined in the test corpus.

The evaluation method we will use comes from the field of information retrieval, and considers the performance of a document retrieval system. We will set up an analogy between the correct set of chunks and a user's so-called "information need", and between the set of returned chunks and a system's returned documents. Consider the following diagram.

The intersection of these sets defines four regions: the true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN). Two standard measures are *precision*, the fraction of guessed chunks that were correct TP/(TP+FP), and *recall*, the fraction of correct chunks that were identified TP/(TP+FN). A third measure, the *F measure*, is the harmonic mean of precision and recall, i.e. 1/(0.5/Precision + 0.5/Recall).

During evaluation of a chunk parser, it is useful to flatten a chunk structure into a tree consisting only of a root node and leaves:

```
>>> from nltk_lite import chunk
>>> correct = chunk.tagstr2tree(
...     "[ the/DT little/JJ cat/NN ] sat/VBD on/IN [ the/DT mat/NN ]")
>>> correct.flatten()
(S: ('the', 'DT') ('little', 'JJ') ('cat', 'NN') ('sat', 'VBD')
('on', 'IN') ('the', 'DT') ('mat', 'NN'))
```

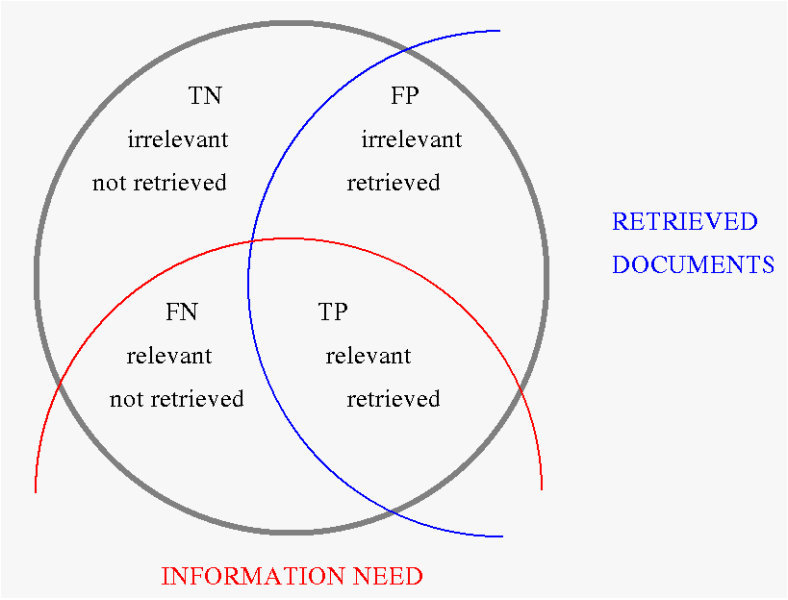


Figure 12.1: True and False Positives and Negatives

We run a chunker over this flattened data, and compare the resulting chunked sentences with the originals, as follows:

```
>>> from nltk_lite.chunk import *
>>> chunkscore = ChunkScore()
>>> rule = ChunkRule('<PRP|DT|POS|JJ|CD|N.*>+',
...                  "Chunk items that often occur in NPs")
>>> chunkparser = RegexpChunk([rule], chunk_node='NP')
>>> guess = chunkparser.parse(correct.flatten())
>>> chunkscore.score(correct, guess)
>>> print chunkscore
ChunkParse score:
    Precision: 100.0%
    Recall:    100.0%
    F-Measure: 100.0%
```

`ChunkScore` is a class for scoring chunk parsers. It can be used to evaluate the output of a chunk parser, using precision, recall, f-measure, missed chunks, and incorrect chunks. It can also be used to combine the scores from the parsing of multiple texts. This is quite useful if we are parsing a text one sentence at a time. The following program listing shows a typical use of the `ChunkScore` class. In this example, `chunkparser` is being tested on each sentence from the Wall Street Journal tagged files.

```
>>> from itertools import islice
>>> from nltk_lite.corpora import treebank
>>> rule = ChunkRule('<DT|JJ|NN>+', "Chunk sequences of DT, JJ, and NN")
>>> chunkparser = RegexpChunk([rule], chunk_node='NP')
>>> chunkscore = ChunkScore()
>>> for chunk_struct in islice(treebank.chunked(), 10):
```

```
...     test_sent = chunkparser.parse(chunk_struct.flatten())
...     chunkscore.score(chunk_struct, test_sent)
>>> print chunkscore
ChunkParse score:
    Precision:  48.6%
    Recall:     34.0%
    F-Measure:  40.0%
```

The overall results of the evaluation can be viewed by printing the `ChunkScore`. Each evaluation metric is also returned by an accessor method: `precision()`, `recall`, `f_measure`, `missed`, and `incorrect`. The `missed` and `incorrect` methods can be especially useful when trying to improve the performance of a chunk parser. Here are the missed chunks:

```
>>> from random import shuffle
>>> missed = chunkscore.missed()
>>> shuffle(missed)
>>> print missed[:10]
[(('A', 'DT'), ('Lorillard', 'NNP'), ('spokeswoman', 'NN')),
 (('even', 'RB'), ('brief', 'JJ'), ('exposures', 'NNS')),
 (('its', 'PRP$'), ('Micronite', 'NN'), ('cigarette', 'NN'), ('filters', 'NNS')),
 (('30', 'CD'), ('years', 'NNS')),
 (('workers', 'NNS'),),
 (('preliminary', 'JJ'), ('findings', 'NNS')),
 (('Medicine', 'NNP'),),
 (('Consolidated', 'NNP'), ('Gold', 'NNP'), ('Fields', 'NNP'), ('PLC', 'NNP')),
 (('its', 'PRP$'), ('Micronite', 'NN'), ('cigarette', 'NN'), ('filters', 'NNS')),
 (('researchers', 'NNS'),)]
```

Here are the incorrect chunks:

```
>>> incorrect = chunkscore.incorrect()
>>> shuffle(incorrect)
>> print incorrect[:10]
[(('New', 'JJ'), ('York-based', 'JJ')),
 (('Micronite', 'NN'), ('cigarette', 'NN')),
 (('a', 'DT'), ('forum', 'NN'), ('likely', 'JJ')),
 (('later', 'JJ'),),
 (('preliminary', 'JJ'),),
 (('New', 'JJ'), ('York-based', 'JJ')),
 (('resilient', 'JJ'),),
 (('group', 'NN'),),
 (('the', 'DT'),),
 (('Micronite', 'NN'), ('cigarette', 'NN'))]
```

As we saw with tagging, we need to interpret the performance scores for a chunker relative to a baseline. Perhaps the most naive chunking method is to classify every tag in the training data as to whether it occurs inside or outside a chunk more often. We can do this easily using a chunked corpus and a conditional frequency distribution as shown below:

```
>>> from nltk_lite.probability import ConditionalFreqDist
>>> from nltk_lite.parse import Tree
>>> import re
>>> cfdist = ConditionalFreqDist()
>>> chunk_data = list(treebank.chunked())
```

```
>>> split = len(chunk_data)*9/10
>>> train, test = chunk_data[:split], chunk_data[split:]
>>> for chunk_struct in train:
...     for constituent in chunk_struct:
...         if isinstance(constituent, Tree):
...             for (word, tag) in constituent.leaves():
...                 cfdist[tag].inc(True)
...         else:
...             (word, tag) = constituent
...             cfdist[tag].inc(False)

>>> chunk_tags = [tag for tag in cfdist.conditions() if cfdist[tag].max() == True]
>>> chunk_tags = [re.sub(r'(\W)', r'\\\1', tag) for tag in chunk_tags]
>>> tag_pattern = '<' + '|'.join(chunk_tags) + '>+'
>>> print 'Chunking:', tag_pattern
Chunking: <PRP\$|VBG\|NN|POS|WDT|JJ|WP|DT|\#|\$|NN|FW|PRP|NNS|NNP|LS|PDT|RBS|CD|EX|WP\$
```

Now, in the evaluation phase we chunk any sequence of those tags:

```
>>> rule = ChunkRule(tag_pattern, 'Chunk any sequence involving commonly chunked tags')
>>> chunkparser = RegexpChunk([rule], chunk_node='NP')
>>> chunkscore = ChunkScore()
>>> for chunk_struct in test:
...     test_sent = chunkparser.parse(chunk_struct.flatten())
...     chunkscore.score(chunk_struct, test_sent)
>>> print chunkscore
ChunkParse score:
    Precision:  90.7%
    Recall:     94.0%
    F-Measure:  92.3%
```

12.6.1 Exercises

1. ◑ **Chunker Evaluation:** Carry out the following evaluation tasks for any of the chunkers you have developed earlier. (Note that most chunking corpora contain some internal inconsistencies, such that any reasonable rule-based approach will produce errors.)

   a) Evaluate your chunker on 100 sentences from a chunked corpus, and report the precision, recall and F-measure.

   b) Use the `chunkscore.missed()` and `chunkscore.incorrect()` methods to identify the errors made by your chunker. Discuss.

   c) Compare the performance of your chunker to the baseline chunker discussed in the evaluation section of this chapter.

2. ★ **Transformation-Based Chunking:** Apply the n-gram and Brill tagging methods to IOB chunk tagging. Instead of assigning POS tags to words, here we will assign IOB tags to the POS tags. E.g. if the tag `DT` (determiner) often occurs at the start of a chunk, it will be tagged `B` (begin). Evaluate the performance of these chunking methods relative to the regular expression chunking methods covered in this chapter.

1. ★ **Statistically Improbable Phrases:** Design an algorithm to find the statistically improbable phrases of a document collection. http://www.amazon.com/gp/search-inside/sipshelp.html

12.7 Information Extraction

Information Extraction is the task of converting **unstructured data** (e.g., unrestricted text) or **semi-structured data** (e.g., web pages marked up with HTML) into **structured data** (e.g., tables in a relational database). For example, let's suppose we are given a text containing the fragment (212), and let's also suppose we are trying to find pairs of entities *X* and *Y* that stand in the relation 'organization *X* is located in location *Y*'.

(212) ... said William Gale, an economist at the Brookings Institution, the research group in Washington.

As a result of processing this text, we should be able to add the pair *Brookings Institution*, *Washington* to this relation. As we will see shortly, Information Extraction proceeds on the assumption that we are only looking for specific sorts of information, and these have been decided in advance. This limitation has been a necessary concession to allow the robust processing of unrestricted text.

Potential applications of Information Extraction are many, and include business intelligence, resume harvesting, media analysis, sentiment detection patent search, and email scanning. A particularly important area of current research involves the attempt to extract structured data out of electronically-available scientific literature, most notably in the domain of biology and medicine.

Information Extraction is usually broken down into at least two major steps: **Named Entity Recognition** and **Relation Extraction**. Named Entities (NE s) are usually taken to be noun phrases that denote specific types of individuals such as organizations, persons, dates, and so on. Thus, we might use the following XML annotations to mark-up the NEs in (212):

(213) ... said <ne type='PERSON'>William Gale</ne>, an economist at the <ne type='ORGANIZATION'>Brook
Institution</ne>, the research group in <ne type='LOCATION'>Washington<ne>.

How do we go about identifying NEs? Our first thought might be that we could look up candidate expressions in an appropriate list of names. For example, in the case of locations, we might try using a resource such as the Alexandria Gazetteer. Depending on the nature of our input data, this be adequate — such a gazetteer is likely to have good coverage of international cities and many locations in the U.S.A., but will probably be missing the names of obscure villages in remote regions. However, a list of names for people or organization will probably have poor coverage. New organizations, and new names for them, are coming into existence every day, so if we are trying to deal with contemporary newswire or blog entries, say, it is unlikely that we will be able to recognize many of the NEs by using gazetteer lookup.

A second consideration is that many NE terms are ambiguous. Thus *May* and *North* are likely to be parts of NEs for DATE and LOCATION, respectively, but could both be part of a PERSON NE; conversely *Christian Dior* looks like a PERSON NE but is more likely to be of type ORGANIZATION. A terms like *Yankee* will be ordinary modifier in some contexts, but will be marked as an NE of type ORGANIZATION in the phrase *Yankee infielders*. To summarize, we cannot reliably detect NEs by looking them up in a gazetteer, and it is also hard to develop rules that will correctly recognize ambiguous NEs on the basis of their context of occurrence. Although lookup may contribute to a solution, most contemporary approaches to Named Entity Recognition treat it as a statistical classification task that requires training data for good performance. This task is facilitated by adopting an appropriate data representation, such as the IOB tags which we saw being deployed in the CoNLL chunk data (Chapter 5). For example, here are a representative few lines from the CONLL 2002 (`conll2002`) Dutch training data:

```
Eddy N B-PER
Bonte N I-PER
is V O
woordvoerder N O
van Prep O
diezelfde Pron O
Hogeschool N B-ORG
. Punc O
```

As noted before, in this representation, there is one token per line, each with its part-of-speech tag and its NE tag. When NEs have been identified in a text, we then want to extract relations that hold between them. As indicated earlier, we will typically be looking for relations between specified types of NE. One way of approaching this task is to initially look for all triples of the form X, α, Y, where X and Y are NE s of the required types, and α is the string of words that intervenes between X and Y. We can then use regular expressions to pull out just those instances of α that express the relation that we are looking for. The following example searches for strings that contain the word *in*. The special character expression `(?!\b.+ing\b)` is a negative lookahead condition which allows us to disregard strings such as *success in supervising the transition of*, where *in* is followed by a gerundive verb.

```
>>> from nltk_lite.contrib.ieer_rels import *
>>> from itertools import islice
>>> IN = re.compile(r'.*\bin\b(?!\b.+ing\b)')
>>> ieer_trees = (d['text'] for d in ieer.dictionary())
>>> for r in islice(ieer_trees, relextract('ORG', 'LOC', pattern = IN), 29, 39):
...     print show_tuple(r)
[ORG: Cooper-Hewitt Museum] in [LOC: New York]
[ORG: Canadian Museum of Civilization] in [LOC: Hull]
[ORG: Kramerbooks and Afterwords] , an independent store in [LOC: Washington]
[ORG: Waterford Foundation] in historic [LOC: Waterford]
[ORG: U.S. District Court] in [LOC: Manhattan]
[ORG: House Transportation Committee] , secured the most money in the [LOC: New York]
[ORG: Angels] in [LOC: Anaheim]
[ORG: Bayona] , which is in the heart of the [LOC: French Quarter]
[ORG: CBS Studio Center] in [LOC: Studio City]
[ORG: Smithsonian Institution] in [LOC: Washington]
```

As you will see, although searching for *in* does a reasonably good job, there is some inaccuracy in the output which is hard to avoid — there is unlikely to be simple string-based method of excluding fillers such as *secured the most money in the*.

As shown above, the `conll2002` corpus contains not just NE annotation but also part-of-speech tags. In principle, this allows us to devise patterns which are sensitive to these tags.

```
>>> vnv = """
>>> (
>>> is/V|
>>> was/V|
>>> werd/V|
>>> wordt/V
>>> )
>>> .*
>>> van/Prep
>>> """
```

```
>>> VAN = re.compile(vnv, re.VERBOSE)
>>> for r in relextract('PER', 'ORG', corpus='conll2002-ned', pattern = VAN):
...     print show_tuple(r)
```

12.7.1 Exercises

12.8 Conclusions

[To be written]

12.9 Further Reading

Sekine -- 140 types of NE.

Chapter 13

Managing Linguistic Data

13.1 Introduction

Linguistic fieldwork deals with a variety of data types, the most important being lexicons, paradigms and texts. A lexicon is a database of words, minimally containing part of speech information and glosses. A paradigm, broadly construed, is any kind of rational tabulation of words or phrases to illustrate contrasts and systematic variation. A text is essentially any larger unit such as a narrative or a conversation. In addition to these data types, linguistic fieldwork involves various kinds of description, such as field notes, grammars and analytical papers.

These various kinds of data and description enter into a complex web of relations. For example, the discovery of a new word in a text may require an update to the lexicon and the construction of a new paradigm (e.g. to correctly classify the word). Such updates may occasion the creation of some field notes, the extension of a grammar and possibly even the revision of the manuscript for an analytical paper. Progress on description and analysis gives fresh insights about how to organise existing data and it informs the quest for new data. Whether one is sorting data, or generating tabulations, or gathering statistics, or searching for a (counter-)example, or verifying the transcriptions used in a manuscript, the principal challenge is computational.

In the following we will consider various methods for manipulating linguistic field data using the Natural Language Toolkit. We begin by considering methods for processing data created with proprietary tools (e.g. Microsoft Office products). The bulk of the discussion focusses on field data stored in the popular *Toolbox* format.

> **Note**
>
> Other sections, still to be written, will cover the collection and curation of corpora; the lifecycle of linguistic data, including coding/annotation and automatic learning of annotations; and language resources more generally which are crucial in linguistic research and commercial NLP.

13.2 XML and ElementTree

- inspecting and processing XML
- example: find nodes matching some criterion and add an attribute
- Shakespeare XML corpus example

13.3 Tools and technologies for language documentation and description

Language documentation projects are increasing in their reliance on new digital technologies and software tools. Bird and Simons (2003) identified and categorized a wide variety of these tools. We briefly review these here, and mention various ways that our own programs can interface with them.

13.3.1 General purpose tools

Conventional office software is widely used in computer-based language documentation work, given its familiarity and ready availability. This includes word processors and spreadsheets.

Word Processors. These are often used in creating dictionaries and interlinear texts. However, it is rather time-consuming to maintain the consistency of the content and format. Consider a dictionary in which each entry has a part-of-speech field, drawn from a set of 20 possibilities, displayed after the pronunciation field, and rendered in 11-point bold. No convential word processor has search or macro functions capable of verifying that all part-of-speech fields have been correctly entered and displayed. This task requires exhaustive manual checking. If the word processor permits the document to be saved in a non-proprietary format, such as RTF, HTML, or XML, it may be possible to write programs to do this checking automatically.

Consider the following fragment of a lexical entry: "sleep [sli:p] **vi** *condition of body and mind...*". We can enter this in MSWord, then "Save as Web Page", then inspect the resulting HTML:

```
<p class=MsoNormal>sleep
  <span style='mso-spacerun:yes'> </span>
  [<span class=SpellE>sli:p</span>]
  <span style='mso-spacerun:yes'> </span>
  <b><span style='font-size:11.0pt'>vi</span></b>
  <span style='mso-spacerun:yes'> </span>
  <i>a condition of body and mind ...<o:p></o:p></i>
</p>
```

Observe that the entry is represented as an HTML paragraph, using the `<p>` element, and that the part of speech appears inside a `<span style='font-size:11.0pt'>` element. The following program defines the set of legal parts-of-speech, `legal_pos`. Then it extracts all 11-point content from the `dict.htm` file and stores it in the set `used_pos`. Observe that the search pattern contains a parenthesized sub-expression; only the material that matches this sub-expression is returned by `re.findall`. Finally, the program constructs the set of illegal parts-of-speech as `used_pos - legal_pos`:

```
>>> import re
>>> legal_pos = set(['n', 'v.t.', 'v.i.', 'adj', 'det'])
>>> pattern = re.compile(r"'font-size:11.0pt'>([a-z.]+)<")
>>> document = open("dict.htm").read()
>>> used_pos = set(re.findall(pattern, document))
>>> illegal_pos = used_pos.difference(legal_pos)
>>> print list(illegal_pos)
['v.i', 'intrans']
```

This simple program represents the tip of the iceberg. We can develop sophisticated tools to check the consistency of word processor files, and report errors so that the maintainer of the dictionary can correct the original file *using the original word processor*. We can write other programs to *convert*

the data into a different format. For example, the following program extracts the words and their pronunciations and generates output in "comma-separated value" (CSV) format:

```
>>> import re
>>> document = open("dict.htm").read()
>>> document = re.sub("[\r\n]", "", document)
>>> word_pattern = re.compile(r">([\w]+)")
>>> pron_pattern = re.compile(r"\[.*>([a-z:]+)<.*\]")
>>> for entry in document.split("<p"):
...     word_match = word_pattern.search(entry)
...     pron_match = pron_pattern.search(entry)
...     if word_match and pron_match:
...         lex = word_match.group(1)
...         pos = pron_match.group(1)
...         print '"%s","%s"' % (lex, pos)
"sleep","sli:p"
"walk","wo:k"
"wake","weik"
```

We could also produce output in the Toolbox format (to be discussed in detail later):

```
\lx sleep
\ph sli:p
\ps v.i
\gl a condition of body and mind ...

\lx walk
\ph wo:k
\ps v.intr
\gl progress by lifting and setting down each foot ...

\lx wake
\ph weik
\ps intrans
\gl cease to sleep
```

Spreadsheets. These are often used for wordlists or paradigms. A comparative wordlist may be stored in a spreadsheet, with a row for each cognate set, and a column for each language. Examples are available from `www.rosettaproject.org`. Programs such as Excel can export spreadsheets in the CSV format, and we can write programs to manipulate them, with the help of Python's `csv` module. For example, we may want to print out cognates having an edit-distance of at least three from each other (i.e. 3 insertions, deletions, or substitutions).

Databases. Sometimes lexicons are stored in a full-fledged relational database. When properly normalized, these databases can implement many well-formedness constraints. For example, we can require that all parts-of-speech come from a specified vocabulary by declaring that the part-of-speech field is an *enumerated type*. However, the relational model is often too restrictive for linguistic data, which typically has many optional and repeatable fields (e.g. dictionary sense definitions and example sentences). Query languages such as SQL cannot express many linguistically-motivated queries, e.g. *find all words that appear in example sentences for which no dictionary entry is provided*. Now supposing that the database supports exporting data to CSV format, and that we can save the data to a file `dict.csv`:

```
    "sleep","sli:p","v.i","a condition of body and mind ..."
    "walk","wo:k","v.intr","progress by lifting and setting down each foot ..."
    "wake","weik","intrans","cease to sleep"
```

Now we can express this query in the following program:

```
>>> import csv
>>> lexemes = []
>>> defn_words = []
>>> for row in csv.reader(open("dict.csv")):
...     lexeme, pron, pos, defn = row
...     lexemes.append(lexeme)
...     defn_words += defn.split()
>>> undefined = list(set(defn_words).difference(set(lexemes)))
>>> undefined.sort()
>>> print undefined
['...', 'a', 'and', 'body', 'by', 'cease', 'condition', 'down', 'each',
'foot', 'lifting', 'mind', 'of', 'progress', 'setting', 'to']
```

13.4 Processing Toolbox Data

Over the last two decades, several dozen tools have been developed that provide specialized support for linguistic data management. (Please see Bird and Simons 2003 for a detailed list of such tools.) Perhaps the single most popular tool for managing linguistic field data is *Shoebox*, recently renamed *Toolbox*. Toolbox uses a simple file format which we can easily read and write, permitting us to apply computational methods to linguistic field data. In this section we discuss a variety of techniques for manipulating Toolbox data in ways that are not supported by the Toolbox software.

A Toolbox file consists of a collection of *entries* (or *records*), where each record is made up of one or more *fields*. Here is an example of an entry taken from a Toolbox dictionary of Rotokas. (Rotokas is an East Papuan language spoken on the island of Bougainville; this data was provided by Stuart Robinson, and is a sample from a larger lexicon):

```
\lx kaa
\ps N
\pt MASC
\cl isi
\ge cooking banana
\tkp banana bilong kukim
\pt itoo
\sf FLORA
\dt 12/Aug/2005
\ex Taeavi iria kaa isi kovopaueva kaparapasia.
\xp Taeavi i bin planim gaden banana bilong kukim tasol long paia.
\xe Taeavi planted banana in order to cook it.
```

This lexical entry contains the following fields: `lx` lexeme; `ps` part-of-speech; `pt` part-of-speech; `cl` classifier; `ge` English gloss; `tkp` Tok Pisin gloss; `sf` Semantic field; `dt` Date last edited; `ex` Example sentence; `xp` Pidgin translation of example; `xe` English translation of example. These field names are preceded by a backslash, and must always appear at the start of a line. The characters of the field names must be alphabetic. The field name is separated from the field's contents by whitespace. The contents can be arbitrary text, and can continue over several lines (but cannot contain a line-initial backslash).

13.4.1 Accessing Toolbox Data

We can use the `toolbox.parse_corpus()` method to access a Toolbox file and load it into an `elementtree` object.

```
>>> from nltk_lite.corpora import toolbox
>>> lexicon = toolbox.parse_corpus('rotokas.dic')
```

There are two ways to access the contents of the lexicon object, by indexes and by paths. Indexes use the familiar syntax, thus `lexicon[3]` returns entry number 3 (which is actually the fourth entry counting from zero). And `lexicon[3][0]` returns its first field:

```
>>> lexicon[3][0]
<Element lx at 77bd28>
>>> lexicon[3][0].tag
'lx'
>>> lexicon[3][0].text
'kaa'
```

We can iterate over all the fields of a given entry:

```
>>> print toolbox.to_sfm_string(lexicon[3])
\lx kaa
\ps N
\pt MASC
\cl isi
\ge cooking banana
\tkp banana bilong kukim
\pt itoo
\sf FLORA
\dt 12/Aug/2005
\ex Taeavi iria kaa isi kovopaueva kaparapasia.
\xp Taeavi i bin planim gaden banana bilong kukim tasol long paia.
\xe Taeavi planted banana in order to cook it.
```

The second way to access the contents of the lexicon object uses paths. The lexicon is a series of `record` objects, each containing a series of field objects, such as `lx` and `ps`. We can conveniently address all of the lexemes using the path `record/lx`. Here we use the `findall()` function to search for any matches to the path `record/lx`, and we access the text content of the element, normalising it to lowercase.

```
>>> [lexeme.text.lower() for lexeme in lexicon.findall('record/lx')]
['kaa', 'kaa', 'kaa', 'kaakaaro', 'kaakaaviko', 'kaakaavo', 'kaakaoko',
'kaakasi', 'kaakau', 'kaakauko', 'kaakito', 'kaakuupato', ..., 'kuvuto']
```

13.4.2 Adding and Removing Fields

It is often convenient to add new fields that are derived from existing ones. Such fields often facilitate analysis. For example, let us define a function which maps a string of consonants and vowels to the corresponding CV sequence, e.g. `kakapua` would map to `CVCVCVV`.

```
>>> import re
>>> def cv(s):
...     s = s.lower()
```

```
...     s = re.sub(r'[^a-z]',     r'_', s)
...     s = re.sub(r'[aeiou]',    r'V', s)
...     s = re.sub(r'[^V_]',      r'C', s)
...     return (s)
```

This mapping has four steps. First, the string is converted to lowercase, then we replace any non-alphabetic characters `[^a-z]` with an underscore. Next, we replace all vowels with `V`. Finally, anything that is not a `V` or an underscore must be a consonant, so we replace it with a `C`. Now, we can scan the lexicon and add a new `cv` field after every `lx` field.

```
>>> from nltk_lite.etree.ElementTree import SubElement
>>> for entry in lexicon:
...     for field in entry:
...         if field.tag == 'lx':
...             cv_field = SubElement(entry, 'cv')
...             cv_field.text = cv(field.text)
```

Let's see what this does for a particular entry. Note the last line of output, which shows the new CV field:

```
>>> print toolbox.to_sfm_string(lexicon[53])
\lx kaeviro
\ps V
\pt A
\ge lift off
\ge take off
\tkp go antap
\sc MOTION
\vx 1
\nt used to describe action of plane
\dt 03/Jun/2005
\ex Pita kaeviroroe kepa kekesia oa vuripierevo kiuvu.
\xp Pita i go antap na lukim haus win i bagarapim.
\xe Peter went to look at the house that the wind destroyed.
\cv CVVCVCV
```

(NB. To insert this field in a different position, we need to create the new `cv` field using `Element('cv')`, assign a text value to it then use the `insert()` method of the parent element.)

This technique can be used to make copies of Toolbox data that *lack* particular fields. For example, we may want to sanitise our lexical data before giving it to others, by removing unnecessary fields (e.g. fields containing personal comments.)

```
>>> retain = ('lx', 'ps', 'ge')
>>> for entry in lexicon:
...     entry[:] = [f for f in entry if f.tag in retain]
>>> print toolbox.to_sfm_string(lexicon[53])
\lx kaeviro
\ps V.A
\ge lift off
\ge take off
```

13.4.3 Formatting Entries

We can also print a formatted version of a lexicon. It allows us to request specific fields without needing to be concerned with their relative ordering in the original file.

```
>>> lexicon = toolbox.parse_corpus('rotokas.dic')
>>> for entry in lexicon[70:80]:
...     lx = entry.findtext('lx')
...     ps = entry.findtext('ps')
...     ge = entry.findtext('ge')
...     print "%s (%s) '%s'" % (lx, ps, ge)
kakae (???) 'small'
kakae (CLASS) 'child'
kakaevira (ADV) 'small-like'
kakapikoa (???) 'small'
kakapikoto (N) 'newborn baby'
kakapu (V) 'place in sling for purpose of carrying'
kakapua (N) 'sling for lifting'
kakara (N) 'arm band'
Kakarapaia (N) 'village name'
kakarau (N) 'frog'
```

Note

Producing CSV output
We could have produced comma-separated value (CSV) format with a slightly different print statement: `print '"%s";"%s";"%s"\n' % (lx, ps, ge)`

We can use the same idea to generate HTML tables instead of plain text. This would be useful for publishing a Toolbox lexicon on the web. It produces HTML elements `<table>`, `<tr>` (table row), and `<td>` (table data).

```
>>> html = "<table>\n"
>>> for entry in lexicon[70:80]:
...     lx = entry.findtext('lx')
...     ps = entry.findtext('ps')
...     ge = entry.findtext('ge')
...     html += "  <tr><td>%s</td><td>%s</td><td>%s</td></tr>\n" % (lx, ps, ge)
>>> html += "</table>"
>>> print html
<table>
  <tr><td>kakae</td><td>???</td><td>small</td></tr>
  <tr><td>kakae</td><td>CLASS</td><td>child</td></tr>
  <tr><td>kakaevira</td><td>ADV</td><td>small-like</td></tr>
  <tr><td>kakapikoa</td><td>???</td><td>small</td></tr>
  <tr><td>kakapikoto</td><td>N</td><td>newborn baby</td></tr>
  <tr><td>kakapu</td><td>V</td><td>place in sling for purpose of carrying</td></tr>
  <tr><td>kakapua</td><td>N</td><td>sling for lifting</td></tr>
  <tr><td>kakara</td><td>N</td><td>arm band</td></tr>
  <tr><td>Kakarapaia</td><td>N</td><td>village name</td></tr>
  <tr><td>kakarau</td><td>N</td><td>frog</td></tr>
</table>
```

XML output

```
>>> import sys
>>> from nltk_lite.etree.ElementTree import ElementTree
>>> tree = ElementTree(lexicon[3])
>>> tree.write(sys.stdout)
<record>
  <lx>kaa</lx>
  <ps>N</ps>
  <pt>MASC</pt>
  <cl>isi</cl>
  <ge>cooking banana</ge>
  <tkp>banana bilong kukim</tkp>
  <pt>itoo</pt>
  <sf>FLORA</sf>
  <dt>12/Aug/2005</dt>
  <ex>Taeavi iria kaa isi kovopaueva kaparapasia.</ex>
  <xp>Taeavi i bin planim gaden banana bilong kukim tasol long paia.</xp>
  <xe>Taeavi planted banana in order to cook it.</xe>
</record>
```

13.4.4 Exploration

In this section we consider a variety of analysis tasks.

Reduplication: First, we will develop a program to find reduplicated words. In order to do this we need to store all verbs, along with their English glosses. We need to keep the glosses so that they can be displayed alongside the wordforms. The following code defines a Python dictionary `lexgloss` which maps verbs to their English glosses:

```
>>> lexgloss = {}
>>> for entry in lexicon:
...     lx = entry.findtext('lx')
...     if lx and entry.findtext('ps')[0] == 'V':
...         lexgloss[lx] = entry.findtext('ge')
kasi (burn); kasikasi (angry)
kee (shatter); keekee (chipped)
kauo (jump); kauokauo (jump up and down)
kea (confused); keakea (lie)
kape (unable to meet); kapekape (embrace)
kapo (fasten.cover.strip); kapokapo (fasten.cover.strips)
kavo (collect); kavokavo (perform sorcery)
karu (open); karukaru (open)
kare (return); karekare (return)
kari (rip); karikari (tear)
kae (blow); kaekae (tempt)
```

Next, for each verb `lex`, we will check if the lexicon contains the reduplicated form `lex+lex`. If it does, we report both forms along with their glosses.

```
>>> for lex in lexgloss:
...     if lex+lex in lexgloss:
...         print "%s (%s); %s (%s)" %\
...             (lex, lexgloss[lex], lex+lex, lexgloss[lex+lex])
```

Complex Search Criteria: Phonological description typically identifies the segments, alternations, syllable canon and so forth. It is relatively straightforward to count up the occurrences of all the different types of CV syllables that occur in lexemes.

In the following example, we first import the regular expression and probability modules. Then we iterate over the lexemes to find all sequences of a non-vowel `[^aeiou]` followed by a vowel `[aeiou]`.

```
>>> from nltk_lite.tokenize import regexp
>>> from nltk_lite.probability import FreqDist
>>> fd = FreqDist()
>>> lexemes = [lexeme.text.lower() for lexeme in lexicon.findall('record/lx')]
>>> for lex in lexemes:
...     for syl in regexp(lex, pattern=r'[^aeiou][aeiou]'):
...         fd.inc(syl)
```

Now, rather than just printing the syllables and their frequency counts, we can tabulate them to generate a useful display.

```
>>> for vowel in 'aeiou':
...     for cons in 'ptkvsr':
...          print '%s%s:%4d ' % (cons, vowel, fd.count(cons+vowel)),
...     print
pa:  83  ta:  47  ka: 428  va:  93  sa:   0  ra: 187
pe:  31  te:   8  ke: 151  ve:  27  se:   0  re:  63
pi: 105  ti:   0  ki:  94  vi: 105  si: 100  ri:  84
po:  34  to: 148  ko: 430  vo:  48  so:   2  ro:  89
pu:  51  tu:  37  ku: 175  vu:  49  su:   1  ru:  79
```

Consider the `t` and `s` columns, and observe that `ti` is not attested, while `si` is frequent. This suggests that a phonological process of palatalisation is operating in the language. We would then want to consider the other syllables involving `s` (e.g. the single entry having *su*, namely *kasuari* 'cassowary' is a loanword).

Prosodically-motivated search: A phonological description may include an examination of the segmental and prosodic constraints on well-formed morphemes and lexemes. For example, we may want to find trisyllabic verbs ending in a long vowel. Our program can make use of the fact that syllable onsets are obligatory and simple (only consist of a single consonant). First, we will encapsulate the syllabic counting part in a separate function. It gets the CV template of the word `cv(word)` and counts the number of consonants it contains:

```
>>> def num_cons(word):
...     template = cv(word)
...     return template.count('C')
```

We also encapsulate the vowel test in a function, as this improves the readability of the final program. This function returns the value `True` just in case `char` is a vowel.

```
>>> def is_vowel(char):
...     return (char in 'aeiou')
```

Over time we may create a useful collection of such functions. We can save them in a file `utilities.py`, and then at the start of each program we can simply import all the functions in one go using `from utilities import *`. We take the entry to be a verb if the first letter of its part of speech is a `V`. Here, then, is the program to display trisyllabic verbs ending in a long vowel:

```
>>> for entry in lexicon:
...     lx = entry.findtext('lx')
...     if lx:
...         ps = entry.findtext('ps')
...         if num_cons(lx) == 3 and ps[0] == 'V'\
...           and is_vowel(lx[-1]) and is_vowel(lx[-2]):
...             ge = entry.findtext('ge')
...             print "%s (%s) '%s'" % (lx, ps, ge)
kaetupie (V.B) 'tighten'
kakupie (V.B) 'shout'
kapatau (V.B) 'add to'
kapuapie (V.B) 'wound'
kapupie (V.B) 'close tight'
kapuupie (V.B) 'close'
karepie (V.B) 'return'
karivai (V.A) 'have an appetite'
kasipie (V.B) 'care for'
kaukaupie (V.B) 'shine intensely'
kavorou (V.A) 'covet'
kavupie (V.B) 'leave.behind'
...
kuverea (V.A) 'incomplete'
```

Finding Minimal Sets: In order to establish a contrast segments (or lexical properties, for that matter), we would like to find pairs of words which are identical except for a single property. For example, the words pairs *mace* vs *maze* and *face* vs *faze* — and many others like them — demonstrate the existence of a phonemic distinction between *s* and *z* in English. NLTK-Lite provides flexible support for constructing minimal sets, using the `MinimalSet()` class. This class needs three pieces of information for each item to be added: `context`: the material that must be fixed across all members of a minimal set; `target`: the material that changes across members of a minimal set; `display`: the material that should be displayed for each item.

Examples of Minimal Set Parameters			
Minimal Set	**Context**	**Target**	**Display**
bib, *bid*, *big*	first two letters	third letter	word
deal (N), *deal (V)*	whole word	pos	word (pos)

Table 13.1:

We begin by creating a list of parameter values, generated from the full lexical entries. In our first example, we will print minimal sets involving lexemes of length 4, with a target position of 1 (second segment). The `context` is taken to be the entire word, except for the target segment. Thus, if lex is `kasi`, then context is `lex[:1]+'_'+lex[2:]`, or `k_si`. Note that no parameters are generated if the lexeme does not consist of exactly four segments.

```
>>> from nltk_lite.utilities import MinimalSet
>>> pos = 1
>>> ms = MinimalSet((lex[:pos] + '_' + lex[pos+1:], lex[pos], lex)
...                 for lex in lexemes if len(lex) == 4)
```

Now we print the table of minimal sets. We specify that each context was seen at least 3 times.

```
>>> for context in ms.contexts(3):
...     print context + ':',
...     for target in ms.targets():
...         print "%-4s" % ms.display(context, target, "-"),
...     print
k_si: kasi -    kesi kusi kosi
k_va: kava -    -    kuva kova
k_ru: karu kiru keru kuru koru
k_pu: kapu kipu -    -    kopu
k_ro: karo kiro -    -    koro
k_ri: kari kiri keri kuri kori
k_pa: kapa -    kepa -    kopa
k_ra: kara kira kera -    kora
k_ku: kaku -    -    kuku koku
k_ki: kaki kiki -    -    koki
```

Observe in the above example that the context, target, and displayed material were all based on the lexeme field. However, the idea of minimal sets is much more general. For instance, suppose we wanted to get a list of wordforms having more than one possible part-of-speech. Then the target will be part-of-speech field, and the context will be the lexeme field. We will also display the English gloss field.

```
>>> entries = [(e.findtext('lx'), e.findtext('ps'), e.findtext('ge'))
...                for e in lexicon
...                if e.findtext('lx') and e.findtext('ps') and e.findtext('ge')]
>>> ms = MinimalSet((lx, ps[0], "%s (%s)" % (ps[0], ge))
...           for (lx, ps, ge) in entries)
>>> for context in ms.contexts()[:10]:
...     print "%10s:" % context, "; ".join(ms.display_all(context))
  kokovara: N (unripe coconut); V (unripe)
     kapua: N (sore); V (have sores)
      koie: N (pig); V (get pig to eat)
      kovo: C (garden); N (garden); V (work)
    kavori: N (crayfish); V (collect crayfish or lobster)
    korita: N (cutlet?); V (dissect meat)
      keru: N (bone); V (harden like bone)
  kirokiro: N (bush used for sorcery); V (write)
    kaapie: N (hook); V (snag)
       kou: C (heap); V (lay egg)
```

The following program uses `MinimalSet` to find pairs of entries in the corpus which have different attachments based on the *verb* only.

```
>>> from nltk_lite.utilities import MinimalSet
>>> ms = MinimalSet()
>>> for entry in ppattach.dictionary('training'):
...     target  = entry['attachment']
...     context = (entry['noun1'], entry['prep'], entry['noun2'])
...     display = (target, entry['verb'])
...     ms.add(context, target, display)
>>> for context in ms.contexts():
...     print context, ms.display_all(context)
```

Here is one of the pairs found by the program.

(214)
```
received (NP offer) (PP from group)
rejected (NP offer (PP from group))
```

This finding gives us clues to a structural difference: the verb *receive* usually comes with two following arguments; we receive something *from* someone. In contrast, the verb *reject* only needs a single following argument; we can reject something without needing to say where it originated from.

13.4.5 Example Applications: Improving Access to Lexical Resources

A lexicon constructed as part of field-based research is a potential *language resource* for speakers of a language. Even when the language in question has a standard writing system, many speakers will not be literate in the language. They may be able to attempt an approximate spelling for a word, or they may prefer to access the dictionary via an index which uses the language of wider communication. In this section we deal with the first of these. The second is left to the reader as an exercise. We will also generate a wordfinder puzzle which can be used to test knowledge of lexical items.

Fuzzy Spelling (notes)

Confusible sets of segments: if two segments are confusible, map them to the same integer.

```
>>> group = {
...     ' ':0,                          # blank (for short words)
...     'p':1,  'b':1,  'v':1,          # labials
...     't':2,  'd':2,  's':2,          # alveolars
...     'l':3,  'r':3,                  # sonorant consonants
...     'i':4,  'e':4,                  # high front vowels
...     'u':5,  'o':5,                  # high back vowels
...     'a':6                           # low vowels
... }
```

Soundex: idea of a signature. Words with the same signature considered confusible. Consider first letter of a word to be so cognitively salient that people will not get it wrong.

```
>>> def soundex(word):
...     if len(word) == 0: return word  # sanity check
...     word += '    '                  # ensure word long enough
...     c0 = word[0].upper()
...     c1 = group[word[1]]
...     cons = filter(lambda x: x in 'pbvtdslr ', word[2:])
...     c2 = group[cons[0]]
...     c3 = group[cons[1]]
...     return "%s%d%d%d" % (c0, c1, c2, c3)
>>> print soundex('kalosavi')
K632
>>> print soundex('ti')
T400
```

Now we can build a soundex index of the lexicon:

```
>>> soundex_idx = {}
>>> for lex in lexemes:
...     code = soundex(lex)
...     if code not in soundex_idx:
```

```
...             soundex_idx[code] = set()
...         soundex_idx[code].add(lex)
```

We should sort these candidates by proximity with the target word.

```
>>> from nltk_lite.utilities import edit_dist
>>> def fuzzy_spell(target):
...     scored_candidates = []
...     code = soundex(target)
...     for word in soundex_idx[code]:
...         dist = edit_dist(word, target)
...         scored_candidates.append((dist, word))
...     scored_candidates.sort()
...     return [w for (d,w) in scored_candidates[:10]]
```

Finally, we can look up a word to get approximate matches:

```
>>> fuzzy_spell('kokopouto')
['kokopeoto', 'kokopuoto', 'kokepato', 'koovoto', 'koepato', 'kooupato', 'kopato', 'kopiito
>>> fuzzy_spell('kogou')
['kogo', 'koou', 'kokeu', 'koko', 'kokoa', 'kokoi', 'kokoo', 'koku', 'kooe', 'kooku']
```

Wordfinder Puzzle

Here we will generate a grid of letters, containing words found in the dictionary. First we remove any duplicates and disregard the order in which the lexemes appeared in the dictionary. We do this by converting it to a set, then back to a list. Then we select the first 200 words, and then only keep those words having a reasonable length.

```
>>> words = list(set(lexemes))
>>> words = words[:200]
>>> words = [w for w in words if 3 <= len(w) <= 12]
```

Now we generate the wordfinder grid, and print it out.

```
>>> from nltk_lite.misc.wordfinder import wordfinder
>>> grid, used = wordfinder(words)
>>> for i in range(len(grid)):
...     for j in range(len(grid[i])):
...         print grid[i][j],
...     print
O G H K U U V U V K U O R O V A K U N C
K Z O T O I S E K S N A I E R E P A K C
I A R A A K I O Y O V R S K A W J K U Y
L R N H N K R G V U K G I A U D J K V N
I I Y E A U N O K O O U K T R K Z A E L
A V U K O X V K E R V T I A A E R K R K
A U I U G O K U T X U I K N V V L I E O
R R K O K N U A J Z T K A K O O S U T R
I A U A U A S P V F O R O O K I C A O U
V K R R T U I V A O A U K V V S L P E K
A I O A I A K R S V K U S A A I X I K O
P S V I K R O E O A R E R S E T R O J X
O I I S U A G K R O R E R I T A I Y O A
```

```
R R R A T O O K O I K I W A K E A A R O
O E A K I K V O P I K H V O K K G I K T
K K L A K A A R M U G E P A U A V Q A I
O O O U K N X O G K G A R E A A P O O R
K V V P U J E T Z P K B E I E T K U R A
N E O A V A E O R U K B V K S Q A V U E
C E K K U K I K I R A E K O J I Q K K K
```

Finally we generate the words which need to be found.

```
>>> for i in range(len(used)):
...     print "%-12s" % used[i],
...     if float(i+1)%5 == 0: print
KOKOROPAVIRA KOROROVIVIRA KAEREASIVIRA KOTOKOTOARA  KOPUASIVIRA
KATAITOAREI  KAITUTUVIRA  KERIKERISI   KOKARAPATO   KOKOVURITO
KAUKAUVIRA   KOKOPUVIRA   KAEKAESOTO   KAVOVOVIRA   KOVAKOVARA
KAAREKOPIE   KAEPIEVIRA   KAPUUPIEPA   KOKORUUTO    KIKIRAEKO
KATAAVIRA    KOVOKOVOA    KARIVAITO    KARUVIRA     KAPOKARI
KUROVIRA     KITUKITU     KAKUPUTE     KAEREASI     KUKURIKO
KUPEROO      KAKAPUA      KIKISI       KAVORA       KIKIPI
KAPUA        KAARE        KOETO        KATAI        KUVA
KUSI         KOVO         KOAI
```

13.4.6 Generating Reports

Finally, we take a look at simple methods to generate summary reports, giving us an overall picture of the quality and organisation of the data.

First, suppose that we wanted to compute the average number of fields for each entry. This is just the total length of the entries (the number of fields they contain), divided by the number of entries in the lexicon:

```
>>> sum(len(entry) for entry in lexicon) / len(lexicon)
10
```

Next, let's consider some methods for discovering patterns in the lexical entries. Which fields are the most frequent?

```
>>> fd = FreqDist()
>>> for entry in lexicon:
...     for field in entry:
...         fd.inc(field.tag)
>>> fd.sorted_samples()[:10]
['ge', 'ex', 'xe', 'xp', 'gp', 'lx', 'ps', 'dt', 'rt', 'eng']
```

Which sequences of fields are the most frequent?

```
>>> fd = FreqDist()
>>> for entry in lexicon:
...     fd.inc(':'.join(field.tag for field in entry))
>>> top_ten = fd.sorted_samples()[:10]
>>> print '\n'.join(top_ten)
lx:rt:ps:ge:gp:dt:ex:xp:xe
lx:ps:ge:gp:dt:ex:xp:xe
lx:ps:ge:gp:dt:ex:xp:xe:ex:xp:xe
```

```
lx:rt:ps:ge:gp:dt:ex:xp:xe:ex:xp:xe
lx:ps:ge:gp:nt:dt:ex:xp:xe
lx:ps:ge:gp:dt
lx:ps:ge:ge:gp:dt:ex:xp:xe:ex:xp:xe
lx:rt:ps:ge:ge:gp:dt:ex:xp:xe:ex:xp:xe
lx:ps:ge:ge:gp:dt:ex:xp:xe
lx:rt:ps:ge:ge:gp:dt:ex:xp:xe
```

Which pairs of consecutive fields are the most frequent?

```
>>> fd = FreqDist()
>>> for entry in lexicon:
...     previous = "0"
...     for field in entry:
...         current = field.tag
...         fd.inc("%s->%s" % (previous, current))
...         previous = current
>>> fd.sorted_samples()[:10]
['ex->xp', 'xp->xe', '0->lx', 'ge->gp', 'ps->ge', 'dt->ex', 'lx->ps',
'gp->dt', 'xe->ex', 'lx->rt']
```

Once we have analyzed the sequences of fields, we might want to write down a grammar for lexical entries, and look for entries which do not conform to the grammar. In general, toolbox entries have nested structure. Thus they correspond to a tree over the fields. We can check for well-formedness by parsing the field names. We first set up a putative grammar for the entries:

```
>>> from nltk_lite import parse
>>> grammar = parse.cfg.parse_cfg('''
...   S -> Head "ps" Glosses Comment "dt" Examples
...   Head -> "lx" | "lx" "rt"
...   Glosses -> Gloss Glosses
...   Glosses ->
...   Gloss -> "ge" | "gp"
...   Examples -> Example Examples
...   Examples ->
...   Example -> "ex" "xp" "xe"
...   Comment -> "cmt"
...   Comment ->
...   ''')
>>> rd_parser = parse.RecursiveDescent(grammar)
```

Now we try to parse each entry. Those that are accepted by the grammar prefixed with a `'+'`, and those that are rejected are prefixed with a `'-'`.

```
>>> for entry in lexicon[10:20]:
...     marker_list = [field.tag for field in entry]
...     if rd_parser.get_parse_list(marker_list):
...         print "+", ':'.join(marker_list)
...     else:
...         print "-", ':'.join(marker_list)
- lx:ps:ge:gp:sf:nt:dt:ex:xp:xe:ex:xp:xe
- lx:rt:ps:ge:gp:nt:dt:ex:xp:xe:ex:xp:xe
- lx:ps:ge:gp:nt:dt:ex:xp:xe:ex:xp:xe
- lx:ps:ge:gp:nt:sf:dt
```

```
- lx:ps:ge:gp:dt:cmt:ex:xp:xe:ex:xp:xe
+ lx:ps:ge:ge:ge:gp:cmt:dt:ex:xp:xe
+ lx:rt:ps:ge:gp:cmt:dt:ex:xp:xe:ex:xp:xe
+ lx:rt:ps:ge:ge:gp:dt
- lx:rt:ps:ge:ge:ge:gp:dt:cmt:ex:xp:xe:ex:xp:xe:ex:xp:xe
+ lx:rt:ps:ge:gp:dt:ex:xp:xe
```

Looking at Timestamps

```
>>> fd = FreqDist()
>>> from string import split
>>> for entry in lexicon:
...     date = entry.findtext('dt')
...     if date:
...         (day, month, year) = split(date, '/')
...         fd.inc((month, year))
>>> for time in fd.sorted_samples():
...     print time[0], time[1], ':', fd.count(time)
Feb 2005 : 307
Dec 2004 : 151
Jan 2005 : 123
Feb 2004 : 64
Sep 2004 : 49
May 2005 : 46
Mar 2005 : 37
Apr 2005 : 29
Jul 2004 : 14
Nov 2004 : 5
Oct 2004 : 5
Aug 2004 : 4
May 2003 : 2
Jan 2004 : 1
May 2004 : 1
```

To put these in time order, we need to set up a special comparison function. Otherwise, if we just sort the months, we'll get them in alphabetical order.

```
>>> month_index = {
...     "Jan" : 1, "Feb" : 2,  "Mar" : 3,  "Apr" : 4,
...     "May" : 5, "Jun" : 6,  "Jul" : 7,  "Aug" : 8,
...     "Sep" : 9, "Oct" : 10, "Nov" : 11, "Dec" : 12
... }
>>> def time_cmp(a, b):
...     a2 = a[1], month_index[a[0]]
...     b2 = b[1], month_index[b[0]]
...     return cmp(a2, b2)
```

The comparison function says that we compare two times of the form `('Mar', '2004')` by reversing the order of the month and year, and converting the month into a number to get `('2004', '3')`, then using Python's built-in `cmp` function to compare them.

Now we can get the times found in the Toolbox entries, sort them according to our `time_cmp` comparison function, and then print them in order. This time we print bars to indicate frequency:

```
>>> times = fd.samples()
>>> times.sort(cmp=time_cmp)
>>> for time in times:
...     print time[0], time[1], ':', '#' * (1 + fd.count(time)/10)
May 2003 : #
Jan 2004 : #
Feb 2004 : #######
May 2004 : #
Jul 2004 : ##
Aug 2004 : #
Sep 2004 : #####
Oct 2004 : #
Nov 2004 : #
Dec 2004 : ################
Jan 2005 : #############
Feb 2005 : ##############################
Mar 2005 : ####
Apr 2005 : ###
May 2005 : #####
```

13.4.7 Exercises

1. ☼ Write a program to filter out just the date field (`dt`) without having to list the fields we wanted to retain.

2. ☼ Print an index of a lexicon. For each lexical entry, construct a tuple of the form `(gloss , lexeme)`, then sort and print them all.

3. ☼ What is the frequency of each consonant and vowel contained in lexeme fields?

4. ☼ How many entries were last modified in 2004?

5. ◑ Write a program that scans an HTML dictionary file to find entries having an illegal part-of-speech field, and reports the *headword* for each entry.

6. ◑ Write a program to find any parts of speech (`ps` field) that occurred less than ten times. Perhaps these are typing mistakes?

7. ◑ We saw a method for discovering cases of whole-word reduplication. Write a function to find words that may contain partial reduplication. Use the `re.search()` method, and the following regular expression: `(..+)\1`

8. ◑ We saw a method for adding a `cv` field. There is an interesting issue with keeping this up-to-date when someone modifies the content of the `lx` field on which it is based. Write a version of this program to add a `cv` field, replacing any existing `cv` field.

9. ◑ Write a program to add a new field `syl` which gives a count of the number of syllables in the word.

10. ◑ Write a function which displays the complete entry for a lexeme. When the lexeme is incorrectly spelled it should display the entry for the most similarly spelled lexeme.

11. ★ Obtain a comparative wordlist in CSV format, and write a program that prints those cognates having an edit-distance of at least three from each other.

12. ★ Build an index of those lexemes which appear in example sentences. Suppose the lexeme for a given entry is *w*. Then add a single cross-reference field `xrf` to this entry, referencing the headwords of other entries having example sentences containing w. Do this for all entries and save the result as a toolbox-format file.

13.5 Language Archives

Language technology and the linguistic sciences are confronted with a vast array of language resources, richly structured, large and diverse. Multiple communities depend on language resources, including linguists, engineers, teachers and actual speakers. Thanks to recent advances in digital technologies, we now have unprecedented opportunities to bridge these communities to the language resources they need. First, inexpensive mass storage technology permits large resources to be stored in digital form, while the Extensible Markup Language (XML) and Unicode provide flexible ways to represent structured data and ensure its long-term survival. Second, digital publication on the web is the most practical and efficient means of sharing language resources. Finally, a standard resource description model and interchange method provided by the Open Language Archives Community (OLAC) makes it possible to construct a *union catalog* over multiple repositories and archives (see `http://www.language-archives.org/`).

13.5.1 Managing Metadata for Language Resources

OLAC metadata extends the *Dublin Core* metadata set with descriptors that are important for language resources.

The container for an OLAC metadata record is the element `<olac>`. Here is a valid OLAC metadata record from the Pacific And Regional Archive for Digital Sources in Endangered Cultures (PARADISEC):

```
<olac:olac xsi:schemaLocation="http://purl.org/dc/elements/1.1/ http://www.lang
  http://purl.org/dc/terms/ http://www.language-archives.org/OLAC/1.0/dcterms.x
  http://www.language-archives.org/OLAC/1.0/ http://www.language-archives.org/O
  <dc:title>Tiraq Field Tape 019</dc:title>
  <dc:identifier>AB1-019</dc:identifier>
  <dcterms:hasPart>AB1-019-A.mp3</dcterms:hasPart>
  <dcterms:hasPart>AB1-019-A.wav</dcterms:hasPart>
  <dcterms:hasPart>AB1-019-B.mp3</dcterms:hasPart>
  <dcterms:hasPart>AB1-019-B.wav</dcterms:hasPart>
  <dc:contributor xsi:type="olac:role" olac:code="recorder">Brotchie, Amanda</d
  <dc:subject xsi:type="olac:language" olac:code="x-sil-MME"/>
  <dc:language xsi:type="olac:language" olac:code="x-sil-BCY"/>
  <dc:language xsi:type="olac:language" olac:code="x-sil-MME"/>
  <dc:format>Digitised: yes;</dc:format>
  <dc:type>primary_text</dc:type>
  <dcterms:accessRights>standard, as per PDSC Access form</dcterms:accessRights
  <dc:description>SIDE A<p>1. Elicitation Session - Discussion and
    translation of Lise's and Marie-Claire's Songs and Stories from
    Tape 18 (Tamedal)<p><p>SIDE B<p>1. Elicitation Session: Discussion
```

```
    of and translation of Lise's and Marie-Clare's songs and stories
    from Tape 018 (Tamedal)<p>2. Kastom Story 1 - Bislama
    (Alec). Language as given: Tiraq</dc:description>
</olac:olac>
```

Note

The remainder of this section will discuss how to manipulate OLAC metadata.

13.6 Linguistic Annotation

Note

to be written

13.7 Further Reading

Bird, Steven (1999). Multidimensional exploration of online linguistic field data *Proceedings of the 29th Meeting of the North-East Linguistic Society*, pp 33-50.

Bird, Steven and Gary Simons (2003). Seven Dimensions of Portability for Language Documentation and Description, *Language* 79: 557-582.

About this document...

Part IV

APPENDICES

Appendix A

Appendix: Regular Expressions

This section explores regular expressions in detail, with examples drawn from language processing. It builds on the brief overview given in the introductory programming chapter.

We have already noted that a text can be viewed as a string of characters. What kinds of processing are performed at the character level? Perhaps word games are the most familiar example of such processing. In completing a crossword we may want to know which 3-letter English words end with the letter `c` (e.g. `arc`). We might want to know how many words can be formed from the letters: `a`, `c`, `e`, `o`, and `n` (e.g. `ocean`). We may want to find out which unique English word contains the substring `gnt` (left as an exercise for the reader). In all these examples, we are considering which word - drawn from a large set of candidates - matches a given pattern. To put this in a more computational framework, we could imagine searching through a large digital corpus in order to find all words that match a particular pattern. There are many serious uses of this so-called *pattern matching*.

One instructive example is the task of finding all doubled words in a text; an example would be the string `for for example`. Notice that we would be particularly interested in finding cases where the words were split across a linebreak (in practice, most erroneously doubled words occur in this context). Consequently, even with such a relatively banal task, we need to be able to describe patterns which refer not just to "ordinary" characters, but also to formatting information.

There are conventions for indicating structure in strings, also known as *formatting*. For example, there are a number of alternative ways of formatting a "date string", such as `23/06/2002`, `6/23/02`, or `2002-06-23`. Whole texts may be formatted, such as an email message which contains header fields followed by the message body. Another familiar form of formatting involves visual structure, such as tabular format and bulleted lists.

Finally, texts may contain explicit "markup", such as `<abbrev>Phil</abbrev>`, which provides information about the interpretation or presentation of some piece of text. To summarize, in language processing, strings are ubiquitous, and they often contain important structure.

So far we have seen elementary examples of pattern matching, the matching of individual characters. More often we are interested in matching *sequences* of characters. For example, part of the operation of a naive spell-checker could be to remove a word-final `s` from a suspect word token, in case the word is a plural, and see if the putative singular form exists in the dictionary. For this we must locate `s` and remove it, but only if it precedes a word boundary. This requires matching a pattern consisting of two characters.

Beyond this pattern matching on the *content* of a text, we often want to process the *formatting* and *markup* of a text. We may want to check the formatting of a document (e.g. to ensure that every sentence begins with a capital letter) or to reformat a document (e.g. replacing sequences of space

characters with a single space). We may want to find all date strings and extract the year. We may want to extract all words contained inside the `<abbrev> </abbrev>` markup in order to construct a list of abbreviations.

Processing the content, format and markup of strings is a central task in most kinds of NLP. The most widespread method for string processing uses *regular expressions*.

A.1 Simple Regular Expressions

In this section we will see the building blocks for simple regular expressions, along with a selection of linguistic examples. We can think of a regular expression as *a specialised notation for describing patterns that we want to match*. In order to make explicit when we are talking about a pattern *patt*, we will use the notation «`patt`». The first thing to say about regular expressions is that most letters match themselves. For example, the pattern «`sing`» exactly matches the string `sing`. In addition, regular expressions provide us with a set of *special characters*[2] which give us a way to match *sets of strings*, and we will now look at these.

A.1.1 The Wildcard

The "`.`" symbol is called a *wildcard*: it matches any single character. For example, the regular expression «`s.ng`» matches the following English words: `sang`, `sing`, `song`, and `sung`. Note that «`.`» will match not only alphabetic characters, but also numeric and whitespace characters. Consequently, «`s.ng`» will also match non-words such as `s3ng`.

We can also use the wildcard symbol for counting characters. For instance «`....zy`» matches six-letter strings that end in `zy`. The pattern «`....berry`» finds words like `cranberry`. In our text from Wall Street Journal below, the pattern «`t...`» will match the words `that` and `term`, and will also match the word sequence `to a` (since the third "`.`" in the pattern can match the space character):

```
Paragraph 12 from wsj_0034:
It's probably worth paying a premium for funds that invest in markets
that are partially closed to foreign investors, such as South Korea,
some specialists say.  But some European funds recently have
skyrocketed; Spain Fund has surged to a startling 120% premium.  It has
been targeted by Japanese investors as a good long-term play tied to
1992's European economic integration.  And several new funds that aren't
even fully invested yet have jumped to trade at big premiums.

"I'm very alarmed to see these rich valuations," says Smith Barney's
Mr. Porter.
```

> **Note**
>
> Note that the wildcard matches *exactly* one character, and must be repeated for as many characters as should be matched. To match a variable number of characters we must use notation for *optionality*.

We can see exactly where a regular expression matches against a string using NLTK's `re_show` function. Readers are encouraged to use `re_show` to explore the behaviour of regular expressions.

[4] These are often called *metacharacters*; that is, characters which express properties of (ordinary) characters.

```
>>> from nltk_lite.utilities import re_show
>>> string = """
... It's probably worth paying a premium for funds that invest in markets
... that are partially closed to foreign investors, such as South Korea, ...
... """
>>> re_show('t...', string)
I{t's }probably wor{th p}aying a premium for funds {that} inves{t in} markets
{that} are par{tial}ly closed {to f}oreign inves{tors}, such as Sou{th K}orea, ...
```

A.1.2 Optionality and Repeatability

The "?" symbol indicates that the immediately preceding regular expression is optional. The regular expression «colou?r» matches both British and American spellings, colour and color. The expression that precedes the ? may be punctuation, such as an optional hyphen. For instance «e-?mail» matches both e-mail and email.

The "+" symbol indicates that the immediately preceding expression is repeatable, up to an arbitrary number of times. For example, the regular expression «coo+l» matches cool, coool, and so on. This symbol is particularly effective when combined with the . symbol. For example, «f.+f» matches all strings of length greater than two, that begin and end with the letter f (e.g. foolproof). The expression «.+ed» finds strings that potentially have the past-tense -ed suffix.

The "*" symbol indicates that the immediately preceding expression is both optional and repeatable. For example «.*gnt.*» matches all strings that contain gnt.

Occasionally we need to match material that spans a linebreak. For example, we may want to strip out the HTML markup from a document. To do this we must delete material between angle brackets. The most obvious expression is: «<.*>». However, this has two problems: it will not match an HTML tag that contains a linebreak, and the «.*» will consume as much material as possible (including the > character). To permit matching over a linebreak we must use Python's DOTALL flag, and to ensure that the > matches against the first instance of the character we must do non-greedy matching using *?:

```
>>> import re
>>> text = """one two three <font
...           color="red">four</font> five"""
>>> re.sub(r'<.*?>', ' ', text, re.DOTALL)
```

A.1.3 Choices

Patterns using the wildcard symbol are very effective, but there are many instances where we want to limit the set of characters that the wildcard can match. In such cases we can use the [] notation, which enumerates the set of characters to be matched - this is called a *character class*. For example, we can match any English vowel, but no consonant, using «[aeiou]». Note that this pattern can be interpreted as saying "match a or e or ... or u"; that is, the pattern resembles the wildcard in only matching a string of length one; unlike the wildcard, it restricts the characters matched to a specific class (in this case, the vowels). Note that the order of vowels in the regular expression is insignificant, and we would have had the same result with the expression «[uoiea]». As a second example, the expression «p[aeiou]t» matches the words: pat, pet, pit, pot, and put.

We can combine the [] notation with our notation for repeatability. For example, expression «p[aeiou]+t» matches the words listed above, along with: peat, poet, and pout.

Often the choices we want to describe cannot be expressed at the level of individual characters. As discussed in the tagging tutorial, different parts of speech are often *tagged* using labels from a tagset. In

the Brown tagset, for example, singular nouns have the tag `NN1`, while plural nouns have the tag `NN2`, while nouns which are unspecified for number (e.g., `aircraft`) are tagged `NN0`. So we might use «`NN.*`» as a pattern which will match any nominal tag. Now, suppose we were processing the output of a tagger to extract string of tokens corresponding to noun phrases, we might want to find all nouns (`NN.*`), adjectives (`JJ.*`) and determiners (`DT`), while excluding all other word types (e.g. verbs `VB.*`). It is possible, using a single regular expression, to search for this set of candidates using the *choice operator* "`|`" as follows: «`NN.*|JJ.*|DT`». This says: match `NN.*` *or* `JJ.*` *or* `DT`.

As another example of multi-character choices, suppose that we wanted to create a program to simplify English prose, replacing rare words (like `abode`) with a more frequent, synonymous word (like `home`). In this situation, we need to map from a potentially large set of words to an individual word. We can match the set of words using the choice operator. In the case of the word `home`, we would want to match the regular expression «`dwelling|domicile|abode`».

> **Note**
>
> Note that the choice operator has wide scope, so that «`abc|def`» is a choice between `abd` and `def`, and not between `abced` and `abdef`. The latter choice must be written using parentheses: «`ab(c|d)ed`».

A.2 More Complex Regular Expressions

In this section we will cover operators which can be used to construct more powerful and useful regular expressions.

A.2.1 Ranges

Earlier we saw how the `[]` notation could be used to express a set of choices between individual characters. Instead of listing each character, it is also possible to express a *range* of characters, using the `-` operator. For example, «`[a-z]`» matches any lowercase letter. This allows us to avoid the overpermissive matching we noted above with the pattern «`t...`». If we were to use the pattern «`t[a-z][a-z][a-z]`», then we would no longer match the two word sequence `to a`.

As expected, ranges can be combined with other operators. For example «`[A-Z][a-z]*`» matches words that have an initial capital letter followed by any number of lowercase letters. The pattern «`20[0-4][0-9]`» matches year expressions in the range 2000 to 2049.

Ranges can be combined, e.g. «`[a-zA-Z]`» which matches any lowercase or uppercase letter. The expression «`[b-df-hj-np-tv-z]+`» matches words consisting only of consonants (e.g. `pygmy`).

A.2.2 Complementation

We just saw that the character class «`[b-df-hj-np-tv-z]+`» allows us to match sequences of consonants. However, this expression is quite cumbersome. A better alternative is to say: let's match anything which isn't a vowel. To do this, we need a way of expressing *complementation*. We do this using the symbol "`^`" as the first character inside a class expression `[]`. Let's look at an example. The regular expression « `[^aeiou]` » is just like our earlier character class «`[aeiou]`», except now the set of vowels is preceded by `^`. The expression as a whole is interpreted as matching anything which *fails* to match «`[aeiou]`». In other words, it matches all lowercase consonants (plus all uppercase letters and non-alphabetic characters).

As another example, suppose we want to match any string which is enclosed by the HTML tags for boldface, namely <B> and </B>. We might try something like this: «<B>.*</B>». This would successfully match <B>important</B>, but would also match <B>important</B> and <B>urgent</B>, since the «.*» subpattern will happily match all the characters from the end of important to the end of urgent. One way of ensuring that we only look at matched pairs of tags would be to use the expression «<B>[^<]*</B>», where the chararacter class matches anything other than a left angle bracket.

Finally, note that character class complementation also works with ranges. Thus «[^a-z]» matches anything other than the lower case alphabetic characters a through z.

A.2.3 Common Special Symbols

So far, we have only looked at patterns which match with the content of character strings. However, it is also useful to be able to refer to formatting properties of texts. Two important symbols in this regard are "^" and "$" which are used to *anchor* matches to the beginnings or ends of lines in a file.

> **Note**
>
> "^" has two quite distinct uses: it is interpreted as complementation when it occurs as the first symbol within a character class, and as matching the beginning of lines when it occurs elsewhere in a pattern.

For example, suppose we wanted to find all the words that occur at the beginning of lines in the WSJ text above. Our first attempt might look like «^[A-Za-z]+». This says: starting at the beginning of a line, look for one or more alphabetic characters (upper or lower case), followed by a space. This will match the words that, some, been, and even. However, it fails to match It's, since ' isn't an alphabetic character. A second attempt might be «^[^]+», which says to match any string starting at the beginning of a line, followed by one or more characters which are *not* the space character, followed by a space. This matches all the previous words, together with It's, skyrocketed, 1992s, I'm and "Mr.. As a second example, «[a-z]*s$» will match words ending in s that occur at the end of a line. Finally, consider the pattern «^$»; this matches strings where no character occurs between the beginning and the end of a line - in other words, empty lines!

As we have seen, special characters like ".", "*", "+" and "$" give us powerful means to generalise over character strings. But suppose we wanted to match against a string which itself contains one or more special characters? An example would be the arithmetic statement $5.00 * ($3.05 + $0.85). In this case, we need to resort to the so-called *escape* character "\" ("backslash"). For example, to match a dollar amount, we might use «\$[1-9][0-9]*\.[0-9][0-9]». The same goes for matching other special characters.

Special Sequences	
\b	Word boundary (zero width)
\d	Any decimal digit (equivalent to [0-9])
\D	Any non-digit character (equivalent to [^0-9])
\s	Any whitespace character (equivalent to [\t\n\r\f\v]
\S	Any non-whitespace character (equivalent to [^ \t\n\r\f\v])
\w	Any alphanumeric character (equivalent to [a-zA-Z0-9_])
\W	Any non-alphanumeric character (equivalent to [^a-zA-Z0-9_])

Table A.1:

A.3 Python Interface

The Python `re` module provides a convenient interface to an underlying regular expression engine. The module allows a regular expression pattern to be compiled into a object whose methods can then be called.

In the next example, we assume that we have a local copy (i.e., `words`) of the Unix dictionary, which may be found in the NLTK `data/words` directory. This file contains over 400,000 words, one per line.

```
>>> from re import *
>>> from nltk_lite.corpora import words
```

Next we read in the list of words and count them:

```
>>> wordlist = list(words.raw())
>>> len(wordlist)
45378
```

Now we can compile a regular expression for words containing a sequence of two 'a's and find the matches:

```
>>> r1 = compile('.*aa.*')
>>> [w for w in wordlist if r1.match(w)]
['Afrikaans', 'bazaar', 'bazaars', 'Canaan', 'Haag', 'Haas', 'Isaac', 'Isaacs', 'Is
```

Suppose now that we want to find all three-letter words ending in the letter "`c`". Our first attempt might be as follows:

```
>>> r1 = compile('..c')
>>> [w for w in wordlist if r1.match(w)][:10]
['accede', 'acceded', 'accedes', 'accelerate', 'accelerated', 'accelerates', 'accel
```

The problem is that we have matched words containing three-letter sequences ending in "`c`" which occur *anywhere within a word*. For example, the pattern will match "`c`" in words like `aback`, `Aerobacter` and `albacore`. Instead, we must revise our pattern so that it is anchored to the beginning and ends of the word: «`^...$`»:

```
>>> r2 = compile('^..c$')
>>> [w for w in wordlist if r2.match(w)]
['arc', 'Doc', 'Lac', 'Mac', 'Vic']
```

In the section on complementation, we briefly looked at the task of matching strings which were enclosed by HTML markup. Our first attempt is illustrated in the following code example, where we incorrectly match the whole string, rather than just the substring "`<B>important</B>`".

```
>>> html = '<B>important</B> and <B>urgent</B>'
>>> r2 = compile('<B>.*</B>')
>>> print r2.findall(html)
['<B>important</B> and <B>urgent</B>']
```

As we pointed out, one solution is to use a character class which matches with the complement of "<":

```
>>> r4 = compile('<B>[^<]*</B>')
>>> print r4.findall(html)
['<B>important</B>', '<B>urgent</B>']
```

However, there is another way of approaching this problem. «`<B>.*</B>`» gets the wrong results because the «`*`» operator tries to consume as much input as possible. That is, the matching is said to be *greedy*. In the current case, «`*`» matches everything after the first `<B>`, including the following `</B>` and `<B>`. If we instead use the non-greedy star operator «`*?`», we get the desired match, since «`*?`» tries to consume as little input as possible.

A.3.1 Exercises

2. *Pig Latin* is a simple transliteration of English: words starting with a vowel have `way` appended (e.g. `is` becomes `isway`); words beginning with a consonant have all consonants up to the first vowel moved to the end of the word, and then `ay` is appended (e.g. `start` becomes `artstay`).

   a) Write a program to convert English text to Pig Latin.

   b) Extend the program to convert text, instead of individual words.

   c) Extend it further to preserve capitalisation, to keep `qu` together (i.e. so that `quiet` becomes `ietquay`), and to detect when `y` is used as a consonant (e.g. `yellow`) vs a vowel (e.g. `style`).

3. An interesting challenge for tokenisation is words that have been split across a linebreak. E.g. if *long-term* is split, then we have the string `long-\nterm`.

   a) Write a regular expression that identifies words that are hyphenated at a linebreak. The expression will need to include the `\n` character.

   b) Use `re.sub()` to remove the `\n` character from these words.

4. Write a utility function that takes a URL as its argument, and returns the contents of the URL, with all HTML markup removed. Use `urllib.urlopen` to access the contents of the URL, e.g. `raw_contents = urllib.urlopen('http://nltk.sourceforge.net/').read()`.

5. Write a program to guess the number of syllables from the orthographic representation of words (e.g. English text).

6. Download some text from a language that has vowel harmony (e.g. Hungarian), extract the vowel sequences of words, and create a vowel bigram table.

7. Obtain a pronunciation lexicon, and try generating nonsense rhymes.

A.4 Further Reading

A.M. Kuchling. *Regular Expression HOWTO*, http://www.amk.ca/python/howto/regex/

About this document...

Appendix B

Appendix: NLP in Python vs other Programming Languages

Many programming languages have been used for NLP. As we will explain in more detail in the introductory chapter, we have chosen Python because we believe it is well-suited to the special requirements of NLP. Here we present a brief survey of several programming languages, for the simple task of reading a text and printing the words that end with `ing`. We begin with the Python version, which we believe is readily interpretable, even by non Python programmers:

```
import sys
for line in sys.stdin:
    for word in line.split():
        if word.endswith('ing'):
            print word
```

Like Python, Perl is a scripting language. However, its syntax is obscure. For instance, it is difficult to guess what kind of entities are represented by: `<>`, `$`, `my`, and `split`, in the following program:

```
while (<>) {
    foreach my $word (split) {
        if ($word =~ /ing$/) {
            print "$word\n";
        }
    }
}
```

We agree that "it is quite easy in Perl to write programs that simply look like raving gibberish, even to experienced Perl programmers" (Hammond 2003:47). Having used Perl ourselves in research and teaching since the 1980s, we have found that Perl programs of any size are inordinately difficult to maintain and re-use. Therefore we believe Perl is no longer a particularly suitable choice of programming language for linguists or for language processing.

Prolog is a logic programming language which has been popular for developing natural language parsers and feature-based grammars, given the inbuilt support for search and the *unification* operation which combines two feature structures into one. Unfortunately Prolog is not easy to use for string processing or input/output, as the following program code demonstrates for our linguistic example:

```
main :-
    current_input(InputStream),
```

```
    read_stream_to_codes(InputStream, Codes),
    codesToWords(Codes, Words),
    maplist(string_to_list, Words, Strings),
    filter(endsWithIng, Strings, MatchingStrings),
    writeMany(MatchingStrings),
    halt.

codesToWords([], []).
codesToWords([Head | Tail], Words) :-
    ( char_type(Head, space) ->
        codesToWords(Tail, Words)
    ;
        getWord([Head | Tail], Word, Rest),
        codesToWords(Rest, Words0),
        Words = [Word | Words0]
    ).

getWord([], [], []).
getWord([Head | Tail], Word, Rest) :-
    (
        ( char_type(Head, space) ; char_type(Head, punct) )
    ->  Word = [], Tail = Rest
    ;   getWord(Tail, Word0, Rest), Word = [Head | Word0]
    ).

filter(Predicate, List0, List) :-
    ( List0 = [] -> List = []
    ;   List0 = [Head | Tail],
        ( apply(Predicate, [Head]) ->
            filter(Predicate, Tail, List1),
            List = [Head | List1]
        ;   filter(Predicate, Tail, List)
        )
    ).

endsWithIng(String) :- sub_string(String, _Start, _Len, 0, 'ing').

writeMany([]).
writeMany([Head | Tail]) :- write(Head), nl, writeMany(Tail).
```

Java is an object-oriented language incorporating native support for the Internet, that was originally designed to permit the same executable program to be run on most computer platforms. Java has replaced COBOL as the standard language for business enterprise software:

```
import java.io.*;
public class IngWords {
    public static void main(String[] args) {
        BufferedReader in = new BufferedReader(new
          InputStreamReader(
                 System.in));
        String line = in.readLine();
        while (line != null) {
            for (String word : line.split(" ")) {
```

```
                if (word.endsWith("ing"))
                    System.out.println(word);
            }
            line = in.readLine();
        }
    }
}
```

The C programming language is a highly-efficient low-level language that is popular for operating system and networking software:

```
#include <sys/types.h>
#include <regex.h>
#include <stdio.h>
#define BUFFER_SIZE 1024

int main(int argc, char **argv) {
     regex_t space_pat, ing_pat;
     char buffer[BUFFER_SIZE];
     regcomp(&space_pat, "[, \t\n]+", REG_EXTENDED);
     regcomp(&ing_pat, "ing$", REG_EXTENDED | REG_ICASE);

     while (fgets(buffer, BUFFER_SIZE, stdin) != NULL) {
         char *start = buffer;
         regmatch_t space_match;
         while (regexec(&space_pat, start, 1, &space_match, 0) == 0) {
             if (space_match.rm_so > 0) {
                 regmatch_t ing_match;
                 start[space_match.rm_so] = '\0';
                 if (regexec(&ing_pat, start, 1, &ing_match, 0) == 0)
                     printf("%s\n", start);
             }
             start += space_match.rm_eo;
         }
     }
     regfree(&space_pat);
     regfree(&ing_pat);

     return 0;
}
```

LISP is a so-called functional programming language, in which all objects are lists, and all operations are performed by (nested) functions of the form `(function arg1 arg2 ...)`. Many of the earliest NLP systems were implemented in LISP:

```
(defpackage "REGEXP-TEST" (:use "LISP" "REGEXP"))
(in-package "REGEXP-TEST")

(defun has-suffix (string suffix)
  "Open a file and look for words ending in _ing."
  (with-open-file (f string)
     (with-loop-split (s f " ")
        (mapcar #'(lambda (x) (has_suffix suffix x)) s))))
```

```
(defun has_suffix (suffix string)
  (let* ((suffix_len (length suffix))
   (string_len (length string))
    (base_len (- string_len suffix_len)))
    (if (string-equal suffix string :start1 0 :end1 NIL :start2 base_len :end2
        (print string))))

(has-suffix "test.txt" "ing")
```

Ruby is a more recently developed scripting language than Python, best known for its convenient web application framework, *Ruby on Rails*. Here are two Ruby programs for finding words ending in *ing*

```
ARGF.each { |line|
  line.split.find_all { |word|
    word.match(/ing$/)
  }.each { |word|
    puts word
  }
}
```

```
for line in ARGF
  for word in line.split
    if word.match(/ing$/) then
      puts word
    end
  end
end
```

Haskell is another functional programming language which permits a much more compact (but incomprehensible) solution of our simple task:

```
module Main
  where main = interact (unlines.(filter ing).(map (filter isAlpha)).words)
    where ing = (=="gni").(take 3).reverse
```

(We are grateful to the following people for furnishing us with these program samples: Tim Baldwin, Trevor Cohn, Rod Farmer, Aaron Harnly, Edward Ivanovic, Olivia March, and Lars Yencken.)

Appendix C

Appendix: NLTK Modules and Corpora

Corpora and Corpus Samples Distributed with NLTK (starred items with NLTK-Lite)		
Corpus	**Compiler**	**Contents**
Brown Corpus	Francis, Kucera	15 genres, 1.15M words, tagged
CoNLL 2000 Chunking Data	Tjong Kim Sang	270k words, tagged and chunked
Genesis Corpus	Misc web sources	6 texts, 200k words, 6 languages
Project Gutenberg (sel)	Hart, Newby, et al	14 texts, 1.7M words
NIST 1999 Info Extr (sel)	Garofolo	63k words, newswire and named-entity SGML markup
Lexicon Corpus		Words, tags and frequencies from Brown Corpus and WSJ
Names Corpus	Kantrowitz, Ross	8k male and female names
PP Attachment Corpus	Ratnaparkhi	28k prepositional phrases, tagged as noun or verb modifiers
Presidential Addresses	Ahrens	485k words, formatted text
Roget's Thesaurus	Project Gutenberg	200k words, formatted text
SEMCOR	Rus, Mihalcea	880k words, part-of-speech and sense tagged
SENSEVAL 2 Corpus	Ted Pedersen	600k words, part-of-speech and sense tagged
Stopwords Corpus	Porter et al	2,400 stopwords for 11 languages
Penn Treebank (sel)	LDC	40k words, tagged and parsed
TIMIT Corpus (sel)	NIST/LDC	audio files and transcripts for 16 speakers
Wordlist Corpus	OpenOffice.org et al	960k words and 20k affixes for 8 languages

Table C.1:

Appendix D

Appendix: Python and NLTK Cheat Sheet

D.1 Python

D.1.1 Strings

```
>>> x = 'Python'; y = 'NLTK'; z = 'Natural Language Processing'
>>> x + '/' + y
'Python/NLTK'
>>> 'LT' in y
True
>>> x[2:]
'thon'
>>> x[::-1]
'nohtyP'
>>> len(x)
6
>>> z.count('a')
4
>>> z.endswith('ing')
True
>>> z.index('Language')
8
>>> '; '.join([x,y,z])
'Python; NLTK; Natural Language Processing'
>>> y.lower()
'nltk'
>>> z.replace(' ', '\n')
'Natural\nLanguage\nProcessing'
>>> print z.replace(' ', '\n')
Natural
Language
Processing
>>> z.split()
['Natural', 'Language', 'Processing']
```

For more information, type *help(str)* at the Python prompt.

D.1.2 Lists

```
>>> x = ['Natural', 'Language']; y = ['Processing']
>>> x[0]
'Natural'
>>> list(x[0])
['N', 'a', 't', 'u', 'r', 'a', 'l']
>>> x + y
['Natural', 'Language', 'Processing']
>>> 'Language' in x
True
>>> len(x)
2
>>> x.index('Language')
1
```

The following functions modify the list in-place:

```
>>> x.append('Toolkit')
>>> x
['Natural', 'Language', 'Toolkit']
>>> x.insert(0, 'Python')
>>> x
['Python', 'Natural', 'Language', 'Toolkit']
>>> x.reverse()
>>> x
['Toolkit', 'Language', 'Natural', 'Python']
>>> x.sort()
>>> x
['Language', 'Natural', 'Python', 'Toolkit']
```

For more information, type *help(list)* at the Python prompt.

D.1.3 Dictionaries

```
>>> d = {'natural': 'adj', 'language': 'noun'}
>>> d['natural']
'adj'
>>> d['toolkit'] = 'noun'
>>> d
{'natural': 'adj', 'toolkit': 'noun', 'language': 'noun'}
>>> 'language' in d
True
>>> d.items()
[('natural', 'adj'), ('toolkit', 'noun'), ('language', 'noun')]
>>> d.keys()
['natural', 'toolkit', 'language']
>>> d.values()
['adj', 'noun', 'noun']
```

For more information, type *help(dict)* at the Python prompt.

D.1.4 Regular Expressions

Note

to be written

D.2 NLTK

D.2.1 Tokenization

```
>>> text = '''NLTK, the Natural Language Toolkit, is a suite of program
... modules, data sets and tutorials supporting research and teaching in
... computational linguistics and natural language processing.'''
>>> from nltk_lite import tokenize
>>> list(tokenize.line(text))
['NLTK, the Natural Language Toolkit, is a suite of program', 'modules,
data sets and tutorials supporting research and teaching in', 'computational
linguistics and natural language processing.']
>>> list(tokenize.whitespace(text))
['NLTK,', 'the', 'Natural', 'Language', 'Toolkit,', 'is', 'a', 'suite',
 'of', 'program', 'modules,', 'data', 'sets', 'and', 'tutorials',
 'supporting', 'research', 'and', 'teaching', 'in', 'computational',
 'linguistics', 'and', 'natural', 'language', 'processing.']
>>> list(tokenize.wordpunct(text))
['NLTK', ',', 'the', 'Natural', 'Language', 'Toolkit', ',', 'is', 'a',
 'suite', 'of', 'program', 'modules', ',', 'data', 'sets', 'and',
 'tutorials', 'supporting', 'research', 'and', 'teaching', 'in',
 'computational', 'linguistics', 'and', 'natural', 'language',
 'processing', '.']
>>> list(tokenize.regexp(text, ', ', gaps=True))
['NLTK', 'the Natural Language Toolkit', 'is a suite of program\nmodules',
 'data sets and tutorials supporting research and teaching in\ncomputational
 linguistics and natural language processing.']
```

D.2.2 Stemming

```
>>> tokens = list(tokenize.wordpunct(text))
>>> from nltk_lite import stem
>>> stemmer = stem.Regexp('ing$|s$|e$')
>>> for token in tokens:
...     print stemmer.stem(token),
NLTK , th Natural Languag Toolkit , i a suit of program module ,
data set and tutorial support research and teach in computational
linguistic and natural languag process .
>>> stemmer = stem.Porter()
>>> for token in tokens:
...     print stemmer.stem(token),
NLTK , the Natur Languag Toolkit , is a suit of program modul ,
data set and tutori support research and teach in comput linguist
and natur languag process .
```

D.2.3 Tagging

Note

to be written

Index

Bibliography

[Abney, 1996a] Abney, S. (1996a). Part-of-speech tagging and partial parsing. In Church, K., Young, S., and Bloothooft, G., editors, *Corpus-Based Methods in Language and Speech*. Kluwer Academic Publishers, Dordrecht.

[Abney, 1996b] Abney, S. (1996b). Statistical methods and linguistics. In Klavans, J. and Resnik, P., editors, *The Balancing Act: Combining Symbolic and Statistical Approaches to Language*. The MIT Press.

[Blackburn and Bos, 2005] Blackburn, P. and Bos, J. (2005). *Representation and Inference for Natural Language: A First Course in Computational Semantics*. CSLI Publications, Stanford, Ca.

[Brent and Cartwright, 1995] Brent, M. and Cartwright, T. (1995). Distributional regularity and phonotactic constraints are useful for segmentation. In Brent, M., editor, *Computational Approaches to Language Acquisition*. MIT Press.

[Bresnan and Hay, 2006] Bresnan, J. and Hay, J. (2006). Gradient grammar: An effect of animacy on the syntax of *give* in varieties of English. unpublished ms.

[Budanitsky and Hirst, 2006] Budanitsky, A. and Hirst, G. (2006). Evaluating wordnet-based measures of lexical semantic relatedness. *Computational Linguistics*, 32(1):13–48.

[Burton-Roberts, 1997] Burton-Roberts, N. (1997). *Analysing Sentences*. Longman.

[Carpenter, 1992] Carpenter, B. (1992). *The Logic of Typed Feature Structures*. Cambridge University Press, Cambridge, England.

[Carpenter, 1997] Carpenter, B. (1997). *Type-Logical Semantics*. The MIT Press.

[Chomsky, 1970] Chomsky, N. (1970). Remarks on nominalization. In Jacobs, R. and Rosenbaum, P., editors, *Readings in English Transformational Grammar*. Blaisdell, Waltham, MA.

[Chomsky and Halle, 1968] Chomsky, N. and Halle, M. (1968). *The Sound Pattern of English*. New York: Harper and Row.

[Church and Patil, 1982] Church, K. and Patil, R. (1982). Coping with syntactic ambiguity or how to put the block in the box on the table. *American Journal of Computational Linguistics*, 8(3–4):139–149.

[Cole, 1997] Cole, R., editor (1997). *Survey of the State of the Art in Human Language Technology*. Studies in Natural Language Processing. Cambridge University Press. http://cslu.cse.ogi.edu/HLTsurvey/.

[Copestake, 2002] Copestake, A. (2002). *Implementing Typed Feature Structure Grammars*. CSLI Publications, Stanford, CA.

[Dale et al., 2000] Dale, R., Moisl, H., and Somers, H., editors (2000). *Handbook of Natural Language Processing*. Marcel Dekker.

[Dowty et al., 1981] Dowty, D. R., Wall, R. E., and Peters, S. (1981). *Introduction to Montague Semantics*. Kluwer Academic Publishers.

[Earley, 1970] Earley, J. (1970). An efficient context-free parsing algorithm. *Communications of the Association for Computing Machinery*, 13(2):94–102.

[Emele and Zajac, 1990] Emele, M. C. and Zajac, R. (1990). Typed unification grammars. In *Proceedings of the 13th Conference on Computational linguistics*, pages 293–298, Morristown, NJ, USA. Association for Computational Linguistics.

[Gazdar et al., 1985] Gazdar, G., Klein, E., Pullum, G., and (1985), I. S. (1985). *Generalized Phrase Structure Grammar*. Basil Blackwell.

[Harel, 2004] Harel, D. (2004). *Algorithmics: The Spirit of Computing*. Addison Wesley.

[Heim and Kratzer, 1998] Heim, I. and Kratzer, A. (1998). *Semantics in Generative Grammar*. Blackwell.

[Huddleston and Pullum, 2002] Huddleston, R. D. and Pullum, G. K. (2002). *The Cambridge Grammar of the English Language*. Cambridge University Press.

[Hunt and Thomas, 1999] Hunt, A. and Thomas, D. (1999). *The Pragmatic Programmer: From Journeyman to Master*. Addison Wesley.

[Jackendoff, 1977] Jackendoff, R. (1977). *X-Syntax: a Study of Phrase Strucure*. Number 2 in Linguistic Inquiry Monograph. The MIT Press, Cambridge, MA.

[Johnson, 1988] Johnson, M. (1988). *Attribute Value Logic and Theory of Grammar*. CSLI Lecture Notes Series. Chicago University Press.

[Jurafsky and Martin, 2000] Jurafsky, D. and Martin, J. H. (2000). *Speech and Langugage Processing*. Prentice Hall, New Jersey.

[Kaplan, 1989] Kaplan, R. (1989). The formal architecture of Lexical-Functional Grammar. In Huang, C.-R. and Chen, K.-J., editors, *Proceedings of ROCLING II*, pages 1–18. Reprinted in Dalrymple, Kaplan, Maxwell, and Zaenen (eds), Formal Issues in Lexical-Functional Grammar, 7-27. Stanford: Center for the Study of Language and Information. 1995.

[Kaplan and Bresnan, 1982] Kaplan, R. and Bresnan, J. (1982). Lexical-functional grammar: A formal system for grammatical representation. In Bresnan, J., editor, *The Mental Representation of Grammatical Relations*, pages 173–281. The MIT Press, Cambridge, Mass.

[Kasper and Rounds, 1986] Kasper, R. T. and Rounds, W. C. (1986). A logical semantics for feature structures. In *Proceedings of the 24th annual meeting on Association for Computational Linguistics*, pages 257–266, Morristown, NJ, USA. Association for Computational Linguistics.

[Kay, 1985] Kay, M. (1985). Unification in grammar. In Dahl, V. and Saint-Dizier, P., editors, *Natural Language Understanding and Logic Programming*, pages 233–240. North-Holland. Proceedings of the First International Workshop on Natural Language Understanding and Logic Programming.

[Klein and Manning, 2003] Klein, D. and Manning, C. D. (2003). A* parsing: Fast exact viterbi parse selection. In *Proceedings of HLT-NAACL 03*.

[Levitin, 2004] Levitin, A. (2004). *The Design and Analysis of Algorithms*. Addison Wesley.

[Manning and Schutze, 1999] Manning, C. and Schutze, H. (1999). *Foundations of Statistical Natural Language Processing*. The MIT Press, Cambridge, MA.

[Miller and Charles, 1998] Miller, G. and Charles, W. (1998). Contextual correlates of semantic similarity. *Language and Cognitive Processes*, 6:1–28.

[Mitkov, 2002] Mitkov, R. (2002). *Oxford Handbook of Computational Linguistics*. Oxford University Press.

[Müller, 1999] Müller, S. (1999). *Deutsche Syntax deklarativ: Head-Driven Phrase Structure Grammar für das Deutsche*. Number 394 in Linguistische Arbeiten. Max Niemeyer Verlag, Tübingen.

[Nerbonne et al., 1994] Nerbonne, J., Netter, K., and Pollard, C. (1994). *German in Head-Driven Phrase Structure Grammar*. CSLI, Stanford, CA.

[Nivre et al., 2006] Nivre, J., Hall, J., and Nilsson, J. (2006). Maltparser: A data-driven parser-generator for dependency parsing. In *Proceedings of LREC*, pages 2216–2219.

[Partee, 1995] Partee, B. (1995). 'lexical semantics and compositionality. In Gleitman, L. R. and Liberman, M., editors, *An Invitation to Cognitive Science: Language*, volume Volume 1, pages 311–360. The MIT Press.

[Pullum, 2005] Pullum, G. K. (2005). Fossilized prejudices about "however".

[Radford, 1988] Radford, A. (1988). *Transformational Grammar: An Introduction*. Cambridge University Press, Cambridge.

[Ramshaw and Marcus, 1995] Ramshaw, L. A. and Marcus, M. P. (1995). Text chunking using transformation-based learning. In *Proceedings of the Third ACL Workshop on Very Large Corpora*, pages 82–94.

[Sag and Wasow, 1999] Sag, I. A. and Wasow, T. (1999). *Syntactic Theory: A Formal Introduction*. CSLI Publications.

[Shieber et al., 1983] Shieber, S., Uszkoreit, H., Pereira, F., Robinson, J., and Tyson, M. (1983). The formalism and implementation of PATR-II. In Grosz, B. J. and Stickel, M., editors, *Research on Interactive Acquisition and Use of Knowledge*, techreport 4, pages 39–79. SRI International, Menlo Park, CA. Final report for SRI Project 1894.

[Shieber, 1986] Shieber, S. M. (1986). *An Introduction to Unification-Based Approaches to Grammar*, volume 4 of *CSLI Lecture Notes Series*. Center for the Study of Language and Information, Stanford, CA.

[Strunk and White, 1999] Strunk, W. and White, E. B. (1999). *The elements of style*. Boston: Allyn and Bacon.

[Warren and Pereira, 1982] Warren, D. H. D. and Pereira, F. C. N. (1982). An efficient easily adaptable system for interpreting natural language queries. *AJCL*, 8(3-4):110–122.

[Zhao and Zobel, 2007] Zhao, Y. and Zobel, J. (2007). Search with style: authorship attribution in classic literature. In *Proceedings of the Thirtieth Australasian Computer Science Conference*. Association for Computing Machinery.