Chapter 6

Partial Parsing and Interpretation

6.1 Introduction

In processing natural language, we are looking for structure and meaning. Two of the most common methods 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 (e.g., part-of-speech tag) to the segment in question.

We will see that many tasks can be construed as a combination of segmentation and labeling. However, this involves generalizing our notion of segmentation to encompass *sequences* of tokens. Suppose that we are trying to recognize the names of people, locations and organizations in a piece of text (a task that is usually called **Named Entity Recognition**). Many of these names will involve more than one token: *Cecil H. Green*, *Escondido Village*, *Stanford University*; indeed, some names may have sub-parts that are also names: *Cecil H. Green Library*, *Escondido Village Conference Service Center*. In Named Entity Recognition, therefore, we need to be able to identify the beginning and end of multi-token sequences.

Identifying the boundaries of specific types of word sequences is also required when we want to recognize pieces of syntactic structure. Suppose for example that as a preliminary to Named Entity Recognition, we have decided that it would be useful to just pick out noun phrases from a piece of text. To carry this out in a complete way, we would probably want to use a proper syntactic parser. But parsing can be quite challenging and computationally expensive — is there an easier alternative? The answer is Yes: we can look for sequences of part-of-speech tags in a tagged text, using one or more patterns that capture the typical ingredients of a noun phrase.

For example, here is some Wall Street Journal text with noun phrases marked using brackets:

(1) [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 ./.

From the point of view of theoretical linguistics, we seem to have been rather unorthodox in our use of the term "noun phrase"; although all the bracketed strings are noun phrases, not every noun phrase has been captured. We will discuss this issue in more detail shortly. For the moment, let's say that we are identifying noun "chunks" rather than full noun phrases.

In chunking, we carry out segmentation and labeling of multi-token sequences, as illustrated in Figure 6.1. The smaller boxes show word-level segmentation and labeling, while the large boxes show higher-level segmentation and labeling. It is these larger pieces that we will call **chunks**, and the process of identifying them is called **chunking**.

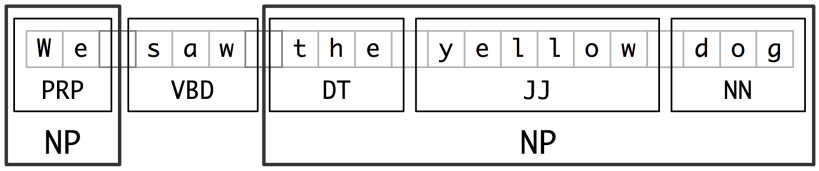


Figure 6.1: Segmentation and Labeling at both the Token and Chunk Levels

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.

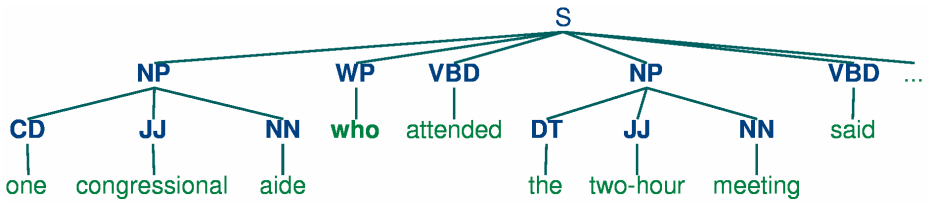
In this chapter, we will explore chunking in some depth, beginning with the definition and representation of chunks. We will see regular expression and n-gram approaches to chunking, and will develop and evaluate chunkers using the CoNLL-2000 chunking corpus. Towards the end of the chapter, we will look more briefly at Named Entity Recognition and related tasks.

6.2 Defining and Representing Chunks

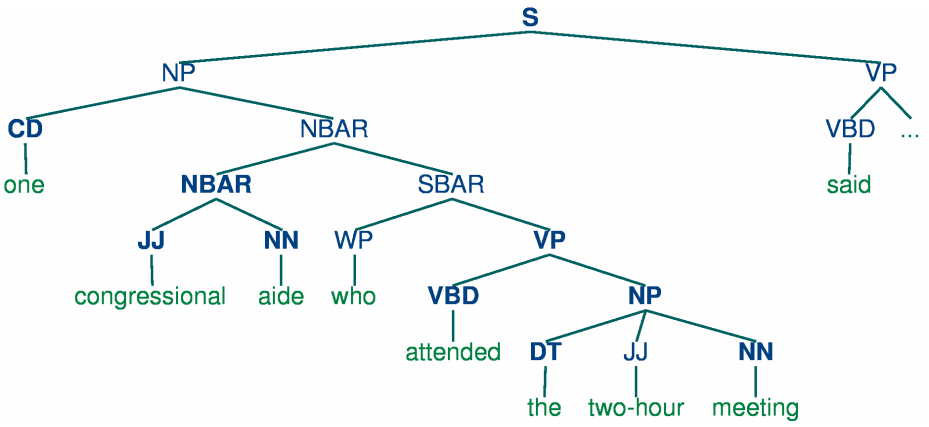
6.2.1 Chunking vs Parsing

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 ignores some items in the surface string. In fact, chunking is sometimes called **partial parsing**. Second, where parsing constructs nested structures that are arbitrarily deep, 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. This is illustrated in (2b) below, which shows an NP chunk structure and a completely parsed counterpart:

(2) a.



b.



A significant motivation for chunking is its robustness and efficiency relative to parsing. As we will see in Chapter 7, parsing has problems with robustness, given the difficulty in gaining broad coverage while minimizing 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.

6.2.2 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 6.2.

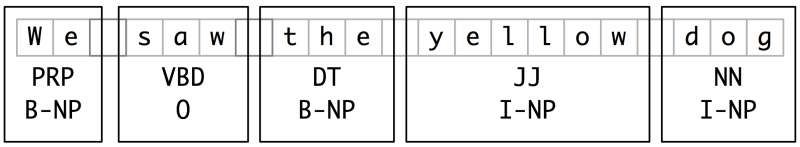


Figure 6.2: Tag Representation of Chunk Structures

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 6.2:

```
We PRP B-NP
saw VBD O
the DT B-NP
little JJ I-NP
yellow 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.

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 6.3:

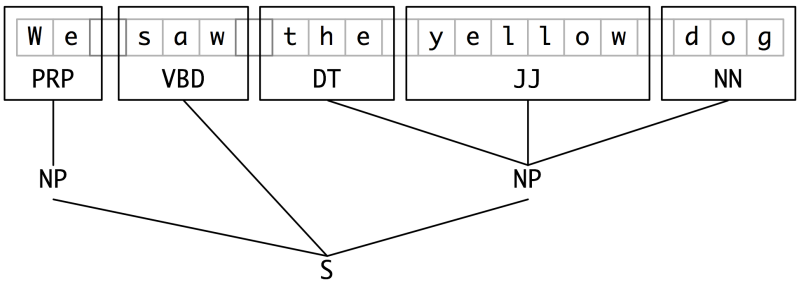


Figure 6.3: Tree Representation of Chunk Structures

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.

6.3 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.

Chunking in NLTK begins with tagged tokens.

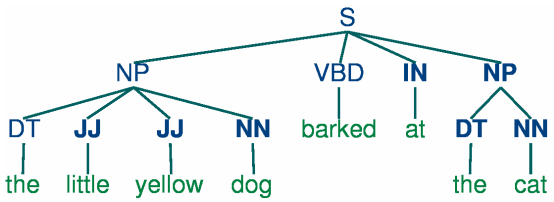
```
>>> tagged_tokens = [("the", "DT"), ("little", "JJ"), ("yellow", "JJ"),
... ("dog", "NN"), ("barked", "VBD"), ("at", "IN"),  ("the", "DT"), ("cat", "NN")]
```

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.

```
>>> cp = nltk.RegexpParser("NP: {<DT>?<JJ>*<NN>}")
```

We create a chunker `cp` that can then be used repeatedly to parse tagged input. The result of chunking is a tree.

```
>>> cp.parse(tagged_tokens).draw()
```



Note

Remember that our program samples assume you begin your interactive session or your program with: `import nltk, re, pprint`

6.3.1 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 that 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. A good way to learn about tag patterns is via a graphical interface `nltk.draw.rechunkparser.demo()`.

6.3.2 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 6.1 shows a simple chunk grammar consisting of two patterns. The first pattern matches an optional determiner or possessive pronoun (recall that `|` indicates disjunction), zero or more adjectives, then a noun. The second rule matches one or more proper nouns. We also define some tagged tokens to be chunked, and run the chunker on this input.

> **Note**
>
> The `$` symbol is a special character in regular expressions, and therefore needs to be escaped with the backslash `\` in order to match the tag `PP$`.

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 = nltk.RegexpParser(grammar)
>>> print cp.parse(nouns)
(S (NP money/NN market/NN) fund/NN)
```

Listing 6.1 Simple Noun Phrase Chunker

```
grammar = r"""
  NP: {<DT|PP\$>?<JJ>*<NN>}   # chunk determiner/possessive, adjectives and nouns
      {<NNP>+}                # chunk sequences of proper nouns
"""
cp = nltk.RegexpParser(grammar)
tagged_tokens = [("Rapunzel", "NNP"), ("let", "VBD"), ("down", "RP"), ("her", "PP$"
              ("golden", "JJ"), ("hair", "NN")]

>>> print cp.parse(tagged_tokens)
(S
  (NP Rapunzel/NNP)
  let/VBD
  down/RP
  (NP her/PP$ long/JJ golden/JJ hair/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>+}`.

6.3.3 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 6.2, 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. As you can see, you need to be careful to put chunk rules in the right order.

You may have noted that we have added explanatory comments, preceded by #, to each of our tag rules. Although it is not strictly necessary to do this, it's a helpful reminder of what a rule is meant to do, and it is used as a header line for the output of a rule application when tracing is on.

You might want to test out some of your rules on a corpus. One option is to use the Brown corpus. However, you need to remember that the Brown tagset is different from the Penn Treebank tagset that we have been using for our examples so far in this chapter; see Table 3.6 in Chapter 3 for a refresher. Because the Brown tagset uses `NP` for proper nouns, in this example we have followed Abney in labeling noun chunks as `NX`.

```
>>> grammar = (r"""
...     NX: {<AT|AP|PP\$>?<JJ.*>?<NN.*>}  # Chunk article/numeral/possessive+adj+nou
...         {<NP>+}                       # Chunk one or more proper nouns
... """)
>>> cp = nltk.RegexpParser(grammar)
```

Listing 6.2 Two Noun Phrase Chunkers Having Identical Rules in Different Orders

```
tagged_tokens = [("The", "DT"), ("enchantress", "NN"),
            ("clutched", "VBD"), ("the", "DT"), ("beautiful", "JJ"), ("hair", "NN")]
cp1 = nltk.RegexpParser(r"""
  NP: {<DT><JJ><NN>}      # Chunk det+adj+noun
      {<DT|NN>+}          # Chunk sequences of NN and DT
  """)
cp2 = nltk.RegexpParser(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>  <NN>  <VBD>  <DT>  <JJ>  <NN>
# Chunk det+adj+noun:
 <DT>  <NN>  <VBD> {<DT>  <JJ>  <NN>}
# Chunk sequences of NN and DT:
{<DT>  <NN>} <VBD> {<DT>  <JJ>  <NN>}
(S
  (NP The/DT enchantress/NN)
  clutched/VBD
  (NP the/DT beautiful/JJ hair/NN))
>>> print cp2.parse(tagged_tokens, trace=1)
# Input:
 <DT>  <NN>  <VBD>  <DT>  <JJ>  <NN>
# Chunk sequences of NN and DT:
{<DT>  <NN>} <VBD> {<DT>} <JJ> {<NN>}
# Chunk det+adj+noun:
{<DT>  <NN>} <VBD> {<DT>} <JJ> {<NN>}
(S
  (NP The/DT enchantress/NN)
  clutched/VBD
  (NP the/DT)
  beautiful/JJ
  (NP hair/NN))
```

```
>>> sent = nltk.corpus.brown.tagged_sents(categories='a')[112]
>>> print cp.parse(sent)
(S
  (NX His/PP$ contention/NN)
  was/BEDZ
  denied/VBN
  by/IN
  (NX several/AP bankers/NNS)
  ,/,
  including/IN
  (NX Scott/NP Hudson/NP)
  of/IN
  (NX Sherman/NP)
  ,/,
  (NX Gaynor/NP B./NP Jones/NP)
  of/IN
  (NX Houston/NP)
  ,/,
  (NX J./NP B./NP Brady/NP)
  of/IN
  (NX Harlingen/NP)
  and/CC
  (NX Howard/NP Cox/NP)
  of/IN
  (NX Austin/NP)
  ./.)
```

6.3.4 Exercises

1. ☼ **Chunk Grammar Development:** Try developing a series of chunking rules using the graphical interface accessible via `nltk.draw.rechunkparser.demo()`

2. ☼ **Chunking Demonstration:** Run the chunking demonstration: `nltk.chunk.demo()`

3. ☼ **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?

4. ☼ 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.

5. ◑ Write a 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.

6. ◑ 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".

6.4 Scaling Up

Now you have a taste of what chunking can do, but we have not explained how to carry out a quantitative evaluation of chunkers. For this, we need to get access to a corpus that has been annotated not only with parts-of-speech, but also with chunk information. 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 a chunked 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. Our focus throughout will be on scaling up the coverage of a chunker.

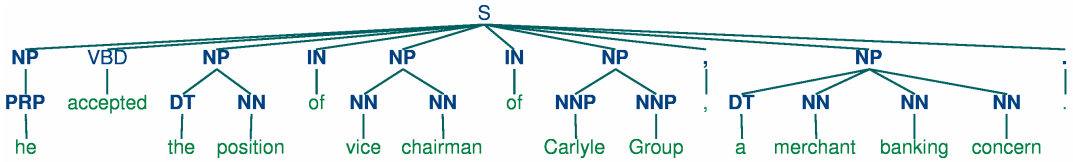
6.4.1 Reading IOB Format and the CoNLL 2000 Corpus

Using the `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
... '''
>>> nltk.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, divided into "train" and "test" portions, annotated with part-of-speech tags and chunk tags in the IOB format. We can access the data using an NLTK corpus reader called `conll2000`. Here is an example that reads the 100th sentence of the "train" portion of the corpus:

```
>>> print nltk.corpus.conll2000.chunked_sents('train.txt')[99]
(S
  (PP Over/IN)
  (NP a/DT cup/NN)
  (PP of/IN)
  (NP coffee/NN)
  ,/,
  (NP Mr./NNP Stone/NNP)
  (VP told/VBD)
  (NP his/PRP$ story/NN)
  ./.)
```

This showed three chunk types, for NP, VP and PP. We can also select which chunk types to read:

```
>>> print nltk.corpus.conll2000.chunked_sents('train.txt', chunk_types=('NP',))[99]
(S
  Over/IN
  (NP a/DT cup/NN)
  of/IN
  (NP coffee/NN)
  ,/,
  (NP Mr./NNP Stone/NNP)
  told/VBD
  (NP his/PRP$ story/NN)
  ./.)
```

6.4.2 Simple Evaluation and Baselines

Armed with a corpus, it is now possible to carry out some simple evaluation. We start off by establishing a baseline for the trivial chunk parser `cp` that creates no chunks:

```
>>> cp = nltk.RegexpParser("")
>>> print nltk.chunk.accuracy(cp, nltk.corpus.conll2000.chunked_sents('train.txt',
0.440845995079
```

This indicates that more than a third of the words are tagged with `O` (i.e., not in an NP chunk). Now let's try a naive regular expression chunker that looks for tags (e.g., `CD`, `DT`, `JJ`, etc.) beginning with letters that are typical of noun phrase tags:

```
>>> grammar = r"NP: {<[CDJNP].*>+}"
>>> cp = nltk.RegexpParser(grammar)
>>> print nltk.chunk.accuracy(cp, nltk.corpus.conll2000.chunked_sents('train.txt',
0.874479872666
```

As you can see, this approach achieves pretty good results. In order to develop a more data-driven approach, let's define 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, where `chtag` is `B-NP` or `I-NP`), or outside a chunk (the `False` case, where `chtag` is `O`). It returns a list of those tags that occur inside chunks more often than outside chunks.

Listing 6.3 Capturing the conditional frequency of NP Chunk Tags

```
def chunked_tags(train):
    """Generate a list of tags that tend to appear inside chunks"""
    cfdist = nltk.ConditionalFreqDist()
    for t in train:
        for word, tag, chtag in nltk.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]

>>> train_sents = nltk.corpus.conll2000.chunked_sents('train.txt', chunk_types=('NP',))
>>> print chunked_tags(train_sents)
['PRP$', 'WDT', 'JJ', 'WP', 'DT', '#', '$', 'NN', 'FW', 'POS',
'PRP', 'NNS', 'NNP', 'PDT', 'RBS', 'EX', 'WP$', 'CD', 'NNPS', 'JJS', 'JJR']
```

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:

The final step is to train this chunker and test its accuracy (this time on the "test" portion of the corpus, i.e., data not seen during training):

```
>>> train_sents = nltk.corpus.conll2000.chunked_sents('train.txt', chunk_types=('NP',))
>>> test_sents  = nltk.corpus.conll2000.chunked_sents('test.txt', chunk_types=('NP',))
>>> cp = baseline_chunker(train_sents)
>>> print nltk.chunk.accuracy(cp, test_sents)
0.914262194736
```

6.4.3 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>"`.]

6.4.4 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**.

Following [Church, Young, & Bloothooft, 1996], we define a **chink** as a sequence of tokens that is not included in a chunk. In the following example, `barked/VBD at/IN` is a chink:

```
[ the/DT little/JJ yellow/JJ dog/NN ] barked/VBD at/IN [ the/DT cat/NN ]
```

Listing 6.4 Deriving a Regexp Chunker from Training Data

```
def baseline_chunker(train):
    chunk_tags = [re.sub(r'(\W)', r'\\\1', tag)
                  for tag in chunked_tags(train)]
    grammar = 'NP: {<%s>+}' % '|'.join(chunk_tags)
    return nltk.RegexpParser(grammar)
```

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 6.1.

	Entire chunk	Middle of a chunk	End of a chunk
Input	[a/DT little/JJ dog/NN]	[a/DT little/JJ dog/NN]	[a/DT little/JJ dog/NN]
Operation	Chink "DT JJ NN"	Chink "JJ"	Chink "NN"
Pattern	"}DT JJ NN{"	"}JJ{"	"}NN{"
Output	a/DT little/JJ dog/NN	[a/DT] little/JJ [dog/NN]	[a/DT little/JJ] dog/NN

Table 6.1: 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:

Listing 6.5 Simple Chinker

```
grammar = r"""
  NP:
    {<.*>+}          # Chunk everything
    }<VBD|IN>+{      # Chink sequences of VBD and IN
  """
tagged_tokens = [("the", "DT"), ("little", "JJ"), ("yellow", "JJ"),
       ("dog", "NN"), ("barked", "VBD"), ("at", "IN"),  ("the", "DT"), ("cat", "NN"
cp = nltk.RegexpParser(grammar)

>>> print cp.parse(tagged_tokens)
(S
  (NP the/DT little/JJ yellow/JJ dog/NN)
  barked/VBD
  at/IN
  (NP the/DT cat/NN))
>>> test_sents = nltk.corpus.conll2000.chunked_sents('test.txt', chunk_types=('NP',
>>> print nltk.chunk.accuracy(cp, test_sents)
0.581041433607
```

A chunk grammar can use any number of chunking and chinking patterns in any order.

6.4.5 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 = nltk.corpus.conll2000.chunked_sents('train.txt')[99]
>>> print example
(S
  (PP Over/IN)
```

```
  (NP a/DT cup/NN)
  (PP of/IN)
  (NP coffee/NN)
  ,/,
  (NP Mr./NNP Stone/NNP)
  (VP told/VBD)
  (NP his/PRP$ story/NN)
  ./.)
>>> print example.flatten()
(S
  Over/IN
  a/DT
  cup/NN
  of/IN
  coffee/NN
  ,/,
  Mr./NNP
  Stone/NNP
  told/VBD
  his/PRP$
  story/NN
  ./.)
```

Now we can set up a multi-stage chunk grammar, as shown in Listing 6.6. It has a stage for each of the chunk types.

6.4.6 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.

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 (cf precision and recall, from Chapter 4).

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:

```
>>> correct = nltk.chunk.tagstr2tree(
...     "[ the/DT little/JJ cat/NN ] sat/VBD on/IN [ the/DT mat/NN ]")
>>> print correct.flatten()
(S the/DT little/JJ cat/NN sat/VBD on/IN the/DT mat/NN)
```

Listing 6.6 A Multistage Chunker

```
cp = nltk.RegexpParser(r"""
  NP: {<DT>?<JJ>*<NN.*>+}     # noun phrase chunks
  VP: {<TO>?<VB.*>}           # verb phrase chunks
  PP: {<IN>}                  # prepositional phrase chunks
  """)

>>> example = nltk.corpus.conll2000.chunked_sents('train.txt')[99]
>>> print cp.parse(example.flatten(), trace=1)
# Input:
 <IN>  <DT>  <NN>  <IN>  <NN>  <,>  <NNP>  <NNP>  <VBD>  <PRP$>  <NN>  <.>
# noun phrase chunks:
 <IN> {<DT>  <NN>} <IN> {<NN>} <,> {<NNP>  <NNP>} <VBD>  <PRP$> {<NN>} <.>
# Input:
 <IN>  <NP>  <IN>  <NP>  <,>  <NP>  <VBD>  <PRP$>  <NP>  <.>
# verb phrase chunks:
 <IN>  <NP>  <IN>  <NP>  <,>  <NP> {<VBD>} <PRP$>  <NP>  <.>
# Input:
 <IN>  <NP>  <IN>  <NP>  <,>  <NP>  <VP>  <PRP$>  <NP>  <.>
# prepositional phrase chunks:
{<IN>} <NP> {<IN>} <NP>  <,>  <NP>  <VP>  <PRP$>  <NP>  <.>
(S
  (PP Over/IN)
  (NP a/DT cup/NN)
  (PP of/IN)
  (NP coffee/NN)
  ,/,
  (NP Mr./NNP Stone/NNP)
  (VP told/VBD)
  his/PRP$
  (NP story/NN)
  ./.)
```

We run a chunker over this flattened data, and compare the resulting chunked sentences with the originals, as follows:

```
>>> grammar = r"NP: {<PRP|DT|POS|JJ|CD|N.*>+}"
>>> cp = nltk.RegexpParser(grammar)
>>> tagged_tokens = [("the", "DT"), ("little", "JJ"), ("cat", "NN"),
... ("sat", "VBD"), ("on", "IN"), ("the", "DT"), ("mat", "NN")]
>>> chunkscore = nltk.chunk.ChunkScore()
>>> guess = cp.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.

```
>>> grammar = r"NP: {<DT|JJ|NN>+}"
>>> cp = nltk.RegexpParser(grammar)
>>> chunkscore = nltk.chunk.ChunkScore()
>>> for file in nltk.corpus.treebank_chunk.files()[:5]:
...     for chunk_struct in nltk.corpus.treebank_chunk.chunked_sents(file):
...         test_sent = cp.parse(chunk_struct.flatten())
...         chunkscore.score(chunk_struct, test_sent)
>>> print chunkscore
ChunkParse score:
    Precision:  42.3%
    Recall:     29.9%
    F-Measure:  35.0%
```

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'))]
```

6.4.7 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.

6.4.8 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 `nltk.RegexpParser`. 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 that 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. ◑ 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.

5. ★ 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.

6. ★ 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.

7. ★ 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.

8. ★ Develop an NP chunker that 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')`...

9. ★ The Penn Treebank contains a section of tagged Wall Street Journal text that has been chunked into noun phrases. The format uses square brackets, and we have encountered it several times during this chapter. The Treebank corpus can be accessed using: `for sent in nltk.corpus.treebank_chunk.chunked_sents(file)`. These are flat trees, just as we got using `nltk.corpus.conll2000.chunked_sents()`.

   a) The functions `nltk.tree.pprint()` and `nltk.chunk.tree2conllstr()` can be used to create Treebank and IOB strings from a tree. Write functions `chunk2brackets()` and `chunk2iob()` that 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.)

6.5 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 as to whether we could use the same *n*-gram tagging methods we saw in Chapter 3, applied to a different vocabulary. In this case, rather than trying to determine the correct part-of-speech tag, given a word, we are trying to determine the correct chunk tag, given a part-of-speech tag.

The first step is to get the `word,tag,chunk` triples from the CoNLL 2000 corpus and map these to `tag,chunk` pairs:

```
>>> chunk_data = [[(t,c) for w,t,c in nltk.chunk.tree2conlltags(chtree)]
...               for chtree in nltk.corpus.conll2000.chunked_sents('train.txt')]
```

We will now train two *n*-gram taggers over this data.

6.5.1 A Unigram Chunker

To start off, we train and score a **unigram chunker** on the above data, just as if it was a tagger:

```
>>> unigram_chunker = nltk.UnigramTagger(chunk_data)
>>> print nltk.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 CONLL 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 out the unigram chunker 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/NN` are correctly tagged as `I-NP`; they are not in the initial position that should be tagged `B-NP`. The chunker incorrectly tags `widely/RB` as `O`, and it incorrectly tags the infinitival `to/TO` as `B-PP`, as if it was a preposition starting a prepositional phrase.

6.5.2 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 = nltk.BigramTagger(chunk_data, backoff=unigram_chunker)
>>> print nltk.tag.accuracy(bigram_chunker, chunk_data)
0.893220987404
```

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.

6.5.3 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.

6.6 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. 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 6.7 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.

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)
```

Listing 6.7 A Chunker that Handles NP, PP, VP and S

```
grammar = 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
  """
cp = nltk.RegexpParser(grammar)
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)))))
```

```
(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 = nltk.RegexpParser(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).

6.7 Shallow Interpretation

The main form of shallow semantic interpretation that we will consider is **Information Extraction**. This refers to 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 (3), 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*'.

(3) ... 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 (NEs) 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 (3):

(4) ... said <ne type='PERSON'>William Gale</ne>, an economist at the <ne type='ORGANIZATION'>Brookings 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 may 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 organizations 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 term 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 that we saw being deployed in the CoNLL chunk data (Chapter 6). 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 NEs 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 that allows us to disregard strings such as *success in supervising the transition of*, where *in* is followed by a gerundive verb.

```
>>> IN = re.compile(r'.*\bin\b(?!\b.+ing\b)')
>>> for doc in nltk.corpus.ieer.parsed_docs('NYT_19980315'):
...     for rel in nltk.sem.extract_rels('ORG', 'LOC', doc, pattern = IN):
...         print nltk.sem.show_raw_rtuple(rel)
[ORG: 'WHYY'] 'in' [LOC: 'Philadelphia']
[ORG: 'McGlashan &AMP; Sarrail'] 'firm in' [LOC: 'San Mateo']
[ORG: 'Freedom Forum'] 'in' [LOC: 'Arlington']
[ORG: 'Brookings Institution'] ', the research group in' [LOC: 'Washington']
[ORG: 'Idealab'] ', a self-described business incubator based in' [LOC: 'Los Angele
[ORG: 'Open Text'] ', based in' [LOC: 'Waterloo']
[ORG: 'WGBH'] 'in' [LOC: 'Boston']
[ORG: 'Bastille Opera'] 'in' [LOC: 'Paris']
[ORG: 'Omnicom'] 'in' [LOC: 'New York']
[ORG: 'DDB Needham'] 'in' [LOC: 'New York']
[ORG: 'Kaplan Thaler Group'] 'in' [LOC: 'New York']
[ORG: 'BBDO South'] 'in' [LOC: 'Atlanta']
[ORG: 'Georgia-Pacific'] 'in' [LOC: 'Atlanta']
```

Searching for the keyword works *in* reasonably well, though it will also retrieve false positives such as `[ORG: House Transportation Committee] , secured the most money in the [LOC: New York]`; there is unlikely to be simple string-based method of excluding filler strings such as this.

```
>>> vnv = """
... (
... is/V|
... was/V|
... werd/V|
... wordt/V
... )
... .*
... van/Prep
... """
```

```
>>> VAN = re.compile(vnv, re.VERBOSE)
>>> for r in nltk.sem.extract_rels('PER', 'ORG', corpus='conll2002-ned', pattern=VAN):
...     print show_tuple(r)
```

6.8 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 [Church, Young, & Bloothooft, 1996], 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.

6.9 Further Reading

For more examples of chunking with NLTK, please see the guide at `http://nltk.org/doc/guides/chunk.html`.

The popularity of chunking is due in great part to pioneering work by Abney e.g., [Church, Young, & Bloothooft, 1996]. Abney's Cass chunker is available at http://www.vinartus.net/spa/97a.pdf

The word **chink** initially meant a sequence of stopwords, according to a 1975 paper by Ross and Tukey [Church, Young, & Bloothooft, 1996].

The IOB format (or sometimes **BIO Format**) was developed for NP chunking by [Ramshaw & Marcus, 1995], and was used for the shared NP bracketing task run by the *Conference on Natural Language Learning* (CoNLL) in 1999. 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.

Section 13.5 of [Jurafsky & Martin, 2008] contains a discussion of chunking.