NLTK Tutorial: Chunking

Edward Loper

Steven Bird

Table of Contents

1. Introduction

For many practical purposes it is not necessary to construct a complete parse tree for a sentence. *Chunk parsing*, also known as *partial parsing*, *light parsing*, or just *chunking*, is an approach in which the parser assigns incomplete syntactic structure to the phrase. The most common use of chunking is in *information extraction* and *message understanding*, where the content of a text is mined for information that will be used to fill out a template. Chunk parsing is actually more like tagging than conventional parsing, in the sense that it typically uses finite-state methods and employs rules that work directly off the (tagged) surface string. This contrasts with conventional parsing, which normally uses context-free rules expressed over abstract categories (e.g. phrasal categories).

In terms of the other NLP tasks, chunking usually takes place after tokenization and tagging.

Typically, chunk parsers are based on finite-state methods. The constraints about well-formed chunks are expressed using regular expressions over the sequence of word tags. This tutorial describes the NLTK regular-expression chunk parser.

2. The Chunk Parser Interface

Unlike conventional parsers, chunk parsers do not need to construct complete parse trees. Instead, they return a simple structure known as a chunk structure. A *chunk structure* is a list, containing a mixture of tokens and sublists of tokens. Here is an example of a chunk structure:

```
[
  [
    'the'/'DT'@[1],
    'cat'/'NN'@[2]
  ],
  'sat'/'VBD'@[3],
  'on'/'IN'@[4],
  [
    'the'/'DT'@[5],
    'mat'/'NN'@[6]
  ]
]
```

We can define chunk structure as follows:

Term		**Definition**
tagged token	:=	a `Token` whose type is a `TaggedType`

Term		Definition
chunk	:=	list of *tagged tokens*
chunk structure	:=	list of (*tagged token* or *chunk*)

Note: Although the terms *chunk* and *tagged text* have the same definition (namely, a list of tagged tokens), they are used in different contexts.

Chunk parsers are defined by the `ChunkParserI` interface. Every chunk parser must define a `parse` method. This method identifies the chunks in a tagged text, by returning a chunk structure that corresponds to the text.

All chunk parsers, including the `REChunkParser` class discussed below, must inherit from `ChunkParserI` and reimplement its `parse` method.

Note: In general, chunk parsers can be defined to use any kind of token (not just tagged token). In the general case, a *chunk* would be defined as *a list of `Tokens`*, and a *chunk structure* as *a list of (`Token` or chunk)*. However, the only chunk parser currently implemented by NLTK operates on tagged tokens; and in general, most chunk parsers that have been created operate on tagged tokens. So for the purposes of this tutorial, we will assume that chunk parsers are defined to use tagged tokens.

Note: The current definition of chunk structure is too weak to express a more general kind of chunking which associates a type with each chunk. For example, there are chunkers which can identify both NP and VP chunks. For these it is necessary to define a more expressive representation. For this and other reasons, we may eventually change the `ChunkParserI` to use the `TreeToken` class instead of chunk structures.

3. The Chunking Tokenizer

A common file representation for chunked tagged text is as follows:

```
[ the/DT little/JJ cat/NN ] sat/VBD on/IN [ the/DT mat/NN ]
```

It is often useful to be able to tokenize text input in this format as part of the evaluation process. For this purpose, the `nltk.chunkparser` module contains a special kind of tagged tokenizer class, the `ChunkTaggedTokenizer`.

```
>>> from nltk.chunkparser import *
>>> correct_sent = """
[ the/DT little/JJ cat/NN ]
sat/VBD on/IN
[ the/DT mat/NN ]
"""
>>> tokenizer = ChunkedTaggedTokenizer()
>>> correct_chunks = tokenizer.tokenize(correct_sent)
[['the'/'DT'@[0w], 'little'/'JJ'@[1w], 'cat'/'NN'@[2w]],
 'sat'/'VBD'@[3w], 'on'/'IN'@[4w],
 ['the'/'DT'@[5w], 'mat'/'NN'@[6w]]]
```

3.1. Un-Chunking

Another utility defined in the `nltk.chunkparser` module, `unchunk`, will convert a chunk structure to a tagged text by removing all chunking:

```
>>> unchunked_sent = unchunk(correct_chunks)
['the'/'DT'@[0w], 'little'/'JJ'@[1w], 'cat'/'NN'@[2w], 'sat'/'VBD'@[3w],
 'on'/'IN'@[4w], 'the'/'DT'@[5w], 'mat'/'NN'@[6w]]
```

3.2. Tokenizing WSJ Tagged Data

The resources CD contains about 2,500 sentences of tagged, chunked data, split between 99 files. Within each file, sentences are split by blank lines and by "divider" lines containing 38 equal signs. We can use a regular expression tokenizer to divide these files into sentences. The following example reads the chunked, tagged data in these 99 files, and prints out each chunked sentence on a separate line.

```
# Read the WSJ corpus as a list of sentences.  Sentences are
# separated by blank lines and by sequences of equal signs.
import os
sent_tokenizer = RETokenizer('(=======+)?\n\n+', 0)
sentences = []
files = os.listdir(path)
for file in files:
    print 'Loading %s...' % file
    text = open(path+file).read()
    sentences += sent_tokenizer.tokenize(text, source=file, unit='s')

# Tokenize each sentence.
ctt = ChunkedTaggedTokenizer()
for sentence in sentences:
    chunked_sent = ctt.tokenize(sentence.type(), source=sentence.loc())
    print chunked_sent
```

Note: In this example, `path` is the directory containing the WSJ tagged data. Depending on what operating system you are using, and where you installed NLTK, path can probably be defined with one of the following statements, or a similar statement:

```
# Unagi:
>>> path = "/spd25/cis530/cdrom/resources/data/wsj/tagged/"

# Linux, with the CD mounted at /mnt/cdrom:
>>> path = "/mnt/cdrom/resources/data/wsj/tagged/"

# Windows, with the CD in drive E:.
>>> path = "E:\\data\\wsj\\tagged/"
```

4. REChunkParser

For the remainder of this tutorial, we will examine `REChunkParser`. This chunk parser uses a sequence of regular expression based *rules* to chunk a tagged text.

`REChunkParser` works by manipulating a *chunking hypothesis*, which records a particular chunking of the text:

1. The `REChunkParser` begins with a chunking hypothesis where no tokens are chunked.
2. Each rule is then applied, in turn, to the chunking hypothesis. Applying a rule to a chunking hypothesis updates the chunking encoded by the hypothesis.
3. After the all of the rules have been applied, the chunk structure corresponding to the final chunking hypothesis is returned.

New `REChunkParsers` are constructed from a list of rules, using the `REChunkParser` constructor:

```
>>> rules = [rule1, rule2, rule3]
>>> chunkparser = REChunkParser(rules)
```

Note: We will discuss how to construct rules in the following sections.

Tagged texts are chunked using the `parse` method:

```
>>> chunkparser.parse(unchunked_sent)
[['the'/'DT'@[0w], 'little'/'JJ'@[1w], 'cat'/'NN'@[2w]],
 'sat'/'VBD'@[3w], 'on'/'IN'@[4w],
    ['the'/'DT'@[5w], 'mat'/'NN'@[6w]]]
```

5. Tag Strings and Tag Patterns

Rules are defined in terms of regular expression patterns over "tag strings." A *tag string* is a string consisting of angle-bracket delimited tag names. An example of a *tag string* is:

```
<DT><JJ><NN><VBD><DT><NN>
```

Note: Tag strings do not contain any whitespace.

Most rules are defined using a special kind of regular expression pattern, called a *tag pattern*. Tag patterns are identical to `re` regular expression patterns in most respects; however, there are a few differences, which are intended to simplify their use with `REChunkParsers`:

- In tag patterns, "<" and ">" act like grouping parentheses; so the tag pattern "`<NN>+`" matches one or more repetitions of "`<NN>`"; and "`<NN|JJ>`" matches "`<NN>`" or "`<JJ>`."
- Whitespace in tag patterns is ignored; so "`<DT> | <NN>`" is equivalent to "`<DT>|<NN>`". This allows you to make your tag patterns easier to read, by inserting whitespace in appropriate places.
- In tag patterns "`.`" is equivalent to "`[^{}<>]`"; so "`<NN.*>`" matches any single tag starting with "`NN`." (The use of `{` and `}` will be explained later.)

6. REChunkParser Rules

`REChunkParser` rules are all defined in terms of regular expression patterns over tag strings. The `nltk.rechunkparser` module currently defines six different kinds of rule. `REChunkParserRule` is the most general kind of rule; all other rules are derived from it. We will cover `REChunkParserRule` itself later in this tutorial. First, we will cover the simpler rules that are derived from `REChunkParserRule`:

- `ChunkRule` chunks anything that matches a given tag pattern.
- `ChinkRule` chinks anything that matches a given tag pattern.
- `UnChunkRule` will un-chunk any chunk that matches a given tag pattern.

- `MergeRule` can be used to merge two contiguous chunks.
- `SplitRule` can be used to split a single chunk into two smaller chunks.

6.1. Rule Descriptions

Each rule must have a "description" associated with it, which provides a short explanation of the purpose or the effect of the rule. This description is accessed via the `descr` method:

```
>>> print rule1.descr()
'Chunk sequences of NN and DT'
```

7. ChunkRule

The simplest type of rule is `ChunkRule`. A `ChunkRule` chunks anything that matches a given tag pattern. `ChunkRules` are created with the `ChunkRule` constructor, which takes a tag pattern and a description string. For example, the following code creates and uses an `REChunkParser` that chunks any sequence of tokens whose tags are all "`NN`" or "`DT`":

```
>>> rule1 = ChunkRule('<NN|DT>+',
...                   'Chunk sequences of NN and DT')
>>> chunkparser = REChunkParser( [rule1] )
>>> chunkparser.parse(unchunked_sent)
[['the'/'DT'@[0w]],
 'little'/'JJ'@[1w],
 ['cat'/'NN'@[2w]],
 'sat'/'VBD'@[3w], 'on'/'IN'@[4w],
 ['the'/'DT'@[5w], 'mat'/'NN'@[6w]]]
```

We can also use more complex tag patterns. The following code chunks any sequence of tokens beginning with an optional determiner ("`DT`"), followed by zero or more adverbs of any type ("`JJ.*`"), followed by a single noun of any type ("`NN.*`").

```
>>> rule1 = ChunkRule('<DT>?<JJ.*>*<NN.*>',
...                   'Chunk optional det, zero or more adj, and a noun')
>>> chunkparser = REChunkParser([rule1])
>>> chunkparser.parse(unchunked_sent)
[['the'/'DT'@[0w], 'little'/'JJ'@[1w], 'cat'/'NN'@[2w]],
 'sat'/'VBD'@[3w], 'on'/'IN'@[4w],
 ['the'/'DT'@[5w], 'mat'/'NN'@[6w]]]
```

If a tag pattern matches at multiple 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 the first two nouns will be chunked:

```
>>> from nltk.tagger import *
>>> noun_str = "dog/NN cat/NN mouse/NN"
'dog/NN cat/NN mouse/NN'
>>> three_nouns = TaggedTokenizer().tokenize(noun_str)
['dog'/'NN'@[0w], 'cat'/'NN'@[1w], 'mouse'/'NN'@[2w]]
>>> rule1 = ChunkRule('<NN><NN>',
...                   'Chunk two consecutive nouns')
>>> chunkparser = REChunkParser([rule1])
>>> chunkparser.parse(three_nouns)
[['dog'/'NN'@[0w], 'cat'/'NN'@[1w]],
 'mouse'/'NN'@[2w]]
```

7.1. Tracing

Both the constructor and `parse` method of `REChunkParser` takes an optional "`trace`" argument, which specifies whether debugging output should be shown during parsing. This output shows the rules that are applied, and shows the chunking hypothesis at each stage of processing.

```
>>> chunkparser = REChunkParser([rule1], 1)
>>> chunkparser.parse(unchunked_sent, 1)
Input:
                <DT>  <JJ>  <NN>  <VBD>  <IN>  <DT>  <NN>
Chunk optional det, zero or more adj, and a noun:
               {<DT>  <JJ>  <NN>} <VBD>  <IN> {<DT>  <NN>}

[['the'/'DT'@[0w], 'little'/'JJ'@[1w], 'cat'/'NN'@[2w]],
 'sat'/'VBD'@[3w], 'on'/'IN'@[4w],
 ['the'/'DT'@[5w], 'mat'/'NN'@[6w]]]
```

Note: The `trace` value given to the `parse` method overrides the value given to the constructor.

In the tracing output, chunking is indicated by braces ("{" and "}"), instead of the more conventional square brackets ("[" and "]"). The reasons for this will be discussed when we cover general transformational rules.

7.2. Cascading Rules

`REChunkParsers` can be generated from multiple `ChunkRules`. In this case, each rule will be applied in turn. For example, the following code chunks the example sentence by first finding all sequences of three tokens whose tags are "`DT`", "`JJ`", and "`NN`"; and then looking for any sequence of tokens whose tags are either "`DT`" or "`NN`".

```
>>> rule1 = ChunkRule('<DT><JJ><NN>',
...                   'Chunk det+adj+noun')
>>> rule2 = ChunkRule('<DT|NN>+',
...                   'Chunk sequences of NN and DT')
>>> chunkparser = REChunkParser( [rule1, rule2] )

# Parse unchunked_sent with tracing turned on.
>>> chunkparser.parse(unchunked_sent, 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>}

[['the'/'DT'@[0w], 'little'/'JJ'@[1w], 'cat'/'NN'@[2w]],
 'sat'/'VBD'@[3w], 'on'/'IN'@[4w],
 ['the'/'DT'@[5w], 'mat'/'NN'@[6w]]]
```

When a `ChunkRule` is applied to a chunking hypothesis, it will only create chunks that do not partially overlap with chunks already in the hypothesis. Thus, if we apply these two rules in reverse order, we will get a different result:

```
>>> chunkparser = REChunkParser( [rule2, rule1] )

# Parse unchunked_sent with tracing turned on.
>>> chunkparser.parse(unchunked_sent, 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>}

[['the'/'DT'@[0w], 'little'/'JJ'@[1w], 'cat'/'NN'@[2w]],
 'sat'/'VBD'@[3w], 'on'/'IN'@[4w],
 ['the'/'DT'@[5w], 'mat'/'NN'@[6w]]]
```

Here, rule 2 ("chunk det+adj+noun") did not find any chunks, since all chunks that matched its tag pattern overlapped with chunks that were already in the hypothesis.

8. ChinkRule and UnChunkRule

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 chunk parser using `UnChunkRule` and `ChinkRule`.

8.1. ChinkRule

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 is in the middle of the chunk, two new chunks are formed; and if the sequence is at the beginning or end of the chunk, one new chunk is formed. These three possibilities are illustrated in the following figure:

	Chink an entire chunk	**Chink the middle of a chunk**	**Chink the 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 "a/DT big/JJ cat/NN"	Chink "big/JJ"	Chink "cat/DT"
Output	a/DT big/JJ cat/NN	[a/DT] big/JJ [cat/NN]	[a/DT big/JJ] cat/NN

A `ChinkRule` chinks anything that matches a given tag pattern. `ChinkRules` are created with the `ChinkRule` constructor, which takes a tag pattern and a description string. For example, the following rule will chink any sequence of tokens whose tags are all "`VBD`" or "`IN`":

```
>>> chink_rule = ChinkRule('<VBD|IN>+',
...                        'Chink sequences of VBD and IN')
```

Remember that `REChunkParser` begins with a chunking hypothesis where nothing is chunked. So before we apply our chink rule, we'll apply another rule that puts the entire sentence in a single chunk:

```
>>> chunkall_rule = ChunkRule('<.*>+',
...                           'Chunk everything')
```

Finally, we can combine these two rules to create a chunk parser:

```
>>> chunkparser = REChunkParser( [chunkall_rule, chink_rule] )

# Parse unchunked_sent with tracing turned on.
>>> chunkparser.parse(unchunked_sent, 1)
Input:
                <DT>  <JJ>  <NN>  <VBD>  <IN>  <DT>  <NN>
Chunk everything:
               {<DT>  <JJ>  <NN>  <VBD>  <IN>  <DT>  <NN>}
Chink sequences of VBD and IN:
               {<DT>  <JJ>  <NN>} <VBD>  <IN> {<DT>  <NN>}

[['the'/'DT'@[0w], 'little'/'JJ'@[1w], 'cat'/'NN'@[2w]],
 'sat'/'VBD'@[3w], 'on'/'IN'@[4w],
 ['the'/'DT'@[5w], 'mat'/'NN'@[6w]]]
```

If a tag pattern matches at multiple overlapping locations, the first match takes precedence.

8.1.1. Cascading Rules

`REChunkParsers` can use any number of `ChunkRules` and `ChinkRules`, in any order. As was discussed in the Section 7, `ChunkRules` only create chunks that do not partially overlap with chunks already in the chunking hypothesis. Similarly, `ChinkRules` only create chinks that do not partially overlap with chinks that are already in the hypothesis.

8.2. UnChunkRule

An `UnChunkRule` removes any chunk that matches a given tag pattern. `UnChunkRule` is very similar to `ChinkRule`; but it will only remove a chunk if the tag pattern matches the entire chunk. In contrast, `ChinkRule` can remove sequences of tokens from the middle of a chunk.

`UnChunkRules` are created with the `UnChunkRule` constructor, which takes a tag pattern and a description string. For example, the following rule will unchunk any sequence of tokens whose tags are all "`NN`" or "`DT`":

```
>>> unchunk_rule = UnChunkRule('<NN|DT>+',
...                            'Unchunk sequences of NN and DT')
```

We can combine this rule with a chunking rule to form a chunk parser:

```
>>> chunk_rule = ChunkRule('<NN|DT|JJ>+',
...                        'Chunk sequences of NN, JJ, and DT')
>>> chunkparser = REChunkParser( [chunk_rule, unchunk_rule] )
```

```
# Parse unchunked_sent with tracing turned on.
>>> chunkparser.parse(unchunked_sent, 1)
Input:
                <DT>  <JJ>  <NN>  <VBD>  <IN>  <DT>  <NN>
Chunk sequences of NN, JJ, and DT:
               {<DT>  <JJ>  <NN>} <VBD>  <IN> {<DT>  <NN>}
Unchunk sequences of NN and DT:
           {<DT>  <JJ>  <NN>} <VBD>  <IN>  <DT>  <NN>

[['the'/'DT'@[0w], 'little'/'JJ'@[1w], 'cat'/'NN'@[2w]],
 'sat'/'VBD'@[3w], 'on'/'IN'@[4w], 'the'/'DT'@[5w], 'mat'/'NN'@[6w]]
```

Note that we would get a different result if we used a `ChinkRule` with the same tag pattern (instead of an `UnChunkRule`), since `ChinkRules` can remove pieces of a chunk:

```
>>> unchink_rule = UnChunkRule('<NN|DT>+',
...                               'Chink sequences of NN and DT')
>>> chunk_rule = ChunkRule('<NN|DT|JJ>+',
...                           'Chunk sequences of NN, JJ, and DT')
>>> chunkparser = REChunkParser( [chunk_rule, chink_rule] )

# Parse unchunked_sent with tracing turned on.
>>> chunkparser.parse(unchunked_sent, 1)
Input:
                <DT>  <JJ>  <NN>  <VBD>  <IN>  <DT>  <NN>
Chunk sequences of NN, JJ, and DT:
               {<DT>  <JJ>  <NN>} <VBD>  <IN> {<DT>  <NN>}
Chink sequences of NN and DT:
                <DT> {<JJ>} <NN>  <VBD>  <IN>  <DT>  <NN>

['the'/'DT'@[0w],
 ['little'/'JJ'@[1w]],
 'cat'/'NN'@[2w]], 'sat'/'VBD'@[3w], 'on'/'IN'@[4w],
 'the'/'DT'@[5w], 'mat'/'NN'@[6w]]
```

9. MergeRule and SplitRule

When constructing complex chunk parsers, it is often convenient to perform operations other than chunking, chinking, and unchunking. In this section, we discuss two more complex rules, which can be used to merge and split chunks.

9.1. MergeRule

`MergeRules` are used to merge two contiguous chunks. Each `MergeRule` is parameterized by two tag patterns: a *left pattern* and a *right pattern*. A `MergeRule` will merge two contiguous chunks C_1 and C_2 if the end of C_1 matches the left pattern, and the beginning of C_2 matches the right pattern. For example, consider the following chunking hypothesis:

```
[the/DT little/JJ][cat/NN]
```

Where C_1 is `[the/DT little/JJ]` and C_2 is `[cat/NN]`. If the left pattern is "`JJ`", and the right pattern is "`NN`", then C_1 and C_2 will be merged to form a single chunk:

```
[the/DT little/JJ cat/NN]
```

`MergeRules` are created with the `MergeRule` constructor, which takes a left tag pattern, a right tag pattern, and a description string. For example, the following rule will merge two contiguous chunks if the first one ends in a determiner, noun or adjective; and the second one begins in a determiner, noun, or adjective:

```
>>> merge_rule = MergeRule('<NN|DT|JJ>', '<NN|DT|JJ>',
...                        'Merge NNs + DTs + JJs')
```

To illustrate this rule, we will use a combine it with a chunking rule that chunks each individual token, and an unchunking rule that unchunks verbs and prepositions:

```
>>> chunk_rule = ChunkRule('<.*>',
...                        'Chunk all individual tokens')
>>> unchunk_rule = UnChunkRule('<IN|VB.*>',
...                            'Unchunk VBs and INs')
>>> rules = [chunk_rule, unchunk_rule, merge_rule]
>>> chunkparser = REChunkParser(rules)

# Parse unchunked_sent with tracing turned on.
>>> chunkparser.parse(unchunked_sent, 1)
Input:
                <DT>  <JJ>  <NN>  <VBD>  <IN>  <DT>  <NN>
Chunk all individual tokens:
               {<DT>}{<JJ>}{<NN>}{<VBD>}{<IN>}{<DT>}{<NN>}
Unchunk VBs and INs:
               {<DT>}{<JJ>}{<NN>} <VBD>  <IN> {<DT>}{<NN>}
Merge NNs + DTs + JJs:
               {<DT>  <JJ>  <NN>} <VBD>  <IN> {<DT>  <NN>}

[['the'/'DT'@[0w], 'little'/'JJ'@[1w], 'cat'/'NN'@[2w]],
 'sat'/'VBD'@[3w], 'on'/'IN'@[4w],
 ['the'/'DT'@[5w], 'mat'/'NN'@[6w]]]
```

9.2. SplitRule

`SplitRule`s are used to split a single chunk into two smaller chunks. Each `SplitRule` is parameterized by two tag patterns: a *left pattern* and a *right pattern*. A `SplitRule` will split a chunk at any point *p*, where the left pattern matches the chunk to the left of *p*, and the right pattern matches the chunk to the right of *p*. For example, consider the following chunking hypothesis:

```
[the/DT little/JJ cat/NN the/DT dog/NN]
```

If the left pattern is "`NN`", and the right pattern is "`DT`", then the chunk will be split in two between "`cat`" and "`dog`", to form two smaller chunks:

```
[the/DT little/JJ cat/NN] [the/DT dog/NN]
```

`SplitRule`s are created with the `SplitRule` constructor, which takes a left tag pattern, a right tag pattern, and a description string. For example, the following rule will split any chunk at a location that has "`NN`" to the left and "`DT`" to the right:

```
>>> split_rule = SplitRule('<NN>', '<DT>',
...                       'Split NN followed by DT')
```

To illustrate this rule, we will use a combine it with a chunking rule that chunks sequences of noun phrases, adjectives, and determiners:

```
# Tokenize a new test text
>>> from nltk.tagger import *
'Bob/NNP saw/VBD the/DT man/NN the/DT cat/NN chased/VBD'
>>> sent='Bob/NNP saw/VBD the/DT man/NN the/DT cat/NN chased/VBD'
['Bob'/'NNP'@[0w], 'saw'/'VBD'@[1w], 'the'/'DT'@[2w], 'man'/'NN'@[3w],
 'the'/'DT'@[4w], 'cat'/'NN'@[5w], 'chased'/'VBD'@[6w]]
>>> ttoks = TaggedTokenizer().tokenize(sent)

# Create the chunk parser
>>> chunk_rule = ChunkRule('<NN.*|DT|JJ>+',
...                          'Chunk sequences of NN, JJ, and DT')
>>> chunkparser = REChunkParser([chunk_rule, split_rule])

# Parse unchunked_sent with tracing turned on.
>>> chunkparser.parse(ttoks, 1)
Input:
                <NNP>  <VBD>  <DT>  <NN>  <DT>  <NN>  <VBD>
Chunk sequences of NN, JJ, and DT:
               {<NNP>} <VBD> {<DT>  <NN>  <DT>  <NN>} <VBD>
Split NN followed by DT:
               {<NNP>} <VBD> {<DT>  <NN>}{<DT>  <NN>} <VBD>

[['Bob'/'NNP'@[0w]],
 'saw'/'VBD'@[1w],
 ['the'/'DT'@[2w], 'man'/'NN'@[3w]],
 ['the'/'DT'@[4w], 'cat'/'NN'@[5w]],
 'chased'/'VBD'@[6w]]
```

10. Transformational Rules

`REChunkParserRules` can be used to define more complex and elaborate kinds of rules. `REChunkParserRule` is more powerful than any of the rules we have discussed; in fact, all of the rules we have discussed are implemented using `REChunkParserRule`.

> **Note:** This section will be completed later; for now, see the reference documentation for more information on `REChunkParserRule`.

11. Defining New Rules

> **Note:** This section will be completed later; if you are interested, you can look at the implementations of `ChunkRule`, `ChinkRule`, `UnChunkRule`, `MergeRule`, and `SplitRule`. They are all relatively straight forward specializations of `REChunkParserRule`.

12. Evaluating Chunk Parsers

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.

From these two sets, we can define five useful metrics:

- *Precision*: What percentage of guessed chunks were correct?
- *Recall*: What percentage of correct chunks were guessed?
- *F Measure*: the harmonic mean of precision and recall.

- *Missed Chunks*: What correct chunks were not guessed?
- *Incorrect Chunks*: What guessed chunks were not correct?

> **Note:** Note that these metrics do not assign any credit for chunks that are "almost" right (e.g., chunks that extend one word too long). It would be possible to design metrics that do assign partial credit for such cases, they would be more complex. We decided to keep our metrics simple, so that it is easy to understand what a given result means.

12.1. ChunkScore

`ChunkScore`is a utility 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.

`ChunkScore`s are created with the `ChunkScore` constructor, which takes no arguments:

```
>>> chunkscore = ChunkScore()
```

The output of a chunk parser can be evaluated using the `score` method:

```
>>> unchunked = unchunk(correct)
>>> guess = chunkparser.parse(unchunked)
>>> chunkscore.score(correct, guess)
```

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.

```
# Import the relevant modules, classes, and functions.
from nltk.token import RETokenizer
from nltk.chunkparser import ChunkedTaggedTokenizer, ChunkScore, unchunk
from nltk.rechunkparser import *
import os

# Location of the training data.
path = "/spd25/cis530/cdrom/resources/data/wsj/tagged/"

# Create the chunk parser that we wish to test.
rule1 = ChunkRule('<DT|JJ|NN>+', "Chunk sequences of DT, JJ, and NN")
chunkparser = REChunkParser( [rule1] )

# Read the WSJ corpus as a list of sentences.  Sentences are
# separated by blank lines and by sequences of equal signs.
sent_tokenizer = RETokenizer('(=======+)?\n\n+', 0)
```

```
    sentences = []
    files = os.listdir(path)
    for file in files:
        print 'Loading %s...' % file
        text = open(path+file).read()
        sentences += sent_tokenizer.tokenize(text, source=file, unit='s')

    # Create a new ChunkScore utility class
    chunkscore = ChunkScore()

    # Test the chunk parser.
    ctt = ChunkedTaggedTokenizer()
    for sentence in sentences:
        correct = ctt.tokenize(sentence.type(), source=sentence.loc())
        unchunked = unchunk(correct)
        guess = chunkparser.parse(unchunked)
        chunkscore.score(correct, guess)

    # Print the results
    print chunkscore
```

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`
- `incorrect`

The `missed` and `incorrect` methods can be especially useful when trying to improve the performance of a chunk parser:

```
    from random import randint

    print 'Chunks missed by the chunk parser:'
    missed = chunkscore.missed()
    for i in range(15):
        print missed[randint(0,len(missed)-1)].type()

    print 'Incorrect chunks returned by the chunk parser:'
    incorrect = chunkscore.incorrect()
    for i in range(15):
        print incorrect[randint(0,len(incorrect)-1)].type()
```

Note: By default, only the first 100 missed chunks and the first 100 incorrect chunks will be remembered by the `ChunkScore`. You can tell `ChunkScore` to record more chunk examples with the `max_fp_examples` (maximum false positive examples) and the `max_fn_examples` (maximum false negative examples) keyword arguments to the `ChunkScore` constructor:

```
>>> chunkscore = ChunkScore(max_fp_examples=1000,
...                         max_fn_examples=1000)
```